Plug-in Development Guide

For Frama-C 30.0 beta (Zinc)

Julien Signoles with Thibaud Antignac, Loïc Correnson, Matthieu Lemerre and Virgile Prevosto

Contents

Foreword

This is the documentation of the Frama-C implementation[1] which aims at helping developers integrate new plug-ins inside this platform.

The content of this document corresponds to the version 30.0 beta (Zinc), released on November 7, 2024, of Frama-C. However the development of Frama-C is still ongoing: features described here may still evolve in the future.

Acknowledgements

We gratefully thank all the people who contributed to this document: Michele Alberti, Gergö Barany, Patrick Baudin, Allan Blanchard, Richard Bonichon, David Bühler, Pascal Cuoq, Zaynah Dargaye, Basile Desloges, Florent Garnier, Pierre-Loïc Garoche, Philippe Herrmann, Boris Hollas, Nikolaï Kosmatov, Jean-Christophe Léchenet, Roma Maliach-Auguste, André Maroneze, Benjamin Monate, Yannick Moy, Anne Pacalet, Valentin Perrelle, Armand Puccetti, Muriel Roger and Boris Yakobowski.

We also thank Johannes Kanig for his Mlpost support[2], the tool formerly used for making figures of this document.

ANR This project has received funding from the French ANR projects CAT (ANR-05-RNTL-00301).

[1] http://frama-c.com

[2] http://mlpost.lri.fr

Introduction 1

Frama-C (Framework for Modular Analyses of C) is a software platform which helps the development of static and dynamic analysis tools for C programs thanks to a plug-in mechanism.

This guide aims at helping developers program within the Frama-C platform, in particular for developing a new analysis or a new source-to-source transformation through a new plug-in. For this purpose, it provides a step-by-step tutorial, a general presentation of the Frama-C software architecture and an overview of the API of the Frama-C kernel. However it does not provide a complete documentation of the Frama-C API and, in particular, it does not describe the API of existing Frama-C plug-ins.

This guide introduces neither the use of Frama-C, which is the purpose of the user manual [3] and of the reference articles [7, 12], nor the use of plug-ins which are documented in separated and dedicated manuals [2, 4, 9, 11, 19]. We assume that the reader of this guide already read the Frama-C user manual and knows the main Frama-C concepts.

The reader of this guide may be either a Frama-C beginner who just finished reading the user manual and wishes to develop their own analysis with the help of Frama-C, an intermediate-level plug-in developer who would like to have a better understanding of one particular aspect of the framework, or a Frama-C expert who wants to remember details about one specific point of the Frama-C development.

Frama-C is fully developed within the OCaml programming language [13]. Motivations for this choice are given in a Frama-C experience report [8]. However this guide *does not* provide any introduction to this programming language: the World Wide Web already contains plenty of resources for OCaml developers (see for instance `https://ocaml.org/`).

1.1 About this document

To ease reading, section heads may state the category of readers they are intended for and a set of prerequisites.

Appendix A references all the changes made to this document between successive Frama-C releases.

In the index, page numbers written in bold italics (e.g. ***1***) reference the defining sections for the corresponding entries while other numbers (e.g. 1) are less important references. Furthermore, the name of each OCaml value in the index corresponds to an actual Frama-C value.

The most important paragraphs are displayed inside orange boxes like this one. A plug-in developer ***must*** *follow them very carefully.*

There are numerous code snippets in this document. Beware that copy/pasting them from the PDF to your favorite text editor may prevent your code from compiling, because the PDF text can contain non-ASCII characters.

1.2 Outline

This guide is organised in three parts.

Chapter 2 is a step-by-step tutorial for developing a new plug-in within the Frama-C platform. At the end of this tutorial, a developer should be able to extend Frama-C with a simple analysis available as a Frama-C plug-in.

Chapter 3 presents the Frama-C software architecture.

Chapter 4 details how to use all the services provided by Frama-C in order to develop a fully integrated plug-in.

Tutorial 2

Target readers: *beginners.*

This chapter aims at helping a developer to write their first Frama-C plug-in. At the end of the tutorial, any developer should be able to extend Frama-C with a simple analysis available as a Frama-C plug-in. This chapter was written as a step-by-step explanation on how to proceed towards this goal. It will get you started, but it does not tell the whole story. You will get it with your own experiments, and by reading the other chapters of this guide as needed.

Section 2.1 provides some tips for setting up a development environment for Frama-C. Section 2.2 shows what a plug-in looks like. Then Section 2.3 explains the basis for writing a standard Frama-C plug-in, while Section 2.4 details how to interact with Frama-C and other plug-ins to implement analyzers of C programs.

2.1 Development Environment

It is easy to develop a plug-in for Frama-C with any IDE, as long as it supports the OCaml language. This includes (but is not limited to) Emacs or Vim with the Merlin[1] tool, or VS Code with the *OCaml platform* extension. The last is probably the easiest to setup for a beginner in OCaml.

Most modern IDEs support (directly or indirectly, via Merlin) OCaml-LSP[2], which is an implementation of LSP (*Language Server Protocol*) for OCaml.

Concerning code formatting, the Frama-C team currently uses the `ocp-indent` opam package for code indentation. Consider installing it if you want to ensure your code follows the same conventions.

Overall, it is **strongly** suggested to use an OCaml-aware IDE and take the time to set it up. Plug-ins use several different parts of the Frama-C API, and a properly setup IDE greatly improves productivity, offering features such as auto-completion, type checking, syntax highlighting, and code navigation.

2.2 What Does a Plug-in Look Like?

Figure 2.1 shows how a plug-in can integrate with the Frama-C platform. This tutorial focuses on specific parts of this figure.

The implementation of the plug-in is provided inside a specific directory. The plug-in registers with the Frama-C platform through kernel-provided registration points. These registrations are performed through hooks (by applying a function or a functor). For instance, the next section shows how to:

[1] https://ocaml.github.io/merlin

[2] https://github.com/ocaml/ocaml-lsp

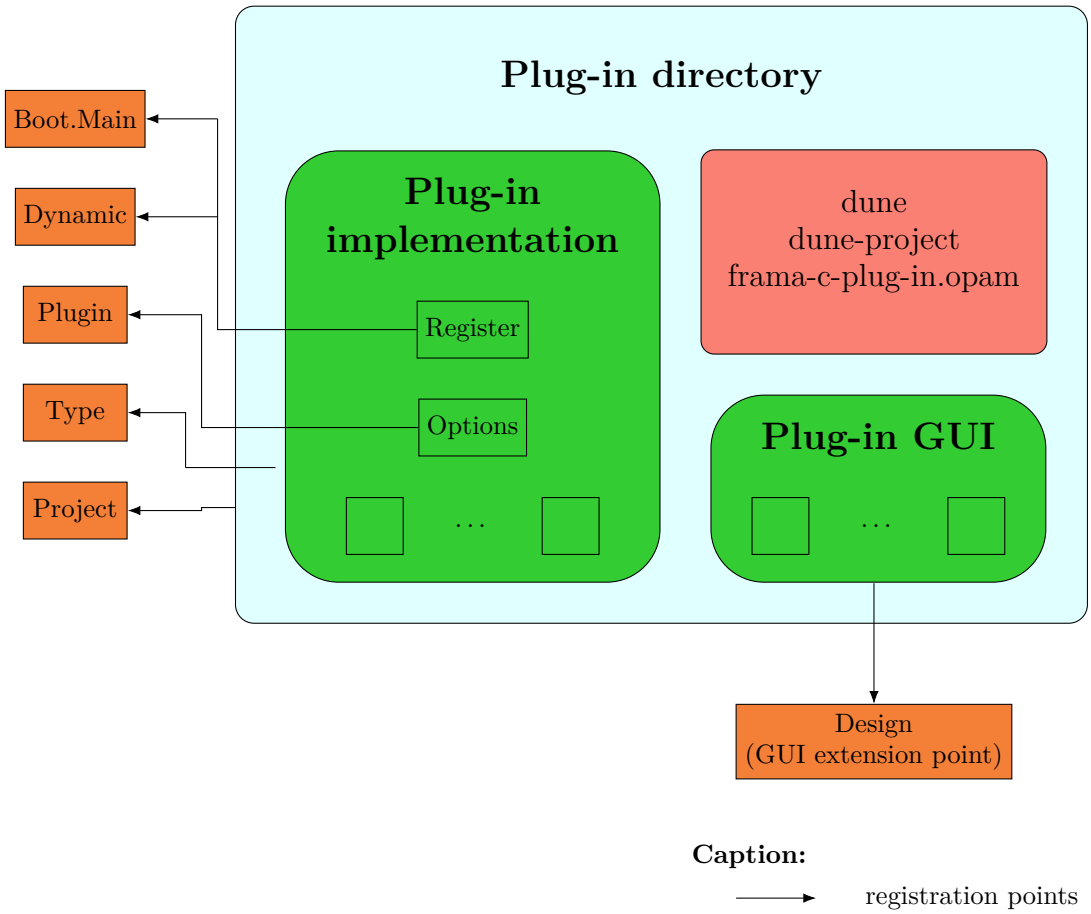


Figure 2.1: Plug-in Integration Overview.

– extend the Frama-C entry point thanks to the function `Boot.Main.extend` if you want to run plug-in specific code whenever Frama-C is executed;
– use specific plug-in services provided by the module `Plugin`, such as adding a new Frama-C option.

2.3 The Hello plug-in

This simple plug-in illustrates how to create your own plug-in basically with several aspects of the Frama-C framework: building the plugin and installing the plug-in, registration, getting command-line options, console output, testing and documentation. (In case of difficulty, it is explained at the end of this section how to generate the whole plug-in.)

2.3.1 Creating a new plug-in

A plug-in is built using Dune[3]. It is composed minimally of the following files:

– a `dune-project` file that describes the project;

[3] https://dune.build

– a `dune` file that describes the build of the project;
– a `<my_plugin>.ml` file that defines the API of the plugin (which can be empty).

For example, for the Hello plugin:

File **./dune-project**

```
(lang dune 3.13)
(using dune_site 0.1)

(name frama-c-hello)
(package (name frama-c-hello))
```

The `dune_site` feature[4] mentioned in this file allows us to rely on `dune` to install the plug-in in a place where Frama-C will find it and load it at runtime.

File **./dune**

```
(library
 (name Hello)
 (public_name frama-c-hello.core)
 (flags -open Frama_c_kernel :standard)
 (libraries frama-c.kernel)
)

(plugin
 (optional)
 (name hello)
 (libraries frama-c-hello.core)
 (site (frama-c plugins)))
```

`hello.ml` is just an empty file.
Then the plugin can be built using the following command:
`dune build`
If Dune is installed, this should compile the project successfully. Note that Dune emits messages *during* compilation, but erases them afterwards. In case of success, there will be no visible output at the end. Note that this behavior can be configured with Dune's option `--display`.

*Dune always looks for `dune-project` files in the parent directories, so make sure that your Hello directory is not inside another one containing a Dune project (*e.g. *if you are running this directly from the* Frama-C *source archive), otherwise* `dune build` *may fail. You can always add option "`--root .`", or create an empty `dune-workspace` file in the current directory to force Dune to ignore parent directories.*

Note a few details about the naming conventions:

– in the `dune-project` file, the name of the plugin is `frama-c-<my-plugin>`
– in the `dune` file, the name of:
  – the library is `my_plugin`;
  – the public name of the library is `frama-c-<my-plugin>.core` (dune project name core);
  – the name of the plugin is `<my-plugin>`;
  – the plugin must at least include the library `frama-c-<my-plugin>.core`.

Of course, for now, our plug-in does nothing, so let us change that.

[4] https://dune.readthedocs.io/en/stable/sites.html

2.3.2 A Very Simple Plug-in

Let us add a simple file to our plug-in:

File **./hello_world.ml**

```
let run () =
  try
    let chan = open_out "hello.out" in
    Printf.fprintf chan "Hello, world!\n";
    flush chan;
    close_out chan
  with Sys_error _ as exc →
    let msg = Printexc.to_string exc in
    Printf.eprintf "There was an error: %s\n" msg

let () = Boot.Main.extend run
```

This file defines a simple function that writes a message to an output file, then registers the function `run` as an entry point. Frama-C will call it among the other plug-in entry points if the plug-in is loaded.

The plug-in is compiled the same way as previously explained in 2.3.1. Then, one can execute the script using the command:

```
dune exec -- frama-c
```

Executing this command creates a `hello.out` file in the current directory.

You can think of `dune exec --` *as a wrapper for* `frama-c` *which adds your plug-in to those already present in* Frama-C*. It operates on a local Dune sandbox, allowing you to test it without risking "polluting" your installed* Frama-C*.*

2.3.3 Registering a Plug-in in Frama-C

To make this script better integrated into Frama-C, its code must register itself as a plug-in. Registration provides general services, such as outputting to the Frama-C console and extending Frama-C with new command-line options.

Registering a plug-in is achieved through the use of the `Plugin.Register` functor:

File **./hello_world.ml**

```
let help_msg = "output a warm welcome message to the user"

module Self = Plugin.Register
    (struct
     let name = "hello world"
     let shortname = "hello"
     let help = help_msg
    end)

let run () =
  try
    let chan = open_out "hello.out" in
    Printf.fprintf chan "Hello, world!\n";
    flush chan;
    close_out chan
  with Sys_error _ as exc →
    let msg = Printexc.to_string exc in
    Printf.eprintf "There was an error: %s\n" msg

let () = Boot.Main.extend run
```

The argument to the `Plugin.Register` functor is a module with three values:

- `name` is an arbitrary, non-empty string containing the full name of the module.
- `shortname` is a small string containing the *short name* of the module, to be used as a prefix for all plug-in options[5]. It cannot contain spaces.
- `help` is a string containing free-form text, with a description of the module.

Visible results of the registration include:

- "hello world" appears in the list of available plug-ins; you can check it by running `dune exec -- frama-c -plugins`;
- default options for the plug-in work, including the inline help (available with `dune exec -- frama-c -hello-help`).

2.3.4 Displaying Messages

The signature of the module `Self` obtained by applying `Plugin.Register` is `General_services`. One of these general services is logging, *i.e.* message display. In Frama-C, one should never attempt to write messages directly to `stderr` or `stdout`: use the general services instead[6].

File **./hello_world.ml**

```
let help_msg = "output a warm welcome message to the user"

module Self = Plugin.Register
    (struct
     let name = "hello world"
     let shortname = "hello"
     let help = help_msg
    end)

let run () =
  Self.result "Hello, world!";
  let product =
    Self.feedback ~level:2 "Computing the product of 11 and 5...";
    11 * 5
  in
  Self.result "11 * 5 = %d" product

let () = Boot.Main.extend run
```

Running this script yields the following output:

```
$ dune exec -- frama-c
[hello] Hello, world!
[hello] 11 * 5 = 55
```

The `result` routine is the function to use to output results of your plug-in. The Frama-C output routines take the same arguments as the OCaml function `Format.printf`[7].

Function `feedback` outputs messages that show progress to the user. In this example, we decided to emit a feedback message with a log level of 2, because we estimated that in most cases the user is not interested in seeing the progress of a fast operation (simple multiplication). The default log level is 1, so by default this message is not displayed. To see it, the verbosity of the `hello` plug-in must be increased:

[5] Frama-C does not enforce that all plug-in options are prefixed with its shortname, but it is customary to do so.

[6] However, writing to a new file using standard OCaml primitives is OK.

[7] The Format module is part of the OCaml standard library and provides advanced pretty-printing features. If you are not familiar with it, consider grepping some Frama-C messages to get a feel for how to use it.

```
$ dune exec -- frama-c -hello-verbose 2
[hello] Hello, world!
[hello] Computing the product of 11 and 5...
[hello] 11 * 5 = 55
```

For a comprehensive list of logging routines and options, see Section 4.6.

2.3.5 Adding Command-Line Options

We now extend the `hello world` plug-in with new options.

If the plug-in is installed (globally, with `dune install`, or locally, with `dune exec`), it will be loaded and executed on *every* invocation of `frama-c`, which is not how most plug-ins work. To change this behavior, we add a boolean option, set by default to false, that conditionally enables the execution of the main function of the plug-in. The usual convention for the name of this option is the short name of the module, without suffix, *i.e.* `-hello` in our case.

We also add another option, `-hello-output`, that takes a string argument. When set, the hello message is displayed in the file given as argument.

File **./hello_world.ml**

```
let help_msg = "output a warm welcome message to the user"

module Self = Plugin.Register
    (struct
     let name = "hello world"
     let shortname = "hello"
     let help = help_msg
    end)

module Enabled = Self.False
    (struct
     let option_name = "-hello"
     let help = "when on (off by default), " ^ help_msg
    end)

module Output_file = Self.String
    (struct
     let option_name = "-hello-output"
     let default = "-"
     let arg_name = "output-file"
     let help =
       "file where the message is output (default: output to the console)"
    end)

let run () =
  try
    if Enabled.get() then
      let filename = Output_file.get () in
      let output msg =
        if Output_file.is_default() then
          Self.result "%s" msg
        else
          let chan = open_out filename in
          Printf.fprintf chan "%s\n" msg;
          flush chan;
          close_out chan;
      in
```

```
    output "Hello, world!"
  with Sys_error _ as exc →
    let msg = Printexc.to_string exc in
    Printf.eprintf "There was an error: %s\n" msg

let () = Boot.Main.extend run
```

Registering these new options is done by calling the `Self.False` and `Self.String` functors, which respectively create a new boolean option whose default value is false and a new string option with a user-defined default value (here, `"-"`). The values of these options are obtained *via* `Enabled.get ()` and `Output_file.get ()`.

With this change, the hello message is displayed only if Frama-C is executed with the `-hello` option.

```
$ dune exec -- frama-c
$ dune exec -- frama-c -hello
[hello] Hello, world!
$ dune exec -- frama-c -hello -hello-output hello.out
$ ls hello.out
hello.out
```

These new options also appear in the inline help for the hello plug-in:

```
$ dune exec -- frama-c -hello-help
...
***** LIST OF AVAILABLE OPTIONS:

-hello         when on (off by default), output a warm welcome message
               to the user (opposite option is -no-hello)
-hello-output <output-file> file where the message is output (default:
               output to the console)
...
```

2.3.6 About the plug-in build process

As mentioned before, each plug-in needs a module declaring its public API. In our examples, this was file `hello.ml`, which was left empty. To make it more explicit to future users of our plug-in, it is customary to add a comment such as the following:

File **./hello.ml**

```
(** Hello World plug-in.

    No function is exported. *)
```

Note that, to avoid issues, the name of each compilation unit (here `hello_world`) must be different from the plug-in name set in the `dune` file (here `hello`), from any other plug-in names (*e.g.* `eva`, `wp`, etc.) and from any other Frama-C kernel OCaml files (*e.g.* `plugin`).

The module name chosen by a plug-in (here `hello`) is used by Dune to *pack* that plug-in; any functions declared in it become part of the plug-in's public API. They become accessible to other plug-ins and need to be maintained by the plug-in developer. Leaving it empty avoids exposing unnecessary implementation details.

Inside the plug-in's directory, `dune build` compiles it and creates the packed module. It can be installed along with the other Frama-C plug-ins using `dune install`.

You can then just launch `frama-c` (without any options), and the `hello` plug-in will be automatically loaded. Check it with the command `frama-c -hello-help`.

You can uninstall it by running `dune uninstall`.

Splitting your source files

Here is a slightly more complex example where the plug-in has been split into several files. Usually, plug-in registration through `Plugin.Register` should be done at the bottom of the module hierarchy, while registration of the `run` function through `Boot.Main.extend` should be at the top. The three following files completely replace the `./hello_world.ml` from the previous section. Modules are directly called by their name in the classical OCaml way.

File **./hello_options.ml**

```
let help_msg = "output a warm welcome message to the user"

module Self = Plugin.Register
    (struct
     let name = "hello world"
     let shortname = "hello"
     let help = help_msg
    end)

module Enabled = Self.False
    (struct
     let option_name = "-hello"
     let help = "when on (off by default), " ^ help_msg
    end)

module Output_file = Self.String
    (struct
     let option_name = "-hello-output"
     let default = "-"
     let arg_name = "output-file"
     let help =
       "file where the message is output (default: output to the console)"
    end)
```

File **./hello_print.ml**

```
let output msg =
  try
    let filename = Hello_options.Output_file.get () in
    if Hello_options.Output_file.is_default () then
      Hello_options.Self.result "%s" msg
    else
      let chan = open_out filename in
      Printf.fprintf chan "%s\n" msg;
      flush chan;
      close_out chan
  with Sys_error _ as exc →
    let msg = Printexc.to_string exc in
    Printf.eprintf "There was an error: %s\n" msg
```

File **./hello_run.ml**

```
let run () =
  if Hello_options.Enabled.get() then
    Hello_print.output "Hello, world!"

let () = Boot.Main.extend run
```

The plug-in can be tested again by running:

```
$ dune build
$ dune install
<...>
$ frama-c -hello -hello-output hello.out
$ more hello.out
Hello, world!
```

However, this does not consist in a proper test *per se*. The next section presents how to properly test plug-ins.

2.3.7 Testing your Plug-in

Frama-C supports non-regression testing of plug-ins. This is useful to check that further plug-in modifications do not introduce new bugs. The tool allowing to perform the tests is called ptests .

Historically, `ptests` was developed before Frama-C *used Dune. It was adapted to Dune to maintain backwards compatibility, but new plug-ins may prefer using Dune's test sytem directly.*

This is the general layout of the `tests` directory in a Frama-C plug-in; each file will be explained later.

```
<plug-in directory>
+- tests
  +- ptests_config
  +- test_config
  +- suite1
    +- test1.c
    +- ...
    +- oracle
      +- test1.res.oracle
      +- test1.err.oracle
      +- ...
    +- result
      +- test1.0.exec.wtests
      +- ...
 +- ...
```

Inside the `tests` directory, `ptests_config` lists the *test suites*, *i.e.* directories containing tests.

File **./tests/ptests_config**

```
DEFAULT_SUITES= hello
```

Small plug-ins typically have a single test directory with the plug-in name.

Then, a default configuration must be provided for the tests. This is done by adding a `test_config` file to the `tests` directory.

File **./tests/test_config**

```
PLUGIN: hello
```

This configuration must include the plug-ins required by the test; usually, the plug-in itself, but also other plug-ins on which it depends. The plug-in name is the one provided in the `plugin` section of the `dune` file.

For non-regression testing, the current behavior of a program is taken as the oracle against which future versions will be tested. In this tutorial, the test will be about the correct `Hello, world!` output made by the option `-hello` of the plug-in.

Each test file should contain a `run.config` comment with test directives and the C source code used for the test (note: there are other ways to declare and control tests, as detailed in Section 4.3.2).

For this tutorial, no actual C code is needed, so `./tests/hello/hello_test.c` will only contain the run.config header:

File **./tests/hello/hello_test.c**

```
/* run.config
   OPT: -hello
 */
```

In this file, there is only one directive, `OPT: -hello`, which specifies that Frama-C will set option `-hello` when running the test. A look at Section 4.3.4 gives you an idea of the kinds of directives which can be used to test plug-ins.

Once `run.config` has been configured, it becomes possible to generate Dune test files via the ptests tool:

```
$ frama-c-ptests
Test directory: tests
Total number of generated dune files: 2
```

This must be done each time a test configuration is modified or a test file or directory is added.

Then, one can execute the tests and get the output generated by the plug-in, by running `dune build @ptests`. Note that if you forget the intermediate step of running `frama-c-ptests`, you may end up with the following error:

```
Error: Alias "ptests" specified on the command line is empty.
It is not defined in tests or any of its descendants.
```

But with the `dune` files generated by `frama-c-ptests`, you can run the tests:

```
$ dune build @ptests
Files ../oracle/hello_test.res.oracle and hello_test.res.log differ
% dune build @"tests/hello/result/hello_test.0.exec.wtests": diff
    failure on diff:
(cd _build/default/tests/hello/result && diff --new-file "
   hello_test.res.log"
"../oracle/hello_test.res.oracle")
File "tests/hello/oracle/hello_test.res.oracle", line 1, characters 0-0:

diff --git a/_build/default/tests/hello/oracle/hello_test.res.oracle
         b/_build/default/tests/hello/result/hello_test.res.log
index e69de29..5f6ab23 100644
--- a/_build/default/tests/hello/oracle/hello_test.res.oracle
+++ b/_build/default/tests/hello/result/hello_test.res.log
@@ -0,0 +1,2 @@
+[kernel] Parsing hello_test.c (with preprocessing)
+[hello] Hello, world!
```

There is a lot of information in the output. The relevant parts can be summarized as:

- Dune runs its tests inside a *sandboxed* environment, in directory `_build/default`, which is (approximately) a copy of the plug-in file tree;
- Dune compared two files, none of which existed before running the test: `result/hello_test.res.log` and `oracle/hello_test.res.oracle`;
- The last lines (which should be colored, if your terminal supports colors) show the textual difference between the expected and actual outputs.

The first file, `result/hello_test.res.log`, is the *actual* output of the execution of the test command. The second file, `oracle/hello_test.res.oracle`, is the *expected* output of the test.

The result file is re-generated each time the test is run. The oracle file, however, is supposed to exist beforehand (unless the test produces no output).

In reality, there are 2 pairs of files for each test: a pair `.res.{log,oracle}` and another `.err.{log,oracle}`. The first one contains results sent to the standard output (*stdout*), while the second one contains messages sent to the standard error (*stderr*). In our example, the error output is empty, so it generates no differences. Note that Dune only outputs messages in case of errors, *i.e.* tests producing the expected result are silent.

Finally, concerning the actual diff in the test (last two lines), the first line (starting with `[kernel]`) is emitted by the Frama-C kernel, and the second one is our plug-in's result, as expected.

Once you have verified the actual output is the one *you* expected, you can *promote* it to the status of "official oracle" for future non-regression tests, by running:

```
$ dune promote
```

Note, however, that if the oracles do not exist, they must be created:

```
$ frama-c-ptests -create-missing-oracles tests
$ dune promote
```

The option `-create-missing-oracles` always creates both result and error oracles. Most of the time, however, only the former is useful. After promoting tests, it is useful to remove empty oracles:

```
$ frama-c-ptests -remove-empty-oracles tests
```

The new oracle should be committed to source control, for future testing.

Once your plug-in has test files, the `dune build` *command presented earlier will not only compile your plug-in, but also run its tests. Therefore, if you want to simply compile it, you will have to run* `dune build @install` *instead. Despite the name, the command will* not *install your plug-in, it will only build and collect all files necessary for its installation.*

Now, let's introduce an error. Assume the plug-in has been modified as follows:

File **./hello_run.ml**

```
let run () =
  if Hello_options.Enabled.get() then
    Hello_print.output "Hello world!" (* removed comma *)

let () = Boot.Main.extend run
```

We assume that "`Hello, world!`" has been unintentionally changed to "`Hello world!`". Running these commands:

```
$ dune build @install
<...>
$ dune build @ptests
Files ../oracle/hello_test.res.oracle and hello_test.res.log are different
% dune build @"tests/hello/result/hello_test.0.exec.wtests": diff failure on
    diff:
 (cd _build/default/tests/hello/result && diff --new-file "hello_test.res.log" "
    ../oracle/hello_test.res.oracle")
File "tests/hello/oracle/hello_test.res.oracle", line 1, characters 0-0:
diff --git a/_build/default/tests/hello/oracle/hello_test.res.oracle b/_build/
    default/tests/hello/result/hello_test.res.log
index 5f6ab2389a..cf2e5c010c 100644
--- a/_build/default/tests/hello/oracle/hello_test.res.oracle
+++ b/_build/default/tests/hello/result/hello_test.res.log
@@ -1,2 +1,2 @@
```

```
 [kernel] Parsing hello_test.c (with preprocessing)
-[hello] Hello, world!
+[hello] Hello world!
```

displays the differences (à la `diff`) between the current execution and the saved oracles. Here, the diff clearly shows that the only difference is the missing comma in the generated message due to our (erroneous) modification. After fixing the OCaml code, running again the previous commands shows that all test cases are successful.

You may check other Frama-C plug-ins as examples of how to integrate a plug-in with ptests. Small plug-ins such as `Report` and `Variadic` are good examples (see directories `src/plugins/report/tests` and `src/plugins/variadic/tests`). Please note Frama-C offers no particular support for other kinds of testing purposes, such as test-driven development (TDD)[8]. Additional information about plug-in testing is available in Sections 4.3.

Summary of Testing Operations

Here's a summarized list of operations in order to add a new test:

1. Create a test case (`.c` or `.i` file in `tests/<suite>`).
2. Add a `run.config` header to the test.
3. `frama-c-ptests -create-missing-oracles`
4. `dune build @ptests`
5. Manually inspect oracle diffs to check that they are ok.
6. `dune promote`
7. `frama-c-ptests -remove-empty-oracles`

Operations to perform when modifying the plug-in code:

1. `dune build @install`
2. `dune build @ptests`
3. If there are expected diffs, run `dune promote`; otherwise, fix code and re-run the first steps.

2.3.8 Documenting your Source Code

One can generate the documentation of a plugin using the command:

`dune build @doc`

This relies on odoc and requires the plug-in to be documented following the odoc guidelines (check https://ocaml.github.io/odoc for details).

We show here how the Hello plug-in could be slightly documented and use odoc features such as @-tags and cross references. First, we modify the `hello.ml` file to export all inner modules, otherwise odoc will not generate documentation for them:

File **./hello.ml**

```
(** Hello World plug-in.
   All modules are exported to illustrate documentation generation by odoc. *)

module Hello_run = Hello_run
module Hello_options = Hello_options
module Hello_print = Hello_print
```

Then, we add some documentation tags to our modules:

File **./hello_options.ml**

[8] For instance, one required feature for TDD that `ptests` does not support, is to *force* the user to manually create `./tests/*/oracle/*.oracle` files before running a new test.

```
(** This module contains the possible command line options
   for the Hello plug-in.
   @author Anne Onymous
   @see < http://frama-c.com/download/frama-c-plugin-development-guide.pdf>
      Frama-C Developer Manual, Tutorial
*)

(** Content of the welcome message. *)
let help_msg = "output a warm welcome message to the user"

(** Registration of the plug-in to Frama-C. *)
module Self = Plugin.Register
    (struct
     let name = "hello world"
     let shortname = "hello"
     let help = help_msg
    end)

(** Enabling of the plug-in. *)
module Enabled = Self.False
    (struct
     let option_name = "-hello"
     let help = "when on (off by default), " ^ help_msg
    end)

(** Output of the plug-in. *)
module Output_file = Self.String
    (struct
     let option_name = "-hello-output"
     let default = "-"
     let arg_name = "output-file"
     let help =
       "file where the message is output (default: output to the console)"
    end)
```

File **./hello_print.ml**

```
(** This module contains the printing method of the Hello plug-in.
   @author Anne Onymous
   @see < http://frama-c.com/download/frama-c-plugin-development-guide.pdf>
      Frama-C Developer Manual, Tutorial
*)

(** Outputs a message to the output selected in
      {!module:Hello_options.Output_file}.
   @param msg Message to output.
   @raise Sys_error if filesystem error.
*)
let output msg =
  try
    let filename = Hello_options.Output_file.get () in
    if Hello_options.Output_file.is_default () then
      Hello_options.Self.result "%s" msg
    else
      let chan = open_out filename in
      Printf.fprintf chan "%s\n" msg;
      flush chan;
```

```
    close_out chan
  with Sys_error _ as exc →
    let msg = Printexc.to_string exc in
    Printf.eprintf "There was an error: %s\n" msg
```

File **./hello_run.ml**

```
(** This module contains the main control logic of the Hello plug-in.
   @author Anne Onymous
   @see < http://frama-c.com/download/frama-c-plugin-development-guide.pdf>
      Frama-C Developer Manual, Tutorial
*)

(** Controls the output of a given message by
      {!val:Hello_print.output} depending on the state of
      {!module:Hello_options.Enabled}.
*)
let run () =
  if Hello_options.Enabled.get() then
    Hello_print.output "Hello, world!"

(** Definition of the entry point of the hello plug-in. *)
let () = Boot.Main.extend run
```

Running `dune build @doc` will generate documentation files in `./_build/default/_doc/_html`. Open `index.html` in your browser and navigate to see them. Note that odoc may report some warnings due to absence of annotation data for Frama-C's modules:

```
Warning: Couldn't find the following modules:
  Frama_c_kernel Frama_c_kernel__Plugin
```

This should not prevent the generation of documentation for the library itself; but links to modules such as `Enabled` and `Output_file` will not work.

2.3.9 Conclusion

This simple tutorial now comes to its end. It focused on the standard features of architectures and interfaces of Frama-C plug-ins. A companion archive `hello.tar.gz` is available in the download section of the Frama-C website[9]. The next tutorial will make you dive in C analysis.

2.4 The ViewCfg plug-in

In this section, we create a new ViewCfg plug-in that computes the control flow graph of a function and outputs it in the DOT format. Through its implementation, we explain some of Frama-C APIs such as how to visit an AST[10], how to hook a plug-in, how to interface a plug-in with other plug-ins, and how to make a plug-in usable in a multi-project setting (which also allows it to save/load data).

This section assumes the reader is already familiar with the basics of plug-ins for Frama-C as covered by the Hello plug-in in the previous section.

[9] The direct link is: https://www.frama-c.com/download/hello.tar.gz.

[10] Abstract Syntax Tree

2.4.1 Visiting the AST

Writing an analysis for C programs is the primary purpose of a Frama-C plug-in. That usually requires visiting the AST to compute information for some C constructs. There are two different ways of doing that in Frama-C:

- through a direct recursive descent; or
- by using the Frama-C visitor.

The first case is usually better if you have to compute information for most C constructs, while the latter is better if only few C constructs are interesting or if you have to write a program transformation. Of course, it is also possible to combine both ways to tune it to specific needs.

Pretty-printing with direct recursive descent

Frama-C already has a function to pretty-print statements (namely `Printer.pp_stmt`), but it is not suitable for us, as it will recursively print substatements of compound statements (blocks, if, while, ...), while we only want to pretty-print the node representing the current statement: substatements will be represented by other nodes. Thus we will use the following small function:

```
open Cil_types

let print_stmt out = function
  | Instr i → Printer.pp_instr out i
  | Return _ → Format.pp_print_string out "< return> "
  | Goto _ → Format.pp_print_string out "< goto> "
  | Break _ → Format.pp_print_string out "< break> "
  | Continue _ → Format.pp_print_string out "< continue> "
  | If (e,_,_,_) → Format.fprintf out "if %a" Printer.pp_exp e
  | Switch(e,_,_,_) → Format.fprintf out "switch %a" Printer.pp_exp e
  | Loop _ → Format.fprintf out "< loop> "
  | Block _ → Format.fprintf out "< block> "
  | UnspecifiedSequence _ → Format.fprintf out "< unspecified sequence> "
  | TryFinally _ | TryExcept _ | TryCatch _ → Format.fprintf out "< try> "
  | Throw _ → Format.fprintf out "< throw> "
```

The `Cil_types` module contains the definition of the AST of a C program, like constructors `Cil_types.Instr`, `Cil_types.Return`, and so on which are of type `Cil_types.stmtkind`. The `Printer` module contains functions that print the different Cil types. The documentation of these module is available on the Frama-C website[11], or by typing `make doc` in the Frama-C source distribution.

Creating the graphs with a visitor

In order to create our output, we must make a pass through the whole AST. An easy way to do that is to use the Frama-C visitor mechanism. A visitor is a class with one method per type of the AST, whose default behavior is to just call the method corresponding to each of its children. By inheriting from the visitor, and redefining some of the methods, one can perform actions on selected parts of the AST, without the need to traverse the AST explicitly.

```
class print_cfg out = object
  inherit Visitor.frama_c_inplace
```

Here we used the so-called "in place" visitor, which should be used for read-only access to the AST. When performing code transformations, a "copy" visitor should be used, that creates a new project (see section 4.14.4).

There are three kinds of nodes where we have something to do. First, at the file level, we create the whole graph structure.

[11] From https://www.frama-c.com/api/index.html

```
  method! vfile _ =
    Format.fprintf out "digraph cfg {\n";
    Cil.DoChildrenPost (fun f → Format.fprintf out "}\n%!"; f)
```

`Cil.DoChildrenPost` is one of the possible `visitActions`, that tells the visitor what to do after the function is executed. With `DoChildrenPost func`, the `func` argument is called once, after all children have been executed. Therefore, we use it to close the curly braces after all functions have been printed in the file.

Then, for each function, we encapsulate the CFG in a subgraph, and do nothing for the other globals.

```
  method! vglob_aux g =
    match g with
    | GFun(f,_) →
      Format.fprintf out " subgraph cluster_%a {\n" Printer.pp_varinfo f.svar;
      Format.fprintf out " graph [label=\"%a\"];\n" Printer.pp_varinfo f.svar;
      Cil.DoChildrenPost (fun g → Format.fprintf out " }\n"; g)
    | _ → Cil.SkipChildren
```

`Cil.SkipChildren` tells the visitor not to visit the children nodes, which makes it more efficient[12].

Last, for each statement, we create a node in the graph, and create the edges toward its successors:

```
  method! vstmt_aux s =
    Format.fprintf out " s%d [label=%S];\n"
      s.sid (Pretty_utils.to_string print_stmt s.skind);
    List.iter (fun succ → Format.fprintf out " s%d → s%d;\n" s.sid succ.sid)
      s.succs;
    Cil.DoChildren
```

This code could be optimized, for instance by replacing the final `DoChildren` by `SkipChildren` for statements that cannot contain other statements, like `Instr`, and avoid visiting the expressions.

Finally, we close the `object` definition:

```
end
```

Hooking into Frama-C

Now we need to ensure the code is called at the appropriate time when Frama-C is run. Note that if we simply add a function at the toplevel (*i.e.* `let () = run ()`), it will *not* work, because Frama-C will not have had the time to parse the sources and produce its AST (used by `Ast.get ()`). For more details about initialization issues, see Section 4.11.

```
let run () =
  let chan = open_out "cfg.dot" in
  let fmt = Format.formatter_of_out_channel chan in
  Visitor.visitFramacFileSameGlobals (new print_cfg fmt) (Ast.get ());
  close_out chan

let () = Boot.Main.extend run
```

Now, since Frama-C uses Dune, this code needs to be integrated as a Dune project, as has been done in the Hello tutorial. We need to create a new directory. In it, we will put all of the code seen so far, in an `.ml` file. We will then add the corresponding `dune` and `dune-project` files:

File **dune**

```
(library
 (name ViewCfg)
```

[12] In a copying visitor, `Cil.JustCopy` should have been used instead.

```
 (public_name frama-c-view-cfg.core)
 (flags -open Frama_c_kernel :standard)
 (libraries frama-c.kernel))

(plugin
 (name view-cfg)
 (libraries frama-c-view-cfg.core)
 (site (frama-c plugins)))
```

File **dune-project**

```
(lang dune 3.13)
(using dune_site 0.1)

(name frama-c-view-cfg)
(package (name frama-c-view-cfg))
```

Finally, we need an (empty) interface file, called `ViewCfg.ml`, in the same directory.
With all of this, we can compile the project with:

```
dune build
```

And run it as a Frama-C plugin:

```
dune exec -- frama-c [options] file.c [file2.c]
```

And the graph can be visualized with an external tool, such as *dotty* or *xdot*.

```
dotty cfg.dot
```

This produces a graph like in Figure 2.2

Further improvements

There are many possible enhancements to this code:

- There is a bug when trying to print statements that contain strings (such as `printf("Hello\n")`, due to double quotes. Such statements must be protected using the `"%S"` Format directive;
- The plug-in could be properly registered as such, allowing it to accept command-line options, for instance to compute the control flow graph of a single function given as argument;
- The graphs could be fancier, in particular by distinguishing between branching nodes and plain ones, or showing block entries and exits; or linking call sites to the called functions.

We will concentrate on another extension, which is to reuse the analysis of the `Eva` plug-in to color unreachable nodes. To do so, because we will combine different plug-ins, we need to ensure their correct ordering. This requires the definition of some command-line options for our plug-in.

2.4.2 Plug-in registration and command-line options

We have already seen how to register options in the previous "Hello" tutorial. We now apply these principles to the ViewCfg plug-in.

```
module Self = Plugin.Register(struct
   let name = "control flow graph"
   let shortname = "viewcfg"
   let help = "control flow graph computation and display"
 end)

module Enabled = Self.False(struct
```

File **test.c**

```
void f(int g)
{
  g++;
  g--;
}

int main(int argc, char **argv)
{
  int i = 3;

  if( i > 0)
    {
      while(--i);
    }
  else
    {
      f(3);
    }
  return 0;
}
```

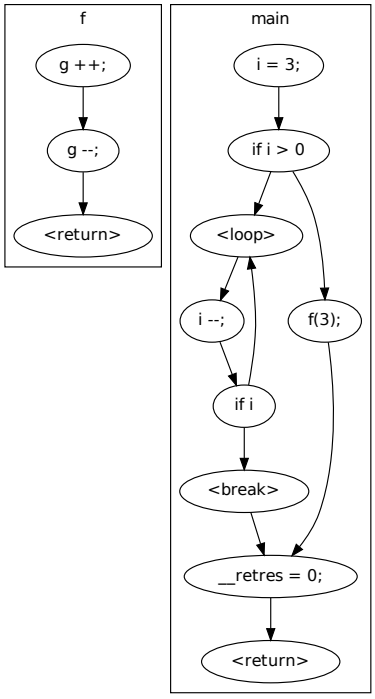


Figure 2.2: Control flow graph for file test.c.

```
    let option_name = "-cfg"
    let help =
      "when on (off by default), computes the CFG of all functions."
  end)

module OutputFile = Self.String(struct
    let option_name = "-cfg-output"
    let default = "cfg.dot"
    let arg_name = "output-file"
    let help = "file where the graph is output, in dot format."
  end)
```

```
let run () =
  if Enabled.get() then
    let filename = OutputFile.get () in
    let chan = open_out filename in
    let fmt = Format.formatter_of_out_channel chan in
    Visitor.visitFramacFileSameGlobals (new print_cfg fmt) (Ast.get ());
    close_out chan

let () = Boot.Main.extend run
```

We added two options, `-cfg` to compute the CFG conditionally (important for ordering plug-in executions), and `-cfg-output` to choose the output file.

An interesting addition would be a `-cfg-target` option, which would take a set of files or functions whose CFG would be computed, using the `Self.Kernel_function_set` functor. Depending on the targets, visiting the AST would have different starting points. This is left as an exercise for the reader.

Another interesting exercise is to solve the following problem. Currently, the complete CFG for the whole application is computed in each Frama-C step, *i.e.* executing `frama-c test.c -cfg -then -report` would compute the CFG twice. Indeed, the `-cfg` option sets `Enabled` to true, and the `run` function is executed once per task. To solve this problem, one has to create a boolean state to remember that the plug-in has already been executed. The `apply_once` function in the `State_builder` module helps dealing with this issue (reading the section 2.4.5 of this tutorial and section 4.9 of this manual should help you understand the underlying notion of states).

With these command-line options, we can properly interface our ViewCfg plug-in with the `eva` plug-in.

2.4.3 Interfacing with other plug-ins

Plug-ins can use functions specified in the public interfaces of other plug-ins, as long as they are declared as dependencies. To do so, you only need to add them to the `libraries` stanza[13] in the `dune` file. We will use a function from the Eva plug-in in our example, so we will add `frama-c-eva.core` to the `dune` file:

```
(libraries frama-c.kernel frama-c-eva.core))
```

Now our plug-in can call all functions and access all types declared in Eva's public interface.

For historical reasons, several kernel-integrated plug-ins, such as From, InOut and Slicing, had their API exposed via the `Boot` module of the Frama-C kernel. This has been deprecated for Eva, and newer plug-ins expose their public interface directly.

In our example, we will use Eva's new API to obtain reachability information computed by the value analysis. The code modification we propose is to color in pink the nodes guaranteed to be unreachable by the value analysis. For this purpose, we change the `vstmt_aux` method in the visitor:

[13] A stanza is, roughly speaking, a "term" in `dune` parlance: a parenthesized expression.

```
  method! vstmt_aux s =
    let color =
      if Eva.Analysis.is_computed () then
        if Eva.Results.is_reachable s
        then "fillcolor=\"#ccffcc\" style=filled"
        else "fillcolor=pink style=filled"
      else ""
    in
    Format.fprintf out " s%d [label=%S %s]\n"
      s.sid (Pretty_utils.to_string print_stmt s.skind) color;
    List.iter
      (fun succ → Format.fprintf out " s%d → s%d;\n" s.sid succ.sid)
      s.succs;
    Cil.DoChildren
```

This code fills the nodes with green if the node may be reachable, and in pink if the node is guaranteed not to be reachable; but only if the value analysis was previously computed.

To test this code, we recompile the plug-in with the modified `dune` file to take into account the dependency on Eva, as well as the modified `vstmt_aux`. We run `dune build @install` and then we run Eva, and then our plug-in:

```
dune exec -- frama-c test.c -eva -then -cfg && dotty cfg.dot
```

The relative order of most options and file names is not important *between* occurrences of `-then`, but it *is* important whether they are before or after `-then`-related options (`-then`, `-then-on`, `-then-last`); see Section 4.11 for details. Here, we want to ensure that Eva is run *before* our plug-in, so we order it as `-eva -then -cfg`. Without `-then`, even if `-eva` is before `-cfg` in the command line, it is *not* guaranteed that it will run before; options in the same "block" can be thought of as *concurrent*: there is no specified order between them.

The resulting graph is shown in Figure 2.3.

2.4.4 Splitting files and providing a mini-GUI for testing

Our plug-in is starting to amass enough code that we should envisage to split it into several modules, for better organizing it. Dune automatically compiles all source files it finds, unless specified otherwise, and it handles dependencies between them, so it is essentially free to do so. As we did in the *Hello* tutorial, we will split our `view_cfg.ml` file into smaller modules.

We will create the following files:

options.ml : will contain the module registration (`Self`) and command-line options (`Enabled` and `OutputFile`);

visit.ml : will contain the `print_stmt` function and the visitor;

run.ml : will contain the definition of function `run` and the call to `Boot.Main.extend`.

Note that a few changes are needed to the code: functions from other files need to include that file name as module, *e.g.* `Enabled.get` becomes `Options.Enabled.get`.

For simplicity's sake, we will remove options `-cfg` and `-cfg-output` and replace them with a single boolean option, `-cfg-gui`, to launch the GUI (this prevents unsuitable combinations of `-cfg` and `-cfg-gui`). The new option is defined as below:

```
module Gui = Self.False(struct
   let option_name = "-cfg-gui"
   let help =
     "when on (off by default), displays a mini-GUI for showing graphs."
 end)
```

File **test.c**

```
void f(int g)
{
  g++;
  g--;
}

int main(int argc, char **argv)
{
  int i = 3;

  if( i > 0)
    {
      while(--i);
    }
  else
    {
      f(3);
    }
  return 0;
}
```

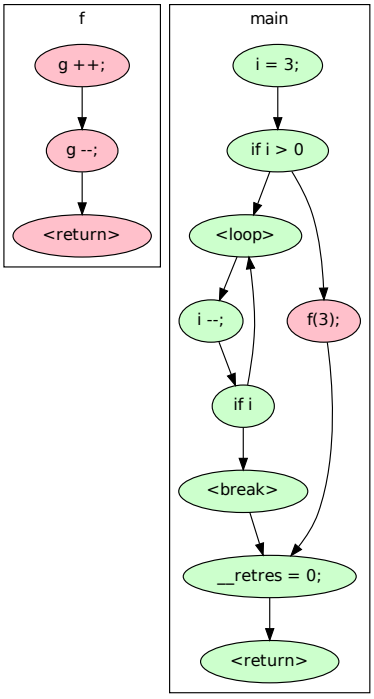


Figure 2.3: Control flow graph colored with reachability information.

And the `run` function in `run.ml` becomes simply:

```
let run () =
  if Options.Gui.get() then
    Gui.show ()
```

We can now erase `view_cfg.ml` and re-run `dune build @install` to compile the plug-in.

Mini-GUI for testing

Extending Frama-C's Ivette graphical user interface is a task too large for this tutorial; Ivette being a desktop Electron application, written in TypeScript and using React, there is a substantial amount of explaining to do before one can show how to integrate a Frama-C plug-in in it.

Instead, for this tutorial, we will use a lightweight OCaml GUI library, BOGUE[14]. You can install it through opam:

```
$ opam install bogue
```

It is based on SDL2, which means you may need to install non-OCaml dependencies[15].

Since we will be using BOGUE in our plug-in, we need to declare it in the `dune` file:

```
 (libraries frama-c.kernel frama-c-eva.core bogue))
```

We also need a way to print an individual function as a standalone graph, without having to call the file visitor (`Visit.vfile`). We will call it `dump_function` and put it in a separate file, `dump.ml`:

File **./dump.ml**

```
open Cil_types

let dump_function fundec fmt =
  Options.Self.feedback "Computing CFG for function %s"
    (fundec.svar.vorig_name);
  Format.fprintf fmt "digraph %s {\n" fundec.svar.vorig_name;
  ignore
    (Visitor.visitFramacFunction (new Visit.print_cfg fmt) fundec);
  Format.fprintf fmt "\n}\n"
```

The code prints a feedback message, then the header, calls the visitor, and prints the footer. This function will be called by our "mini-GUI", and the output will be sent to `dotty`, which will open a window with our graph.

The actual GUI code is put inside a file appropriately named `gui.ml`:

File **./gui.ml**

```
open Bogue
module W = Widget
module L = Layout

let show () =
  (* Create a few widgets for our GUI: a label, a text input with the function
     to be displayed, and a button to show it. *)
  let l = W.label "Show graph for function:" in
  let t = W.text_input ~text:"main" () in
  let b = W.button "Show CFG" in
  let status = W.label (String.make 50 '-') in (* used for error messages *)
```

[14] http://sanette.github.io/bogue/Principles.html

[15] With opam < 2.1, you may need to install and run `depext`. With opam ≥ 2.1, depext is already included.

```
let layout = L.tower_of_w [l;t;b;status] in
let show_graph _button =
  let name = W.get_text t in
  try
    (* Check the function name exists and is defined (not just declared). *)
    let kf = Globals.Functions.find_by_name name in
    let fd = Kernel_function.get_definition kf in
    W.set_text status "";

    (* Create a temporary file with the graph and pass it to 'dotty'. *)
    let (tmpname, oc) = Filename.open_temp_file "cfg_view" ".dot" in
    Dump.dump_function fd (Format.formatter_of_out_channel oc);
    close_out oc;
    let cmd = Format.asprintf "dotty %S" tmpname in
    ignore (Sys.command cmd);
    W.set_text status (String.make 50 '-');
    Unix.unlink tmpname
  with
  | Not_found →
    W.set_text status ("Error: function " ^ name ^ " not found.")
  | Kernel_function.No_Definition →
    W.set_text status ("Error: function " ^ name ^ " is not defined.")
in
W.on_release ~release:show_graph b;
let board = Bogue.make [] [layout] in
Bogue.run board
```

Most of the code is boilerplate for Bogue. The only Frama-C-related part is the checking of the function name: `Globals.Functions.find_by_name` raises exception `Not_found` if the name input by the user does not exist. But even if it does, since we need a function *definition*, we must check that it is not simply *declared*. Besides that, we simply call `dump_function`. Figure 2.4 shows what this mini-GUI looks like.

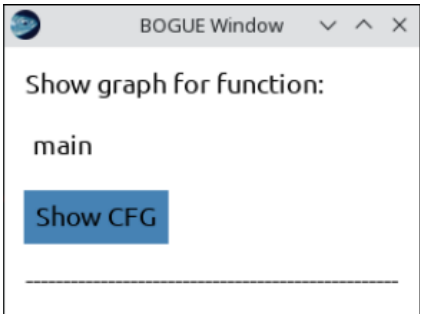


Figure 2.4: Mini-GUI with Bogue for testing our plug-in.

Whenever we click the "Show CFG" button, a new `dotty` window is opened. After we close it, we can repeat the operation as we like.

Note that the feedback message "Computing CFG for function" is emitted each time we click the button, which means we perform a new visit. For large computations and programs, this is wasteful, and we should be able to easily cache the result. The next section will show how to do it using *states*.

2.4.5 Saving/Loading Data, and Usability in a Multi-Project Setting

Registering and using state

In this section, we will learn how to register state into Frama-C. A *state* is a piece of information kept by a plug-in. For instance, we can use a boolean state to store whether an expensive analysis has been executed, to avoid recomputing it. Another example of state: the Eva plug-in computes, for each statement, a table associating to each AST variable a set of values the program may have at runtime; this association table is a state.

State registration provides several features:

- It allows the state to be saved and reloaded with the rest of the session, for instance when using `frama-c -save/frama-c -load`;
- It helps maintaining consistency between the AST and the results and parameters of the analysis of the different plug-ins.

We will modify our `Dump` module to output the DOT graph as a string, and store it in a hash table from `fundec` to `string`. Storing this string will allow us to memoize [15] our computation: the string is computed the first time the CFG of a function is displayed, while the following requests will reuse the result of the computation. Registering the hash table as a Frama-C state is *mandatory* to ensure Frama-C consistency: for instance, by using a standard OCaml hash table, a user that would have loaded several sessions through a GUI could observe the CFG of function of a previous session instead of the one he wants to observe.

Registering a state is done by a functor application:

```
module Cfg_graph_state = State_builder.Hashtbl
    (Cil_datatype.Fundec.Hashtbl)
    (Datatype.String)
    (struct
     let name = "Dump.Cfg_graph_state"
     let dependencies = [ Ast.self; Eva.Analysis.self ]
     let size = 17
```

The `State_builder` module provides several functors that help registering states. `State_builder.Hashtbl` allows the developer to create a hash table. It is parameterized by a module describing the hash table and its key, a module describing the data associated to keys, and other information.

The `Datatype` and `Cil_datatype` modules describe the hash table and its associated data, and explain for instance how the datatype should be copied, printed, or marshalled to the disk. They are part of the `Type` library [18], described in Section 4.7. `Datatype` provides descriptions for standard OCaml types, and `Cil_datatype` for the CIL types (in the `Cil_types` module).

The last module argument describes the initial size of the hash table[16], a name (mainly used for internal debugging), and a list of *dependencies*. Here we expressed that our hash table depends on the AST and the results of the Eva plug-in. For instance, whenever the Frama-C kernel updates one of these states, it will automatically reset our hash table. This ensures consistency of the analysis: if the AST of a function changes, or the value analysis is executed with a different entry point, this potentially affects the display of the control flow graph, that we must recompute.

Once the module has been declared, it is fairly easy to use it.

```
let dump_to_string fundec =
  Options.Self.feedback "Computing CFG for function %s"
    (fundec.svar.vorig_name);
  ignore
    (Visitor.visitFramacFunction
```

[16] This initial size is an optimization feature; the table automatically grows when needed.

```
      (new Visit.print_cfg Format.str_formatter) fundec);
```

```
let dump_to_string_memoized = Cfg_graph_state.memo dump_to_string

let dump_function fundec fmt =
  Format.fprintf fmt "digraph %s {\n%s\n}\n"
    fundec.svar.vorig_name
```

`dump_function` now takes two steps: first the CFG is printed to a string, then the string is printed to the `fmt` argument. This allows the `dump_to_string` part to be *memoized*, *i.e.* the results of `dump_to_string` are saved so that later calls to `dump_function` with the same `fundec` argument reuse that result.

To check this, we re-run our mini-GUI with:

```
$ dune exec -- frama-c <file> -cfg-gui
```

The first graph for each function will show the message "Computing CFG for function ...", but subsequent calls will no longer do it.

Also, we can see the effects of the dependency on the Eva plug-in by first launching the value analysis and then our plug-in:

```
$ dune exec -- frama-c <file> -eva -then -cfg-gui
```

In this case, the graphs will be colored.

Finally, to check that the state dependency on Eva works, we will use a more complex command line:

```
$ dune exec -- frama-c <file> -eva -then -cfg-gui -then -main f
```

We have three stages: first run Eva, then open the mini-GUI, then change the entry point (`-main f` means that the program will start executing from function `f`) *and* re-open the mini-GUI. Remember that most Frama-C options *persist* from one stage to the next: the same way that we do not have to repeat `-eva` after the first `-then`, we do not have to repeat `-cfg-gui` after the second `-then`. If we want to avoid re-running the mini-GUI in future stages, we need to use `apply_once`, as mentioned in Section 2.4.2.

What we observe is the following: when the mini-GUI opens, we click *Show CFG*, see a "Computing CFG for function ..." message, and get a mostly-green CFG. Then, we close the mini-GUI, and it opens again. Clicking *Show CFG* will show the same "Computing CFG" message, but the graph will be entirely pink: with `f` as the entry point, function `main` is never called, therefore entirely unreachable. Because the entry point changed, and Eva depends on its state, it is automatically recomputed. Because our plug-in depends on Eva's state, it recomputes the graph when we ask again for the CFG.

Another way to observe how Frama-C automatically handles states is to display a CFG, save the session (option `-save <file>`), and then load it again:

```
$ dune exec -- frama-c <file> -eva -then -cfg-gui -save session.sav
```

Then click on the "Show CFG" (see the feedback message), then close the CFG and the mini-GUI. Reload the session:

```
$ dune exec -- frama-c -load session.sav
```

Click "Show CFG" and you will *not* see the "Computing CFG" message: the state `Cfg_graph_state` had beem automatically saved by Frama-C and has just been loaded from the session.

Clearing states, selection and projects

There is one caveat though: if the user computes the CFG before running the Eva analysis, and then runs Eva, they will not see a colored graph (unless they re-launch Eva with different parameters). This is because the state of the CFG is reset when the state of Eva is reset, not when it is first computed.

To solve this problem, we will manually reset the `Cfg_graph_state` if we detect that Eva has been run since the last time we computed the CFG. For that, we have to remember the previous value of `Eva.Analysis.is_computed ()`, *i.e.* to register another state:

```
module Eva_is_computed = State_builder.Ref
    (Datatype.Bool)
    (struct
     let name = "Dump.Eva_is_computed"
     let dependencies = []
     let default () = false
    end)
```

This new state only consists of a reference to a boolean value.
Then we just replace `dump_function` in the code above by the following.

```
let dump_function fundec fmt =
  if not (Eva_is_computed.get ()) && Eva.Analysis.is_computed () then begin
    Eva_is_computed.set true;
    let selection = State_selection.with_dependencies Cfg_graph_state.self in
    Project.clear ~selection ();
  end;
  Format.fprintf fmt "digraph %s {\n%s\n}\n"
    fundec.svar.vorig_name
    (dump_to_string_memoized fundec)
```

The only parts that need to be explained are the notions of *selection* and *project*. A selection is just a set of states; here we selected the state `Cfg_graph_state` with all of its dependencies, as resetting this state would also impact states that would depend on it (even if there are none for now). We use `Project.clear` to reset the selection.

Project explanation

A *project* [17] is a consistent version of all the *states* of Frama-C. Frama-C is multi-AST, *i.e.*Frama-C plug-ins can change the AST of the program, or perform incompatible analyses (*e.g.* with different entry points). Projects consistently group a version of the program's AST with the states related to it.

The `Project.clear` function has type:

```
val clear: ?selection:State_selection.t → ?project:t → unit → unit
```

The arguments `selection` and `project` can be seen as a coordinate system, and the function allows to clear specific versions of specific states. By default, Frama-C functions act on the *current* project. The developer has to use `Project.on` or optional arguments to act on different projects. Frama-C automatically handles duplication and switch of states when duplicating or changing of projects. This is the last benefit of state registration.

To summarize:

- To store results, plug-ins should register *states*;
- A *project* is a consistent version of all the states in Frama-C, together with a version of the AST;
- A *session* is a set of *projects*;
- Frama-C transparently handles the versioning of states when changing or duplicating projects, saving and reloading sessions from disk, etc.
- The version of the state in a project can change; by default Frama-C functions operate on the current project.
- A *selection* is a set of states. *Dependencies* allow to create selections.
- As a plug-in developer, you have to remember that is up to you to preserve consistency between your states and their dependencies by clearing the latter when the former is modified in an incompatible way. For instance, it would have been incorrect not to call `State_selection.with_dependencies` in the last code snippet of this tutorial.

Projects are generally created using copy visitors. We encourage the reader to experiment with multi-project development by using them. An interesting exercise would be to change the AST so that execution of each instruction is logged to a file, and then re-read that file to print in the CFG how many times each instruction has been executed. Another interesting exercise would be to use the `apply_once` function so that the ViewCfg plug-in is executed only once, as explained in section 2.4.2 of this tutorial.

Software Architecture 3

Target readers: *beginners.*

In this chapter, we present the software architecture of Frama-C. First, Section 3.1 presents its general overview. Then, we focus on four different parts:

- Section 3.2 explains what a plug-in really is and the main mechanisms of plug-in integration.
- Section 3.3 introduces the libraries that Frama-C provides.
- Section 3.4 introduces the kernel services that plug-in developers might want to use.
- Section 3.5 introduces the kernel internals. You can safely skip it if you are not a Frama-C kernel developer.

3.1 General Description

From a plug-in developer point of view, the main goals of the Frama-C platform is to provide services to ease:

- analysis and source-to-source transformation of big-size C programs;
- addition of new plug-ins; and
- plug-ins collaboration.

In order to reach these goals, Frama-C has a plug-ins based software architecture based on a *kernel*. Historically the *kernel* was itself based on Cil [16]: even if they have evolved separately since the Frama-C Hydrogen age, there are still a lot of similarities between Cil and several modules of the Frama-C kernel (*e.g.* the ASTs).

The Frama-C architecture design is presented in this chapter, and summarized in Figure 3.1. In this figure, each of the four rounded colored boxes represents a subdirectory d of directory `src`, while each of the small square boxes represents a subdirectory in one subdirectory `src/`d. The remaining sections will explain the goal of each of these boxes. They do not detail each module of each directory: use the API documentation generated by `make doc` for that purpose.

3.2 Plug-ins

In Frama-C, plug-ins are program analysis or source-to-source transformations. The ones provided within Frama-C are in directory `src/plugins/`*plugin_name*. Each plug-in is an extension point of Frama-C which has to be registered through `Plugin.Register` (see Section 4.5). Frama-C allows plug-in collaborations: a plug-in p can use a list of plug-ins $p_1, \ldots, p_n$ and conversely. Mutual dependences between plug-ins are even possible. If a plug-in is designed to be used by another plug-in, it has to register its API either by providing a `.mli` file or through module `Dynamic`. The first method is the

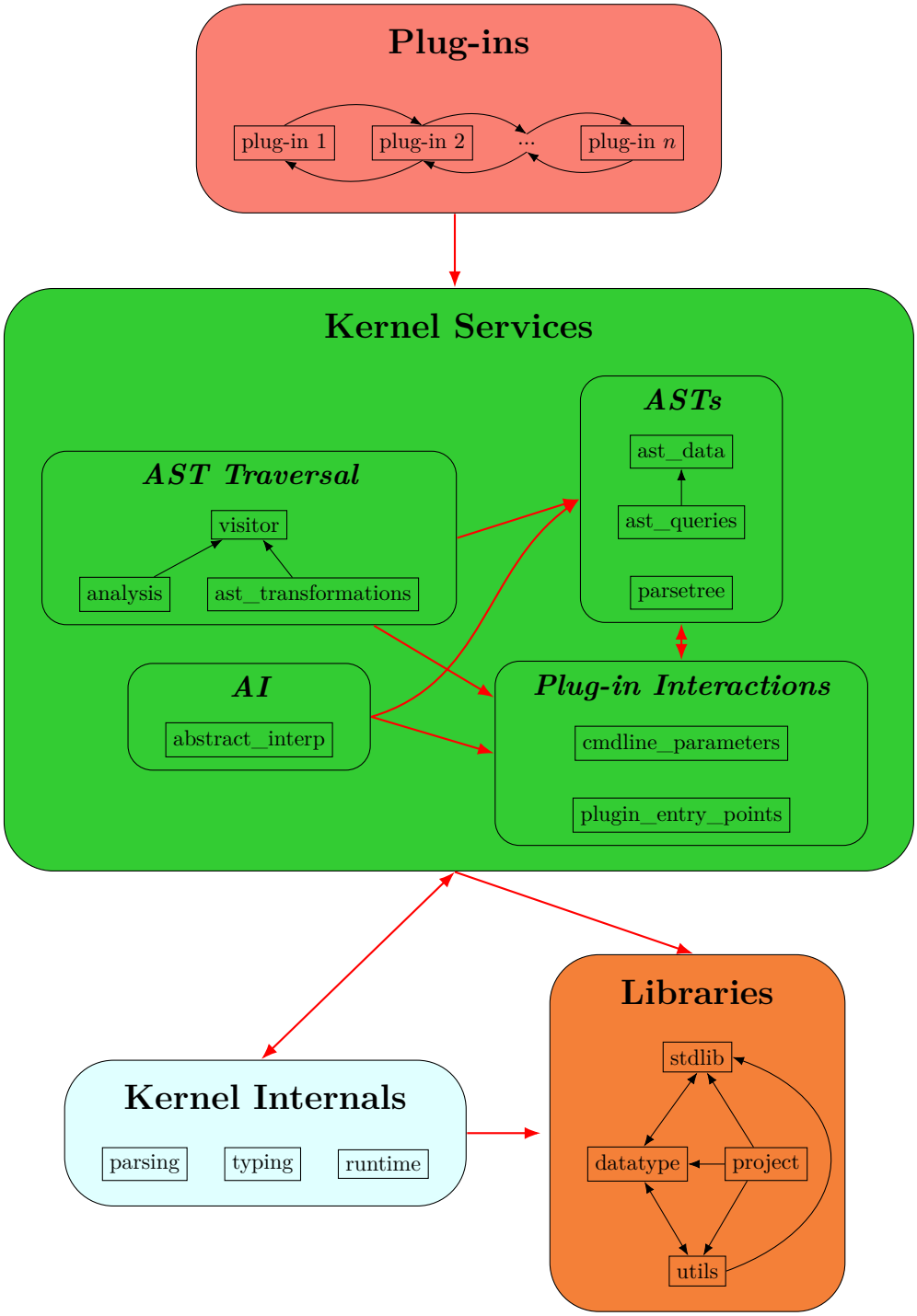


Figure 3.1: Frama-C Architecture Design.

preferred one, the second one (through module `Dynamic`) is the only possible one to define mutually dependent plug-ins.

3.3 Libraries

Libraries are defined in the directory `src/libraries`. They are either third-party libraries or dedicated independent libraries which may be used by other parts of Frama-C.

Extension of the OCaml standard library are provided in directory `src/libraries/stdlib`. For instance, modules `FC`*module_name* (e.g. `FCHashtbl`) provides a *module_name*-like interface (e.g. `Hashtbl`) with an uniform API for all OCaml versions supported by Frama-C and possibly additional useful operations for the corresponding datastructures.

Single-file libraries are provided in directory `src/libraries/utils`. For instance, `Pretty_utils` provides pretty-printing facilities.

datatype (directory `src/libraries/datatype`) and *project* (directory `src/libraries/project`) are two multiple-files libraries. They are respectively presented in Sections 4.7 and 4.9.

3.4 Kernel Services

The kernel services is defined in directory `src/kernel_services`. It is the part of the Frama-C kernel which may be useful to develop plug-ins.

This services may be split in four main parts:

- two ASTs and the data structures directly built upon them;
- plug-in interactions toolbox;
- predefined generic analyzers;
- the abstract interpretation framework.

The standard AST used by most analyzers is defined in module `Cil_types` of directory `src/kernel_services/ast_data`. It contains the types which describes both the C constructs and the ACSL ones [1]. The same directory also contains modules defining datastructures directly based upon the AST (module `Globals`), functions (module `Kernel_function`), annotations (module `Annotations`) and so on. A related directory is `src/kernel_services/ast_queries`. It contains modules which provides specific operations in order to get information about AST-related values.

In the same way, an untyped AST quite close of the C input source is defined in module `Cabs` of directory `src/kernel_services/parsetree`. This AST is only well suited for syntactic analyzers/program transformers.

Frama-C also provides services to plug-ins which help both their integration in the platform and their interactions with the kernel and the other plug-ins. They enforce some platform-wide consistency by ensuring that some common actions (*e.g.* printing messages to end-users) are handled by all plugins in a similar way. Directory `src/kernel_services/cmdline_parameters` provides modules which eases the definition of analyzers' parameters accessible by the end-user through command-line options. Similarly, directory `src/kernel_services/plugin_entry_points` provides modules to interact with the Frama-C kernel or other plug-ins.

Next, Frama-C provides predefined ways to visit the ASTs, in particular through object-oriented visitors defined in directory `src/kernel_services/visitors` (see Section 4.14). Some predefined analyzers, such as a multiple generic dataflow analysis are provided in directory `src/kernel_services/analysis`, while some predefined program transformation, such as cloning a function, are provided in directory `src/kernel_services/ast_transformations`.

Finally, Frama-C provides an abstract interpretation toolbox with various lattices in directory `abstract_interp`.

3.5 Kernel Internals

Target readers: *kernel developers.*

The kernel internals is defined in directory `src/kernel_internals`. It is part of the Frama-C kernel which should be useless to develop analysis plug-ins, except possibly for very low-level interactions, or to extend the C or ACSL services of Frama-C. Consequently, if you are not a kernel developer, you can safely ignore this section.

The internals is split in three different parts, each of them being an independent subdirectory:

- the directory `src/kernel_internals/parsing` contains the lexer and parser which generate the untyped AST (*aka* Cabs) from the C input source code;
- the directory `src/kernel_internals/typing` contains the compiler which generates the standard AST (*aka* Cil) from the untyped one;
- the directory `src/kernel_internals/runtime` contains modules whose primary purpose is to perform side-effects while Frama-C is booting.

Advanced Plug-in Development 4

This chapter details how to use services provided by Frama-C in order to be fully operational with the development of plug-ins. Each section describes technical points a developer should be aware of. Otherwise, one could find oneself in one or more of the following situations [1] (from bad to worse):

1. reinventing the (Frama-C) wheel;
2. being unable to do some specific things (*e.g.* saving results of an analysis on disk, see Section 4.9.2);
3. introducing bugs in his/her code;
4. introducing bugs in other plug-ins using his/her code;
5. breaking the kernel consistency and so potentially breaking all Frama-C plug-ins (*e.g.* if s/he modifies the AST without changing project, see Section 4.9.5).

In this chapter, we suppose that the reader is able to write a minimal plug-in like `hello` described in chapter 2 and knows about the software architecture of Frama-C (chapter 3). Moreover plug-in development requires the use of advanced features of OCaml (module system, classes and objects, *etc*). Plug-in development also requires some familiarity with the Dune build system.

Note that the following subsections can be read in no particular order: their contents are indeed quite independent from one another even if there are references from one section to another one.

4.1 Plug-in dependencies

Target readers: *standard plug-in developers.*

Prerequisite: *Basic knowledge of the Dune build system.*

Some plug-ins only depend on the Frama-C kernel, and in this case, require no special configuration. However, many plug-ins depend either on other Frama-C plug-ins, or on external libraries, tools and components. In both cases, it is important to specify such dependencies, to avoid the user trying and failing to compile or run your plug-in.

Plug-in dependencies are usually defined in the plug-in's `dune` file. There are mainly two kinds of dependencies: *mandatory* and *optional*. Mandatory dependencies are necessary for the plug-in to run *at all*, and their absence means the plug-in must be disabled. Optional dependencies do not prevent compilation and usage of the plug-in, but it may work less efficiently, precisely, or fail when specific features are requested.

Plug-in developers must ensure that optional dependencies are tested for their absence and dealt with gracefully; a plug-in must not *crash when optional dependencies are missing.*

[1] It is fortunately quite difficult (but not impossible) to fall into the worst situation by mistake if you are not a kernel developer.

Most dependencies (OCaml library dependencies, such as other plug-ins or OCaml modules) are specified using Dune's `library` stanza[2]. As is often the case, examples are very instructive; here is an example of mandatory dependencies from the Dive plug-in's `dune` file:

```
(library
  [...]
  (libraries frama-c.kernel frama-c-studia.core frama-c-server.core)
)
```

The `libraries` field contains `frama-c.kernel` (essential for all plug-ins), but also `frama-c-studia.core` and `frama-c-server.core`, meaning this plug-in requires both the Studia and Server plug-ins to be enabled. If any of them is disabled (either due to missing dependencies, or as a result of a user request), then Dive will also be disabled.

Note: to explicitly disable a plug-in, use the `dev/disable-plugins.sh` *script.*

Note that OCaml libraries can also be specified, e.g. adding `zarith` to `libraries` above would require that Zarith be installed and available via `ocamlfind`.

4.1.1 Declaring dependencies

For optional dependencies, you can define two versions of the implementation (both implementing the same interface), in two different files. One of these files will use the required module, while the other will implement a dummy version without it. The `select` notation from the `library` stanza allows telling Dune which file to compile.

Here is an example of `select` from the Aoraï plug-in, which defines an optional dependency on the Eva plug-in:

```
(libraries [...]
 (select aorai_eva_analysis.ml from
  (frama-c-eva.core -> aorai_eva_analysis.enabled.ml)
  (  -> aorai_eva_analysis.disabled.ml)
 ))
```

In the example above, Aoraï defines two files: `aorai_eva_analysis.enabled.ml` and `aorai_eva_analysis.disabled.ml` (we recommend this naming convention, but it is not enforced by `dune`). The general form of the `select` stanza is:

```
(select < file>  from
  (< cond_1>  → < file_1> )
  (< cond_2>  → < file_2> )
  ...
  ( → < default> )
)
```

It then checks for each condition `<cond_i>` in order, and the first one that matches is selected, otherwise it will be the fallback/default one. The selected `<file_i>` will be renamed to `<file>` and used in the build.

4.1.2 Notifying users via frama-c-configure

Frama-C has a special Dune target, `@frama-c-configure`, which prints a summary of the configuration as seen by Dune when run via `dune build @frama-c-configure`. It is especially helpful to understand why a given plug-in is disabled.

---

[2] Full documentation about Dune's |library| stanza is available at https://dune.readthedocs.io/en/stable/concepts.html#library-deps

Plug-in developers should *always* include such a section in their `dune` file, listing each optional library and its current availability, via the `%{lib-available:<lib>}` special variable.

Here is an example from Frama-Clang, which requires the following libraries:

```
(libraries frama-c.kernel frama-c-wp.core camlp-streams zarith)
```

Its plug-in library name is `frama-clang.core`. Its availability must be displayed as the "summary" of the configuration, written as the first line, with each dependency as a subitem:

```
(rule
 (alias frama-c-configure)
 (deps (universe))
 (action (progn
          (echo "Clang:" %{lib-available:frama-clang.core} "\n")
          (echo "  - zarith:" %{lib-available:zarith} "\n")
          (echo "  - camlp5:" %{lib-available:camlp5} "\n") ; for gen_ast
          (echo "  - camlp-streams:" %{lib-available:camlp-streams} "\n")
          (echo "  - wp:" %{lib-available:frama-c-wp.core} "\n")
  )
  )
)
```

Note that `camlp5` is not among the dependencies declared in `libraries`; it is used by another component, `gen_ast`.

4.2 Frama-C Makefiles

Target readers: *standard plug-in developers.*

Prerequisite: *Knowledge of* make.

Since the adoption of Dune, Frama-C's `Makefiles` were largely simplified. They are still used for some tasks, partially due to conciseness (`make` is shorter than `dune build`), partially because some tasks are better suited for them. Some standard targets are defined in various Makefiles that are installed in Frama-C's shared directory.

A minimal plugin's `Makefile` can thus be the following:

File **Makefile**

```
FRAMAC_SHARE:=$(shell frama-c-config -print-share-path)

## Common definitions

include ${FRAMAC_SHARE}/Makefile.common

## Tests-related targets

include ${FRAMAC_SHARE}/Makefile.testing

## Installation-related targets

include ${FRAMAC_SHARE}/Makefile.installation
```

`Makefile.testing` introduces various targets related to Frama-C's testing infrastructure (see Section 4.3). This includes notably `tests` to run all the tests of the plugin, after having taken care of generating the corresponding `dune` files.

`Makefile.installation` provides two targets, `install` and `uninstall`. By default, installation will occur in the current `opam` switch, but this can be modified by using the `PREFIX` variable (note that if you install your plugin in a non-default place, you will have to explicitly instruct Frama-C to load it through option `-load-plugin`.

Other Makefiles include `Makefile.documentation`, providing the `doc` for generating the documentation (see Section 2.3.8), and `Makefile.linting` which is used by Frama-C itself to perform various syntactic checks through target `check-lint` and fix indentation issues through `lint`. If you want to use the same conventions as Frama-C itself, you'll need to have `ocp-indent` installed to launch these targets.

4.3 Testing

In this section, we present ptests, a tool provided by Frama-C in order to perform non-regression and unit tests.

Historically, ptests *has been developed before* Frama-C *switched to Dune. It has been ported to Dune, but some features had to be adapted and others became redundant; it is likely that in the future* ptests *will be replaced with direct usage of the Dune testing framework.*

ptests is a test preparation tool that parses specially crafted C comments to create several Dune test targets, so that users can more easily create test cases with custom Frama-C commands.

The generated test targets are then run via Dune. Each result of the execution is compared with the previously saved result (called the *oracle*). A test is successful if and only if there is no difference. Actually, the number of results is twice the number of tests, because standard and error outputs are compared separately.

First, Section 4.3.1 explains how to use ptests. Next, Section 4.3.2 describes the test file structure for Frama-C plug-ins. Section 4.3.2 introduces the syntax of test headers, that is, how to define test cases and their options. Section 4.3.3 explains how to declare test dependencies (files other than the test itself). Last, Section 4.3.4 presents the full list of ptests directives.

4.3.1 Using ptests

ptests only *prepares* tests (by creating appropriate dune files), but does not run them. Whenever a new test file is added, or test headers of existing test files are modified, you need to run `frama-c-ptests` to regenerate them. Note that, to run all of Frama-C tests (which include several directories, plus the tests of many plug-ins), you should run `make run-ptests` instead, which will call `frama-c-ptests` with the proper arguments to generate test targets for *all* directories.

If new tests are created (by adding a new file, or adding an extra test case to an existing file), you must run `frama-c-ptests -create-missing-oracles`. This will create empty test oracles for Dune.

Then, to run the tests, use `dune build @ptests`. If there are differences between the current run and oracles from previous runs, Dune will indicate them and mark the tests as failures. If the changes are intended, after manual inspection you can *promote* them as new oracles, via `dune promote`. This will simply replace the existing oracles with the new ones. Finally, you can use `frama-c-ptests -remove-empty-oracles` to remove empty oracles (typically, messages sent to the standard error) before committing the files to version control.

If you simply want to re-run all tests for your plug-in (and you have included the shared testing `Makefile`) run `make tests`.

4.3.2 Test directory structure

This is the file structure of the `tests` directory, where Frama-C plug-ins are expected to place their tests:

```
< plug-in directory>
+- tests
  +- ptests_config
  +- test_config
  +- suite1
    +- test1.c
    +- ...
    +- oracle
      +- test1.res.oracle
      +- test1.err.oracle
      +- ...
    +- result
      +- test1.res.log
      +- test1.err.log
      +- ...
 +- ...
```

Files `test_config` and `ptests_config`

Inside `tests`, two files are mandatory: `ptests_config` and `test_config`.

`ptests_config` usually contains a single line with the list of *test suites* for that plug-in, prefixed with `DEFAULT_SUITES=`. A test suite is a sub-directory of the current `tests` directory.

```
DEFAULT_SUITES=basic parsing misc options
```

In the example above, our plug-in has at least four sub-directories inside the `tests` directory. Each test suite can contain any number of test files.

`test_config` contains a list of common *directives* to be applied to each test, in all test suites. It typically contains a line `PLUGIN: <module>` with the name of the plug-in being developed (for more details about `PLUGIN:`, check Section 4.3.4).

Each test suite can also contain its own `test_config` file. Each test case in that suite will then inherit the directives from the parent `test_config` as well as those from the `test_config` in its directory.

Finally, different *test configurations* can be specified by creating other files, e.g. the existence of a file `test_config_prove` will create a configuration named `prove`. A test case can contain multiple *test headers* (to be detailed in the next section), with different directives for each configuration. Configurations are specified when running tests, e.g. instead of running `dune build @ptests`, you can run instead `dune build @ptests_config_prove` to run tests only from the `prove` configuration.

All Dune `@ptests`*-related targets must be run* after `frama-c-ptests` *(or* `make run-ptests`*) has been run, otherwise Dune may not find the generated files and report errors such as* Alias "ptests_config" specified on the command line is empty.

The catch-all command `make tests` run tests for *all* suites, and *all* configurations.

Test headers

Inside each test suite, we find one or more test files: either `.c` or `.i` files which typically start with the following header:

```
/* run.config
   // test directives, one per line
   ...
*/
```

That is, a multiline C comment block (`/* */`) with `run.config` and a list of lines containing test directives. This test header tells ptests how to generate the test commands.

Note that a configuration suffix can be added to `run.config`. Also, we can define multiple test headers, such as:

```
/* run.config
   STDOPT:
*/
/* run.config_eva
   STDOPT: +"-eva"
*/
```

This will create a test case for the default configuration, and a different one for the `eva` configuration.

Multiple configurations may share the same set of directives:

```
/* run.config, run.config_<name> [, ...]
   ... common directives ...
*/
```

The following wildcard is also supported, and matches any configuration: `/* run.config* .` Note that it does not define a new configuration, but generates tests for every existing configuration in the directory (as defined by the existence of `test_config_*` files).

`oracle` and `result` directories

Inside each test suite, there are two directories, `oracle` and `result`[3], each containing at least one file per test case (remember that a single test file can contain several test cases, one per directive).

The `oracle` directory contains the test oracles, that is, the expected results saved from previous runs.

The `result` directory contains the generated Dune files produced by ptests.

The naming of these directories is due to historical reasons; the `result` *directory used to contain the current test outputs, which were then compared to the oracles, mirroring their directory structure. Nowadays, it could be called* `targets`*, for instance.*

Oracle files are named `<test_name><optional_test_number>.<stream>.oracle`. `<test_name>` is simply the filename of the test file, minus its extension[4]. `<optional_test_number>` is a 0-numbered list of test numbers for files with multiple tests: `.0`, then `.1`, `.2`, etc. It is omitted if the test file defines a single test case. `<stream>` is `out` for `stdout` (standard output) and `err` for `stderr` (standard error). Therefore, there are two oracle files for each test case. An absent oracle is equivalent to an empty one.

Files in `result` are named `<test_name>.<test_number>.exec.wtests` (or, in case there are `EXECNOW`[5] directives, they end with `.execnow.wtests`). They are always 0-numbered, even when there is a single test case per file. These are JSON files containing the data to be used (by

[3] If there are other configurations, there will also be an `oracle_<config>` and a `result_<config>` directory per configuration.

[4] Note that this prevents having two tests in the same suite which differ only by their extension, e.g. `test1.c` and `test1.i`.

[5] See Section 4.3.4 for details about this and other directives.

`frama-c-wrapper` via the generated `dune` file) in each test, e.g. the command-line to be run, the output file name, and the associated oracle. These `.wtests` files are not generated when the use of `frama-c-wrapper` is not required (via ptests' option `-wrapper ""`).

4.3.3 What happens when you run a test

Complex tests using relative paths or extra files require understanding how Dune runs the tests, so that they can be properly written.

By default, Dune will create a `_build` directory where it stores compiled files and other resources the user may request. When a test is run, Dune will copy the test file and its dependencies (detailed in the next section) to a subdirectory inside `_build` (usually, mirroring the original directory structure), `cd` to the test subdirectory, and run the test command from there. It will redirect the standard and error streams to two files (`<test>.res.log` and `<test>.err.log`), and compare them to the test oracles. If they are identical, by default Dune will not report anything. Otherwise, it will print a message and the diff between the current result and the previous oracle.

Test dependencies

Dune tracks dependencies between files so that it knows when objects must be rebuilt and tests must be run again. For compiling OCaml files, it usually computes this information automatically. For tests, however, such information must come from the user. ptests will use it to generate Dune rules that will ensure tests are re-run if their dependencies are modified. Properly annotating test dependencies is essential to ensure reproducible, parallelizable tests. Otherwise, non-deterministic behavior may occur.

One way to declare dependencies is to use a `DEPS` directive[6] with a space-separated list of files. For instance, a test `file.c` which #includes a `file.h` must declare it as a dependency:

```
/* run.config
   DEPS: file.h
*/
#include "file.h"
...
```

When tests mention their dependencies in the command line, a shorter syntax is available, using the Dune special variable `%{dep:<file>}`. For instance, tests from `saveload` typically consist of multiple sub-tests, one creating a session file with `-save` and another loading it with `-load`. The second test obviously must run *after* the first one; adding the dependency will ensure that Dune will sequence them correctly. In the example below, each `STDOPT` directive defines a test case.

```
/* run.config
   STDOPT: +"-save test1.sav"
   STDOPT: +"-load %{dep:test1.sav}"
*/
...
```

You can combine `DEPS` and `%{dep:}` as you wish in your tests. Prefer `%{dep:}` for local dependencies, since it does not accumulate towards following tests, and `DEPS` for dependencies which are common to several tests.

Note that forgetting to specify dependencies can lead to test failures in unexpected ways, such as the dependencies not being copied to Dune's test directory, or two tests running in parallel and reading/writing concurrently to the same file.

---

[6] Section 4.3.4 provides details about ptests directives.

Working directory and relative paths

Any directive can identify a file using a relative path. The current working directory (considered as `.`) will be a directory inside Dune's sandbox, that is, a `result` directory inside the "mirror" directory of the test file. This "mirror" directory structure is created by Dune (by default, inside a `_build` directory in the root of the Dune project), with dependencies copied lazily for each test. Therefore, pay attention to the fact that references to the parent directory (`..`) will likely not match what you expect.

4.3.4 Detailed directives

Directives can have various functions: some create test cases, while others modify the environment (and therefore affect other directives). Each directive is specified in a separate line and has the form

```
DIRECTIVE: value (everything until the newline character)
```

Example 4.1 *Test `tests/sparecode/calls.c` declares the following directives.*

```
/* run.config
   OPT: -sparecode-analysis
   OPT: -slicing-level 2 -slice-return main -slice-print
*/
```

These directives state that we want to test sparecode and slicing analyses on this file. Two test cases are defined, each with its specific set of options. The default command (`frama-c`) will be run with these arguments, plus the test file name, plus a few other implicit options (`-check`, `-no-autoload-plugins`, `-load-module`, etc).

Table 4.1 shows all the directives that can be used in the configuration header of a test (or a test suite). Those whose name are underlined are the directives that *actually* create test cases; the others modify or disable test cases.

Name	Specification	default
`CMD`	Program to run.	`frama-c`
`COMMENT`	A configuration comment; this directive does nothing.	*None*
`DEPS`	The list of files this test depends on.	*None*
`DONTRUN`	Disable this test file for the current configuration.	*None*
`ENABLED_IF`	Conditionally enable subsequent tests, using Dune variables (e.g. `%{bin-available:gcc}`).	`true`
`EXECNOW`	Run a custom command.	*None*
`EXIT`	Indicate expected exit code, if not 0.	*None*
`FILEREG`	File name pattern of files considered as test cases.	`.*\.\(c|i\)`
`FILTER`	Command reading the standard input used to filter results. In such a command, the predefined macro `@PTEST_ORACLE@` is set to the basename of the oracle.	*None*
`LIBS`	Libraries to be loaded with each subsequent run (their compilation is not managed by ptests, contrary to the modules of `MODULE` directive).	*None*
`LOG`	Add an additional file that the test must generate and compare against an oracle. Note that this directive is only used by 'OPT' and 'STDOPT' directives. The syntax for `EXECNOW` related to that need is different (see the description of `EXECNOW` directive).	*None*
`MACRO`	Define a new macro.	*None*

MODULE	Register a dynamic module to be built and loaded with each subsequent test.	*None*
NOFRAMAC	Empty the list of frama-c commands to be launched (EXECNOW directives are still executed). Used when the test must not run |frama-c|, but another command.	*None*
OPT	Options given to the program.	*None* (but usually defined in test_config)
PLUGIN	Plugins to be loaded with each subsequent run.	*None* (but usually defined in test_config)
STDOPT	Add and remove options from the default set (see text for syntax details).	*None*
TIMEOUT	Kill the test after the given duration (in seconds of CPU user time) and report a failure.	*None*

Table 4.1: Directives in configuration headers of test files. Underlined directives are the only ones which actually generate test cases.

In the following, we detail some aspects of several directives.

- DONTRUN and NOFRAMAC directives do not need any content, but it might be useful to provide an explanation of why the test should not be run (*e.g* test of a feature that is under development and not fully operational yet).
- CMD allows changing the command that is used for the following OPT directives (until a new CMD directive is found). No new test case is generated if there is no further OPT directive. For a given configuration level, the default value for directive CMD is the last CMD directive of the preceding configuration level.
- LOG adds a file to be compared against an oracle in the next OPT or STDOPT directive. Several files can be monitored from a single OPT/STDOPT directive, through several LOG directives. These files must be generated in the result directory of the corresponding suite (and potentially alternative configuration). After an OPT or STDOPT directive is encountered, the set of additional LOG files is reset to its default. Note that EXECNOW directives can also be prefixed with LOGs, but they are written in the same line, without the separating colon (:).
- By default, the test command (usually, frama-c) is expected to return successfully (i.e., with an exit status of 0). If a test is supposed to lead to an error, an EXIT directive must be used. It takes as argument an integer (typically 1 to denote a user error) representing the expected exit status for the subsequent tests. All tests triggered by OPT or STDOPT directives encountered after the EXIT directive will be expected to exit with the corresponding status, until a new EXIT directive is encountered. (EXIT: 0 will thus indicate that subsequent tests are expected to exit normally).
- If there are several OPT directives in the same configuration level, they correspond to different test cases. The OPT directives of a given configuration level replace the ones of the preceding level.
- The STDOPT directive takes as default set of options the last OPT directive(s) of the preceding configuration level. If the preceding configuration level contains several OPT directives, hence several test cases, STDOPT is applied to each of them, leading to the same number of test cases. The syntax for this directive is the following.

```
STDOPT: [[+#-]"opt" ...]
```

unlike in OPT, here **options are always given between quotes**. An option following a + (resp. #) is added to the end (resp. start) of the current set of options, while an option following a – is removed from it. The directive can be empty (meaning that the corresponding test will use the standard set of options). As with OPT, each STDOPT corresponds to a different (set of) test

case(s). LOG directives preceding an STDOPT are taken into account.

– The syntax for EXECNOW directives is the following.

```
EXECNOW: [ [ LOG file | BIN file ] ... ] cmd
```

Files after LOG are log files generated by command cmd and compared from oracles, whereas files after BIN are binary files, also generated by cmd, but not compared to any oracles. Full access path to these files has to be specified only in cmd. Execution order between different OPT/STDOPT/EXECNOW directives is unspecified, unless there are dependencies between them (see DEPS directive). EXECNOW directives from a given level are added to the directives of the following levels.
Note: An EXECNOW command without BIN and without LOG will not be be executed by Dune; a warning is emitted in this case.

– The MACRO directive has the following syntax:

```
MACRO: macro-name content
```

where macro-name is any sequence of characters containing neither a blank nor an @, and content extends until the end of the line. Once such a directive has been encountered, each occurrence of @macro-name@ in a CMD, LOG, OPT, STDOPT or EXECNOW directive at this configuration level or in any level below it will be replaced by content. Existing predefined macros are listed in section 4.3.5.

– The MODULE directive takes as argument the name of a .cmxs module. It will then add a directive to compile this file with the command @PTEST_MAKE_MODULE@ <MODULE> where @PTEST_MAKE_MODULE@ defaults to make -s. Option -load-module <MODULE> will then be appended to any subsequent Frama-C command triggered by the test.

– The LIBS directive takes as argument the name of a .cmxs module. The -load-module <LIBS> will then be appended to any subsequent Frama-C command triggered by the test. The compilation is not managed by ptests.

– The FILEREG directive contains a regular expression indicating which files in the directory containing the current test suite are actually part of the suite. This directive is only usable in a test_config configuration file.

– The FILTER directive specifies a transformation on the test result files before the comparison to the oracles. The filtering command read the result from the standard input and the oracle will be compared with the standard output of that command. In such a directive, the predefined macro @PTEST_ORACLE@ is set to the basename of the oracle. That allows running a diff command with the oracle of another test configuration config:

```
FILTER: diff --new-file @PTEST_DIR@/oracle_config/@PTEST_ORACLE@ -
```

Chaining multiple filter commands is possible by defining several FILTER directives (they are applied in the reverse order), and an empty command drops the previous FILTER directives.

– The DEPS directive takes a set of filepaths and adds them to the set of dependencies for the next OPT/STDOPT/EXECNOW directives. Whenever these dependencies change, the test cases depending on them must be re-run. Otherwise, Dune does not re-run successful tests. Dependencies also ensure that tests which require output from others are run serially and not in parallel. Note that Dune also has a special variable notation which can be used to specify dependencies: %{dep:<file>}. For instance, the following block:

```
DEPS: file1.h file2.c
OPT: -cpp-extra-args="-Ifile1.h" file2.c
```

Is equivalent to:

```
OPT: -cpp-extra-args="-I%{dep:file1.h}" %{dep:file2.c}
```

The special variable notation is interpreted by Dune before executing the command. All dependencies (either via `DEPS` or `%{dep:}`) are collected and added to the set of dependencies for the test case.

– If there are `OPT`/`STDOPT` directives *after* a `NOFRAMAC` directive, they will be executed, unless they are themselves discarded by another subsequent `NOFRAMAC` directive.

`@` in the text of a directive *As mentioned above, `@` is recognized by* ptests *as the beginning of a macro. If you need to have a literal `@` in the text of the directive itself, it needs to be doubled, i.e. `@@` will be translated as `@`.*

Summary: ordering of test executions
There is no total ordering between the tests in a test file header. The only ordering is dictated by the dependencies declared in the test cases. Dune will by default run tests in parallel.
A consequence of this ordering is that, if you need a test to produce output that will be consumed by another test, the consumer must *declare the produced file as a dependency.*

4.3.5 Pre-defined macros for tests commands

Table 4.2 gives the definition of the most important predefined macros that can be used in ptests' directives. Refer to `frama-c-ptests -help` to have the full list.

Name	Expansion
`frama-c`	path to Frama-C executable
`PTEST_CONFIG`	either the empty string or _ followed by the name of the current alternative configuration (see section 4.3.2).
`PTEST_DIR`	current test suite directory
`PTEST_FILE`	path to the current test file
`PTEST_NAME`	basename of the current test file (without suffix)
`PTEST_NUMBER`	rank of current test in the test file. There are in fact two independent numbering schemes: one for `EXECNOW` commands and one for regular tests (if more than one `OPT`).
`PTEST_RESULT`	shorthand alias to `@PTEST_DIR@/result@PTEST_CONFIG@` (the result directory dedicated to the tested configuration)
`PTEST_ORACLE`	basename of the current oracle file (macro only usable in FILTER directives)
`PTEST_DEFAULT_OPTIONS`	the default option list: `-check -no-autoload-plugins`

Table 4.2: Predefined macros for ptests

4.4 Plug-in Migration from Makefile to Dune

Target readers: *developers who have an existing plug-in for Frama-C 25 or less and want to migrate this plug-in to Frama-C 26 or more.*

Prerequisite: *Being familiar with the plug-in to migrate. Depending on how complex the plug-in is, it may require an advanced knowledge of the Dune build system.*

Please note that this section is a best effort procedure for making the migration as smooth as possible. If the plug-in is particularly complex, please contact us if you need some help for migrating it.

4.4.1 Files organization changes

Due to the way `dune` operates, it is preferable to work on the migration starting from a "clean" directory, without compilation and tests (in `result` directory of the test suites) artifacts. Otherwise, `dune` will complain about conflicts between files being both present in the original source directory and the target of a compilation rule.

Previously for a plug-in named `Plugin`, only the file `Plugin.mli` was necessary to expose the signature of the plug-in. Now, one has to also provide an implementation file `Plugin.ml`. On the other hand, it is not necessary that it begins with a capital letter anymore: you can have `plugin.ml` and `plugin.mli`. If these files are not present, all functions included in the modules constituting the plug-in will be exported.

For most plug-ins, the `autoconf` and `configure` will be useless. In particular, Frama-C does not provide any `autoconf` and `configure` features anymore. So for most plug-ins these files will be entirely removed (see 4.4.3). Thus, the `Makefile.in` does not exist anymore. A `Makefile` may be useful to provide some shortcuts (see Section 4.2).

If a plugin has a graphical user-interface, it is recommended to put the related files into a separate directory in the directory of the plug-in (see 4.4.4).

It was previously possible to indicate `.` as a test suite for ptests. In such a case, tests source files were directly in the `tests` directory. This is not possible anymore. If the plug-in tests follow this architecture, these tests should be moved in a subdirectory of `tests` and the oracles updated before starting the migration.

4.4.2 Template dune file

This basic template should be enough for most plug-ins. The next sections explain how to add more information to this file to handle some common cases.

```
( rule
  (alias frama-c-configure)
  (deps (universe))
  (action
    ( progn
        (echo "MyPlugin:" %{lib-available:frama-c-myplugin.core} "\n")
        (echo " - Frama-C:" %{lib-available:frama-c.kernel} "\n")
    )
  )
)

( library
  (optional)
```

```
  (name myplugin)
  (public_name frama-c-myplugin.core)
  (flags -open Frama_c_kernel :standard)
  (libraries frama-c.kernel)
)

( plugin
  (optional)
  (name myplugin)
    (libraries frama-c-myplugin.core) (site (frama-c plugins))
)
```

For the creation of the `dune-project` file, please refer to Section 2.3.

4.4.3 `autoconf` and `configure`

Indicating whether a plug-in is available and why (availability of the dependencies) is now handled via the `frama-c-configure` rule.

When working in the Frama-C `src/plugins` directory, enabling or disabling the plug-in at build time is done thanks to the script `dev/disable-plugins.sh`.

Plug-ins dependencies are now declared in the `dune` file. In the `libraries` field. For instance, if in the `autoconf` of the plug-in, the following lines are present:

```
plugin_require_external(myplugin,zmq)
plugin_use_external(myplugin,zarith)
plugin_require(myplugin,wp)
plugin_use(myplugin,eva)
```

The `libraries` should be now:

```
  (libraries
     frama-c.kernel
     zmq
     (select zarith_dependent.ml from
       (%{lib-available:zarith} -> zarith_dependent.ok.ml)
       ( -> zarith_dependent.ko.ml)
     )
     frama-c-wp.core
     (select eva_dependent.ml from
       (%{lib-available:frama-c-eva.core} -> eva_dependent.ok.ml)
       ( -> eva_dependent.ko.ml)
     )
  )
```

For external binaries, the keyword is `bin-available`.

In the case some file must be generated at build time, it is recommended to use a rule together with an action of generation. The executable itself can be generated from an OCaml file itself. For example:

```
(executable
 (name myconfigurator)
 (libraries str))

(rule
 (deps VERSION_FILE)
 (targets generated-file)
 (action (run ./myconfigurator.exe))
```

4.4.4 GUI migration

Just like there is a `dune` for the core features of the plug-in, there is now a `dune` file for the GUI, that depends on the core features of the plug-in and the Frama-C GUI. This file is to put in the `gui` subdirectory. Again, if there are additional dependencies, they must be indicated in the `libraries` field:

```
( library
  (name myplugin_gui)
  (public_name frama-c-myplugin.gui)
  (optional)
  (flags -open Frama_c_kernel
         -open Frama_c_gui
         -open MyPlugin :standard)
  (libraries frama-c.kernel frama-c.gui frama-c-myplugin.core)
)

(plugin (optional)
  (name myplugin-gui)
  (libraries frama-c-myplugin.gui)
  (site (frama-c plugins_gui)))
```

4.4.5 Build and `Makefile.in`

Provided that the `dune` files are ready. The plug-in can now be built using the command `dune build @install`. The file `Makefile.in` can now be removed.

4.4.6 Installing Additional Files

If your plug-in has additional files to install besides the compiled files, themselves (typically, files in `share`), you can use an `install` stanza in the `dune` file, as in:

```
(install
  (package frama-c-myplugin)
  (section (site (frama-c share)))
  (files (share/myfile as frama-c-myplugin/myfile)))
```

With the stanza above, the installation of package `frama-c-myplugin` will copy `myfile` from the `share` directory of the plug-in sources into the `frama-c-myplugin` directory inside the `share` directory of Frama-C's installation. Other target sections are available (e.g. `bin` for installing an additional executable), see the `dune` manual[7] for more information.

4.4.7 Migrating tests

In the `test_config*` files, the `PLUGIN` field is now mandatory and must list the plug-in and all the plug-ins on which it directly depends on. For example the plug-in defined in our previous `dune` file, and assuming that the tests use all mandatory and optional dependencies:

```
PLUGIN: myplugin,wp,eva
OPT: ...
```

The `ptests_config` file now lists the test suites. Notice that this file was previously generated and probably list in the ignored files for the versioning system. Now, it must be versioned in such a case. Assuming that the plug-in has three suites `basic`, `eva` and `wp`. This file now contains:

[7] https://dune.readthedocs.io/en/stable/dune-files.html#install-1

```
DEFAULT_SUITES=basic eva wp
```

For most plug-ins, these modifications should be enough so that:

```
dune exec -- frama-c-ptests
dune build @ptests
```

behaves in expected way.

For more advanced usage of `ptests` please refer to Section 4.3.

4.5 Plug-in General Services

Module `Plugin` provides an access to some general services available for all plug-ins. The goal of this module is twofold. First, it helps developers to use general Frama-C services. Second, it provides to the end-user a set of features common to all plug-ins. To access to these services, you have to apply the functor `Plugin.Register`.

Each plug-in must apply this functor exactly once.

Example 4.2 *Here is how the plug-in `From` applies the functor `Plugin.Register` for its own use.*

```
include Plugin.Register
  (struct
    let name = "from analysis"
    let shortname = "from"
    let help = "functional dependencies"
   end)
```

Applying this functor mainly provides two different services. First it gives access to functions for printing messages in a Frama-C-compliant way (see Section 4.6). Second it allows to define plug-in specific parameters available as options on the Frama-C command line to the end-user (see Section 4.10).

4.6 Logging Services

Displaying results of plug-in computations to users, warning them of the hypothesis taken by the static analyzers, reporting incorrect inputs, all these tasks are easy to think about, but turn out to be difficult to handle in a readable way. As soon as your plug-in is registered (see Section 4.5 above), though, you automatically benefit from many logging facilities provided by the kernel. What is more, when logging through these services, messages from your plug-in combine with other messages from other plug-ins, in a consistent and user-friendly way.

As a general rule, you should *never* write to standard output and error channels through OCaml standard libraries. For instance, you should never use `Stdlib.stdout` and `Stdlib.stderr` channels, nor `Format.printf`-like routines.

Instead, you should use `Format.fprintf` to implement pretty-printers for your own complex data, and only the `printf`-like routines of `Log.Messages` to display messages to the user. All these routines are immediately available from your plug-in general services.

Example 4.3 *A minimal example of a plug-in using the logging services:*

```
module Self = Plugin.Register
  (struct
    let name = "foo plugin"
```

```
    let shortname = "foo"
    let help = "illustration of logging services"
  end)

let pp_dg out n =
  Format.fprintf out
    "you have at least debug %d" n

let run () =
  Self.result "Hello, this is Foo Logs !";
  Self.debug ~level:0 "Try higher debug levels (%a)" pp_dg 0;
  Self.debug ~level:1 "If you read this, %a." pp_dg 1;
  Self.debug ~level:3 "If you read this, %a." pp_dg 3;

let () = Boot.Main.extend run ()
```

Running this example, you should see:

```
$ frama-c -foo-debug 2
[foo] Hello, this is Foo Logs !
[foo] Try higher debug levels (you have at least debug 0).
[foo] If you read this, you have at least debug 1.
```

Notice that your plug-in automatically benefits from its own debug command line parameter, and that messages are automatically prefixed with the name of the plug-in. We now get into more details for an advanced usage of logging services.

4.6.1 From `printf` to `Log`

Below is a simple example of how to make a `printf`-based code towards being `Log`-compliant. The original code, extracted from the Occurrence plug-in in Frama-C-Lithium version is as follows:

```
let print_one v l =
  Format.printf "variable %s (%d):@\n" v.vname v.vid;
  List.iter
    (fun (ki, lv) →
      Format.printf " sid %a: %a@\n" d_ki ki d_lval lv)
    l

let print_all () =
  compute ();
  Occurrences.iter print_one
```

The transformation is straightforward. First you add to all your pretty-printing functions an additional `Format.formatter` parameter, and you call `fprintf` instead of `printf`:

```
let print_one fmt v l =
  Format.fprintf fmt "variable %s (%d):@\n" v.vname v.vid;
  List.iter
    (fun (ki, lv) →
      Format.fprintf fmt " sid %a: %a@\n" d_ki ki d_lval lv)
    l
```

Then, you delegate toplevel calls to `printf` towards an appropriate logging routine, with a formatting string containing the necessary `"%t"` and `"%a"` formatters:

```
let print_all () =
  compute ();
  result "%t" (fun fmt → Occurrences.iter (print_one fmt))
```

4.6.2 Log Quick Reference

The logging routines for your plug-ins consist in an implementation of the `Log.Messages` interface, which is included in the `Plugin.S` interface returned by the registration of your plug-in. The main routines of interest are:

result <*options*> `"..."`
: Outputs most of your messages with this routine. You may specify `~level:n` option to discard too detailed messages in conjunction with the *verbose* command line option. The default level is `1`.

feedback <*options*> `"..."`
: Reserved for *short* messages that gives feedback about the progression of long computations. Typically, entering a function body or iterating during fixpoint computation. The level option can be used as for `result`.

debug <*options*> `"..."`
: To be used for plug-in development messages and internal error diagnosis. You may specify `~level:n` option to discard too detailed messages in conjunction with the *debug* command line option. The default message level is `1`, and the default debugging level is `0`. Hence, without any option, `debug` discards all its messages.

warning <*options*> `"..."`
: For reporting to the user an important information about the validity of the analysis performed by your plug-in. For instance, if you locally assume non arithmetic overflow on a given statement, *etc.* Typical options include `~current:true` to localize the message on the current source location.

error <*options*> `"..."`
abort <*options*> `"..."`
: Use these routines for reporting to the user an error in its inputs. It can be used for non valid parameters, for instance. It should *not* be used for some not-yet implemented feature, however.

  The `abort` routine is a variant that raises an exception and thus immediately aborts the computation[8]. If you use `error`, execution will continue until the end of current stage or current group of the running phase (see section 4.11).

failure <*options*> `"..."`
fatal <*options*> `"..."`
: Use these routines for reporting to the user that your plug-in is now in inconsistent state or can not continue its computation. Typically, you have just discovered a bug in your plug-in!

  The `fatal` routine is a variant that raises an exception. `failure` has a behavior similar to `error` above, except that it denotes an internal error instead of a user error.

verify `(condition)` <*options*> `"..."`
: First the routine evaluates the condition and the formatting arguments, then, discards the message if the condition holds and displays a message otherwise. Finally, it returns the condition value.

  A typical usage is for example:

```
assert (verify (x> 0) "Expected a positive value (%d)" x)
```

4.6.3 Logging Routine Options

Logging routines have optional parameters to modify their general behavior. Hence their involved type in `Log.mli`.

[8] The raised exception is not supposed to be caught by anything else than the main entry point of Frama-C.

Level Option. A minimal level of verbosity or debugging can be specified for the message to be emitted. For the result and feedback channels, the verbosity level is used ; for the debug channel, the debugging level is used.

`~level:$n$` minimal level required is n.

Category Option Debug, result, and feedback output can be associated to a debugging key with the optional argument `~dkey` which takes an argument of abstract type `category`. Each category must be registered through the `register_category` function which takes an optional parameter `help` to provide a short description for this category. You can define subcategories by putting colons in the registered name. For instance `a:b:c` defines a subcategory `c` of `a:b`, itself a subcategory of `a`. The user can then choose to output debugging messages belonging to a given category (and its subcategories) with the `-plugin-msg-key <category>` option.

In order to decide whether a message should be output, both level and category options are examined:

- if neither `~level` nor `~dkey`, the effect is the same as having a level of `1` and no category.
- if only `~level` is provided, the message is output if the corresponding verbosity or debugging level is sufficient
- if only `~dkey` is used, the message is output if the corresponding category is in used (even if the verbosity or debugging level is `0`)
- if both `~level` and `~dkey` are present, the message is output if the two conditions above (sufficient verbosity or debugging level and appropriate category in use) hold. As a rule of thumb, you should refrain from using both mechanisms to control the output of a message. If some messages of a category do not have the same importance, use subcategories to give the user a finer control on what they want to see on the output.

Warning Category Option Warning output can also be associated with a category, via the `~wkey` optional argument that takes a value of abstract type `warn_category`. Warning categories are distinct from plain categories, and must be registered with the `register_warn_category` function which takes an optional parameter `help` to provide a short description for this warning category. As explained in the user manual [3], each category can be associated with a status that controls what will happen when a warning is triggered, from completely ignoring it to aborting execution. The default is to emit the warning, but this can be changed by using the `set_warn_status` function.

Source Options. By default, a message is not localized. You may specify a source location, either specifically or by using the current location of an `AST` visitor.

`~source:`s use the source position s (see `Log.mli`)
`~current:true` use the source location managed by `Current_loc` (see below).

The `Current_loc` module is used to manage the code location that is currently under focus. The current location must be set, either by Frama-C's kernel or by the plug-in themselves. In particular, `Cil` visitors update this location when visiting each node.

Example 4.4 *The code samples below show how to use and set the current location.*

```
(* [Current_loc.get()] returns the last location set in Current_loc *)
let print_current_loc () =
  Format.printf "%a@." Cil_datatype.Location.pretty (Current_loc.get ())

(* [with_loc loc f x] set the current location to loc, executes [f x] and set
   the location to its old value before returning the result of [f x]. Raised
   exceptions inside [f] will be caught and re-raised after resetting the
   location to its previous value. *)
let apply_stmt f stmt =
  Current_loc.with_loc (Cil_datatype.Stmt.loc s) f stmt
```

```
(* [with_loc_opt opt_loc f x] behaves like [with_loc] if [loc_opt] is
   [Some loc], and does not update the current location if it is [None].*)
let apply_code_annot f ca =
  Current_loc.with_loc_opt (Cil_datatype.Code_annotation.loc ca) f ca

(* Current_loc defines 2 let-binding operators which call [with_loc] and
   [with_loc_opt]. Here is a function that set the current location to the
   expr location, and reset it after the match. *)
let do_expr e =
  let open Current_loc.Operators in
  let<> UpdatedCurrentLoc = e.eloc in
  match e.enode with
  | ...

(* When we only have a [loc option], we can use the <?> operator *)
let do_code_annot f ca =
  let open Current_loc.Operators in
  let <?>  UpdatedCurrentLoc = Cil_datatype.Code_annotation.loc ca in
  f ca
```

Emission Options. By default, a message is echoed to the user *after* its construction, and it is sent to registered callbacks when emitted. See Section 4.6.4 below for more details on how to globally modify such a behavior. During the message construction, you can locally modify the emission process with the following options:

`~emitwith:`f suppresses the echo and sends the emitted event *only* to the callback function f. Listeners are not fired at all.
`~once:true` finally discards the message if the same one was already emitted before with the `~once` option.

Append Option. All logging routines have the `~append:`f optional parameter, where f is function taking a `Format.formatter` as parameter and returning unit. This function f is invoked to append some text to the logging routine. Such continuation-passing style is sometime necessary for defining new polymorphic formatting functions. It has been introduced for the same purpose than standard `Format.kfprintf`-like functions.

4.6.4 Advanced Logging Services

Message Emission

During message construction, the message content is echoed in the terminal. This echo may be delayed until message completion when `~once` has been used. Upon message completion, the message is *emitted* and sent to all globally registered hook functions, unless the `~emitwith` option has been used.

To interact with this general procedure, the plug-in developer can use the following functions defined in module `Log`:

```
val set_echo: ?plugin:string → ?kinds:kind list → bool → unit
val add_listener: ?plugin:string → ?kinds:kind list → (event → unit) → unit
```

Continuations

The logging routines take as argument a (polymorphic) formatting string, followed by the formatting parameters, and finally return unit. It is also possible to catch the generated message, and to pass it to a continuation that finally returns a value different than unit.

For this purpose, you must use the `with_<log>` routines variants. These routines take a continuation f for additional parameter. After emitting the corresponding message in the normal way, the message is passed to the continuation f. Hence, f has type $event \to \alpha$, and the log routine returns α.

For instance, you typically use the following code fragment to return a degenerated value while emitting a warning:

```
let rec fact n =
  if (n> 12) then
    with_warning (fun _ → 0) "Overflow for %d, return 0 instead" x
  else if n≤ 1 then 1 else n * fact (n-1)
```

Generic Routines

The `Log.Messages` interface provides two generic routines that can be used instead of the basic ones:

log `?kind ?verbose ?debug` <*options*> `"..."`
Emits a message with the given kind, when the verbosity and/or debugging level are sufficient.

logwith `f ?wkey ?emitwith ?once` <*options*> `"..."`
Emits a message like `warning`, and finally pass the generated `event` (or `None`) to the continuation f, and returns its result.

The default kind is `Result`, but all the other kind of message can be specified. For verbosity and debugging levels, the message is emitted when:

`log "..."`	verbosity is at least 1
`log ~verbose:`n	verbosity is at least n
`log ~debug:`n	debugging is at least n
`log ~verbose:`v `~debug:`d	*either* verbosity is at least v *or* debugging is at least d.

Channel Management

The logging services are build upon *channels*, which are basically buffered formatters to standard output extended with locking, delayed echo, and notification services.

The very safe feature of logging services is that recursive calls *are* protected. A message is only echoed upon termination, and a channel buffer is stacked only if necessary to preserve memory.

Services provided at plug-in registration are convenient shortcuts to low-level logging service onto channels. The `Log` interface allows you to create such channels for your own purposes.

Basically, *channels* ensure that no message emission interfere with each others during echo on standard output. Hence the forbidden direct access to `Stdlib.stdout`. However, `Log` interface allows you to create such channels on your own, in addition to the one automatically created for your plug-in.

new_channel *name*
This creates a new channel. There is only one channel *per* name, and the function returns the existing one if any. Plug-in channels are registered under their short-name, and the kernel channel is registered under `Log.kernel_channel_name`.

log_channel `channel ?kind ?prefix`
This routine is similar to the `log` one.

With both logging routines, you may specify a prefix to be used during echo. The available switches are:

`Label` t: use the string t as a prefix for the first echoed line of text, then use an indentation of same length for the next lines.
`Prefix` t: use the string t as a prefix for all lines of text.

`Indent` n: use an indentation of n spaces for all lines of text.

When left unspecified, the prefix is computed from the message kind and the channel name, like for plug-ins.

Output Management

It is possible to ask `Log` to redirect its output to another channel:

set_output `out flush`
: The parameters are the same than those of `Format.make_formatter`: `out` outputs a (sub)-string and `flush` actually writes the buffered text to the underlying device.

It is also possible to have a momentary direct access to `Stdlib.stdout`, or whatever its redirection is:

print_on_output `"..."`
: The routine immediately locks the output of `Log` and prints the provided message. All message echoes are delayed until the routine actually returns. Notification to listeners is not delayed, however.

print_delayed `"..."`
: This variant locks the output *only* when the first character would be written to output. This gives a chance to a message to be echoed before your text is actually written.

Remark that these two routines can *not* be recursively invoked, since they have a lock to a non-delayed output channel. This constraint is verified at runtime to avoid incorrect interleaving, and you would get a fatal error if the situation occurs.

Warning: these routine are dedicated to *expensive* output only. You get the advantage of not buffering your text before printing. But on the other hand, if you have messages to be echoed during printing, they must be stacked until the end of your printing.

You get a similar functionality with `Kernel\_function.CodeOutput.output`. This routine prints your text by calling `Log.print_delayed`, unless the command line option `-ocode` has been set. It this case, your text is written to the specified file.

4.7 The `Datatype` library: Type Values and Datatypes

Type values and *datatypes* are key notions of Frama-C. They are both provided by the `Datatype` library. An overview as well as technical details may also be found in a related article in French [18]. A short summary focusing on (un)marshaling is described in another article [6]. First, Section 4.7.1 introduces type values. Then Section 4.7.2 introduces datatypes built on top of type values.

4.7.1 Type Value

A *type value* is an OCaml value which dynamically represents a static monomorphic OCaml type τ. It gets the type τ `Type.t`. There is at most one type value which represents the type τ. Type values are used by Frama-C to ensure safety when dynamic typing is required (for instance to access to a dynamic plug-in API, see Section 4.8.2).

Type values for standard OCaml monomorphic types are provided in module `Datatype`.

Example 4.5 *The type value for type `int` is `Datatype.int` while the one for type `string` is `Datatype.string`. The former has type `int Type.t` while the latter has type `string Type.t`.*

Type values are created when building datatypes (see Section 4.7.2). There is no type value for polymorphic types. Instead, they have to be created for each instance of a polymorphic type. Functions for accessing such type values for standard OCaml polymorphic types are provided in moduleDatatype.

Example 4.6 *The type value for type `int list` is `Datatype.list Datatype.int` while the one for type `string →char →bool` is `Datatype.func2 Datatype.string Datatype.char Datatype.bool`. The former has type `int list Type.t` while the latter has type `(string →char →bool) Type.t`.*

4.7.2 Datatype

A *datatype* provides in a single module a monomorphic type and usual values over it. Its signature is `Datatype.S`. It contains the type itself, the type value corresponding to this type, its name, functions `equal`, `compare`, `hash` and `pretty` which may respectively be used to check equality, to compare, to hash and to pretty print values of this type. It also contains some other values (for instance required when marshaling). Whenever possible, a datatype implements an extensible version of `Datatype.S`, namely `Datatype.S_with_collections`. For a type τ, this extended signature additionally provides modules `Set`, `Map` and `Hashtbl` respectively implementing sets over τ, maps and hashtables indexed by elements of τ.

Datatypes for OCaml types from the standard library are provided in module `Datatype`, while those for AST's types are provided in module `Cil_datatype`. Furthermore, when a kernel module implements a datastructure, it usually implements `Datatype.S`.

Example 4.7 *The following line of code pretty prints whether two statements are equal.*

```
(* assuming the type of [stmt1] and [stmt2] is Cil_types.stmt *)
Format.fprintf
  fmt (* a formatter previously defined somewhere *)
  "statements %a and %a are %sequal"
  Cil_datatype.Stmt.pretty stmt1
  Cil_datatype.Stmt.pretty stmt2
  (if Cil_datatype.Stmt.equal stmt1 stmt2 then "" else "not ")
```

Example 4.8 *Module `Datatype.String` implements `Datatype.S_with_collections`. Thus you can initialize a set of strings in the following way.*

```
let string_set =
  List.fold_left
    (fun acc s → Datatype.String.Set.add s acc)
    Datatype.String.Set.empty
    [ "foo"; "bar"; "baz" ]
```

Building Datatypes

For each monomorphic type, the corresponding datatype may be created by applying the functor `Datatype.Make`. In addition to the type `t` corresponding to the datatype, several values must be provided in the argument of the functor. These values are properly documented in the Frama-C API. The following example introduces them in a practical way.

Example 4.9 *Here is how to define in the more precise way the datatype corresponding to a simple sum type.*

```
type ab = A | B of int
module AB =
  Datatype.Make
    (struct
```

```
    (* the type corresponding to the datatype *)
    type t = ab
    (* the unique name of the built datatype; usually the name of the
       type *)
    let name = "ab"
    (* representants of the type: a non-empty list of values of this type. It
       is only used for safety check: the best the list represents the
       different possible physical representation of the type, the best the
       check is. *)
    let reprs = [ A; B 0 ]
    (* structural descriptor describing the physical representation of the
       type. It is used when marshaling. *)
    let structural_descr =
      Structural_descr.Structure
        (Structural_desr.Sum [| [| Structural_descr.p_int |] |])
    (* equality, compare and hash are the standard OCaml ones *)
    let equal (x:t) y = x = y
    let compare (x:t) y = Stdlib.compare x y
    let hash (x:t) = Hashtbl.hash x
    (* the type ab is a standard functional type, thus copying and rehashing
       are simply identity. Rehashing is used when marshaling. *)
    let copy = Datatype.identity
    let rehash = Datatype.identity
    (* the type ab does never contain any value of type Project.t *)
    let mem_project = Datatype.never_any_project
    (* pretty printer *)
    let pretty fmt x =
      Format.pp_print_string fmt
        (match x with A → "a" | B n → "b" ^ string_of_int n)
  end)
```

Only providing an effective implementation for the values `name` and `reprs` is mandatory. For instance, if you know that you never use values of type `t` as keys of hashtable, you can define the function hash equal to the function `Datatype.undefined` , and so on. To ease this process, you can also use the predefined structure `Datatype.Undefined`.

Example 4.10 *Here is a datatype where only the function `equal` is provided.*

```
(* the same type than the one of the previous example *)
type ab = A | B of int
module AB =
  Datatype.Make
    (struct
      type t = ab
      let name = "ab"
      let reprs = [ A; B 0 ]
      include Datatype.Undefined
      let equal (x:t) y = x = y
    end)
```

One weakness of `Datatype.Undefined` is that it cannot be used in a projectified state (see Section 4.9.2) because its values cannot be serializable. In such a case, you can use the very useful predefined structure `Datatype.Serializable_undefined` which behaves as `Datatype.Undefined` but defines the values which are relevant for (un)serialization.

Datatypes of Polymorphic Types

As for type values, it is not possible to create a datatype corresponding to polymorphic types, but it is possible to create them for each of their monomorphic instances.

For building such instances, you must *not apply the functor* `Datatype.Make` *since it will create two type values for the same type (and with the same name): that is forbidden.*

Instead, you must use the functor `Datatype.Polymorphic` for types with one type parameter and the functor `Datatype.Polymorphic2` for types with two type parameters[9]. These functors takes as argument how to build the datatype corresponding each monomorphic instance.

Example 4.11 *Here is how to apply* `Datatype.Polymorphic` *corresponding to the type* `'a t` *below.*

```
type α ab = A of α | B of int
module Poly_ab =
  Datatype.Polymorphic
    (struct
      type α t = α ab
      let name ty = Type.name ty ^ " ab"
      let module_name = "Ab"
      let reprs ty = [ A ty ]
      let structural_descr d =
        Structural_descr.Structure
         (Structural_descr.Sum
           [| [| Structural_descr.pack d |]; [| Structural_descr.p_int |] |])
      let mk_equal f x y = match x, y with
        | A x, A y → f x y
        | B x, B y → x = y
        | A _, B _ | B _, A _ → false
      let mk_compare f x y = match x, y with
        | A x, A y → f x y
        | B x, B y → Stdlib.compare x y
        | A _, B _ → 1
        | B _, A _ → -1
      let mk_hash f = function A x → f x | B x → 257 * x
      let map f = function A x → A (f x) | B x → B x
      let mk_internal_pretty_code f prec_caller fmt = function
        | A x →
          Type.par
            prec_caller
            Type.Basic
            fmt
            (fun fmt → Format.fprintf fmt "A %a" (f Type.Call) x)
        | B n →
          Type.par
            prec_caller
            Type.Call
            fmt
            (fun fmt → Format.fprintf fmt "B %d" n)
      let mk_pretty f fmt x =
        mk_internal_pretty_code (fun _ → f) Type.Basic fmt x
      let mk_varname _ = "ab"
```

[9] `Polymorphic3` and `Polymorphic4` also exist in case of polymorphic types with 3 or 4 type parameters.

```
    let mk_mem_project mem f = function
      | A x → mem f x
      | B _ → false
    end)
module Ab = Poly_AB.Make

(* datatype corresponding to the type [int ab] *)
module Ab_int = Ab(Datatype.Int)

(* datatype corresponding to the type [int list ab] *)
module Ab_Ab_string = Ab(Datatype.List(Datatype.Int))

(* datatype corresponding to the type [(string, int) Hashtbl.t ab] *)
module HAb = Ab(Datatype.String.Hashtbl.Make(Datatype.Int))
```

Clearly it is a bit painful. However you probably will never apply this functor yourself. It is already applied for the standard OCaml polymorphic types like list and function (respectively `Datatype.List` and `Datatype.Function`).

4.8 Plug-in Registration and Access

In this section, we present how to register plug-ins and how to access them. Actually, there are three different ways, but the recommended one is through a `.mli` file.

Section 4.8.1 indicates how to register and access a plug-in through a `.mli` file. Section 4.8.2 details how to register and access a *standard* plug-in.

4.8.1 Registration through a `.mli` File

Target readers: *plug-in developers.*

Prerequisite: *Basic knowledge of `dune`.*

From `dune` point of view, a plug-in is simply an OCaml library. In order for plugin `B` to use a function which is declared in the interface of plugin `A`, the `dune` file of `B` must contain in its `libraries` clause `frama-c-a.core` (see Section 4.1), or more generally the `public_name` under which `A` is declared.

4.8.2 Dynamic Registration and Access

Target readers: *standard plug-ins developers.*

Dynamic registration is obsolete. Newer development should favor exporting a static API, as explained in Section 4.8.1.

The Frama-C kernel provides another way for registering a plug-in through the module `Dynamic`.

In short, you have to use the function `Dynamic.register` in order to register a value from a dynamic plug-in and you have to use function `Dynamic.get` in order to apply a function previously registered with `Dynamic.register`.

Registering a value

The signature of `Dynamic.register` is as follows.

```
val register: plugin:string → string → α Type.t → α →
unit
```

The first argument is the name of the plug-in registering the value and the second one is a binding name of the registered OCaml value. The pair (plug-in name, binding name) must not be used for value registration anywhere else in Frama-C. It is required in order for another plug-in to access to this value (see next paragraph). The third argument is the *type value* of the registered value (see Section 4.7.1). It is required for safety reasons when accessing to the registered value (see the next paragraph). The fourth argument is the value to register.

Example 4.12 *Here is how the function `run` of the plug-in `hello` of the tutorial is registered. The type of this function is `unit → unit`.*

```
let run () : unit = ...
let () =
  Dynamic.register
    ~plugin:"Hello"
    "run"
    (Datatype.func Datatype.unit Datatype.unit)
    run
```

If the string `"Hello.run"` is already used to register a dynamic value, then the exception `Type.AlreadyExists` is raised during plug-in initialization (see Section 4.11).

The function call `Datatype.func Datatype.unit Datatype.unit` returns the type value representing `unit → unit`. Note that, because of the type of `Dynamic.register` and the types of its arguments, the OCaml type checker complains if the third argument (here the value `run`) has not the type `unit → unit`.

Accessing to a registered value

The signature of function `Dynamic.get` is as follows.

```
val get: plugin:string → string → α Type.t → α
```

The arguments must be the same than the ones used at value registration time (with `Dynamic.register`). Otherwise, depending on the case, you will get a compile-time or a runtime error.

Example 4.13 *Here is how the previously registered function `run` of `Hello` may be applied.*

```
let () =
  Dynamic.get
    ~plugin:"Hello"
    "run"
    (Datatype.func Datatype.unit Datatype.unit)
    ()
```

The given strings and the given type value must be the same than the ones used when registering the function. Otherwise, an error occurs at runtime. Furthermore, the OCaml type checker will complain either if the third argument (here `()`) is not of type `unit` or if the returned value (here `()` also) is not of type `unit`.

The above-mentioned mechanism requires access to the type value corresponding to the type of the registered value. Thus it is not possible to access a value of a plug-in-defined type. For solving this issue, Frama-C provides a way to access type values of plug-in-defined types in an abstract way through the functor `Type.Abstract`.

Example 4.14 *There is no current example in the Frama-C open-source part, but consider a plug-in which provides a dynamic API for callstacks as follows.*

```
module P =
  Plugin.Register
    (struct
     let name = "Callstack"
     let shortname = "Callstack"
     let help = "callstack library"
    end)

(* A callstack is a list of a pair (kf * stmt) where [kf] is the kernel
   function called at statement [stmt]. Building the datatype also
   creates the corresponding type value [ty]. *)
type callstack = (Kernel_function.t * Cil_datatype.Stmt.t) list

(* Implementation *)
let empty = []
let push kf stmt stack = (kf, stmt) :: stack
let pop = function [] -> [] | _ :: stack -> stack
let rec print = function
  | [] -> P.feedback ""
  | (kf, stmt) :: stack ->
    P.feedback "function %a called at stmt %a"
      Kernel_function.pretty kf
      Cil_datatype.Stmt.pretty stmt;
    print stack

(* Type values *)
let kf_ty = Kernel_function.ty
let stmt_ty = Cil_datatype.Stmt.ty

module D =
  Datatype.Make
    (struct
     type t = callstack
     let name = "Callstack.t"
     let reprs = [ empty; [ Kernel_function.dummy (), Cil.dummyStmt ] ]
     include Datatype.Serializable_undefined
    end)

(* Dynamic API registration *)
let register name ty =
  Dynamic.register ~plugin:"Callstack" name ty

let empty = register "empty" D.ty empty
let push = register "push" (Datatype.func3 kf_ty stmt_ty D.ty D.ty) push
let pop = register "pop" (Datatype.func D.ty D.ty) pop
let print = register "print" (Datatype.func D.ty Datatype.unit) print
```

You have to use the functor `Type.Abstract` to access to the type value corresponding to the type of callstacks (and thus to access to the above dynamically registered functions).

```
(* Type values *)
let kf_ty = Kernel_function.ty
let stmt_ty = Cil_datatype.Stmt.ty

(* Access to the type value for abstract callstacks *)
module C = Type.Abstract(struct let name = "Callstack.t" end)

let get name ty = Dynamic.get ~plugin:"Callstack" name ty
```

```
(* mutable callstack *)
let callstack_ref = ref (get "empty" C.ty)

(* operations over this mutable callstack *)

let push_callstack =
  (* getting the function outside the closure is more efficient *)
  let push = get "push" (Datatype.func3 kf_ty stmt_ty C.ty C.ty) in
  fun kf stmt → callstack_ref ← push kf stmt !callstack_ref

let pop_callstack =
  (* getting the function outside the closure is more efficient *)
  let pop = get "pop" (Datatype.func C.ty C.ty) in
  fun () → callstack_ref ← pop !callstack_ref

let print_callstack =
  (* getting the function outside the closure is more efficient *)
  let print = get "print" (Datatype.func C.ty Datatype.unit) in
  fun () → print !callstack_ref

(* ... algorithm using the callstack ... *)
```

4.9 Project Management System

Prerequisite: *Knowledge of the* OCaml *module system and labels.*

In Frama-C, a key notion detailed in this section is the one of *project*. An overview as well as technical details may also be found in a related article in French [17]. Section 4.9.1 first introduces the general principle of project. Section 4.9.2 introduces the notion of *states*. State registration is detailed in Sections 4.9.3 and 4.9.4. The former is dedicated to standard (high-level) registration, while the latter is dedicated to low-level registration. Then Section 4.9.5 explains how to use projects. Finally Section 4.9.6 details state selections.

4.9.1 Overview and Key Notions

A *project* groups together an AST with the set of global values attached to it. Such values are called *states*. Examples of states are parameters (see Section 4.10) and results of analyses (Frama-C extensively uses memoization [14, 15] in order to prevent running analyses twice).

In a Frama-C session, several projects (and thus several ASTs) can exist at the same time. The project library ensures project non-interference: modifying the value of a state in a project does not impact any value of any state in any other project. To ensure this property, each state must be registered in the project library as explained in Sections 4.9.3 and 4.9.4. Relations between states and projects are summarized in Figure 4.1.

To ease development, Frama-C maintains a current project (`Project.current ()`): all operations are automatically performed on it. For instance, calling `Ast.get ()` returns the Frama-C AST of the current project. It is also possible to access values in others projects as explained in Section 4.9.5.

4.9.2 State: Principle

If some data should be part of the state of Frama-C*, you must register it in the project library (see Sections 4.9.3 and 4.9.4).*

Projects / **States**	Project p_1	...	Project p_n
AST a	value of a in p_1	...	value of a in p_n
data d_1	value of d_1 in p_1	...	value of d_1 in p_n
...	...	...	...
data d_m	value of d_m in p_1	...	value of d_m in p_n

Figure 4.1: Representation of the Frama-C State.

Here we first explain what are the functionalities of each state and then we present the general principle of registration.

State Functionalities

Whenever you want to attach some data (*e.g.* a table containing results of an analysis) to an AST, you have to register it as an internal state. The main functionalities provided to each internal state are the following.

– It is automatically updated whenever the current project changes. Your data are thus always consistent with the current project. More precisely, you still work with your global data (for instance, a hashtable or a reference) as usual in OCaml. The project library silently changes the data when required (usually when the current project is changing). The extra cost due to the project system is usually an extra indirection. Figure 4.2 summarizes these interactions between the project library and your state.

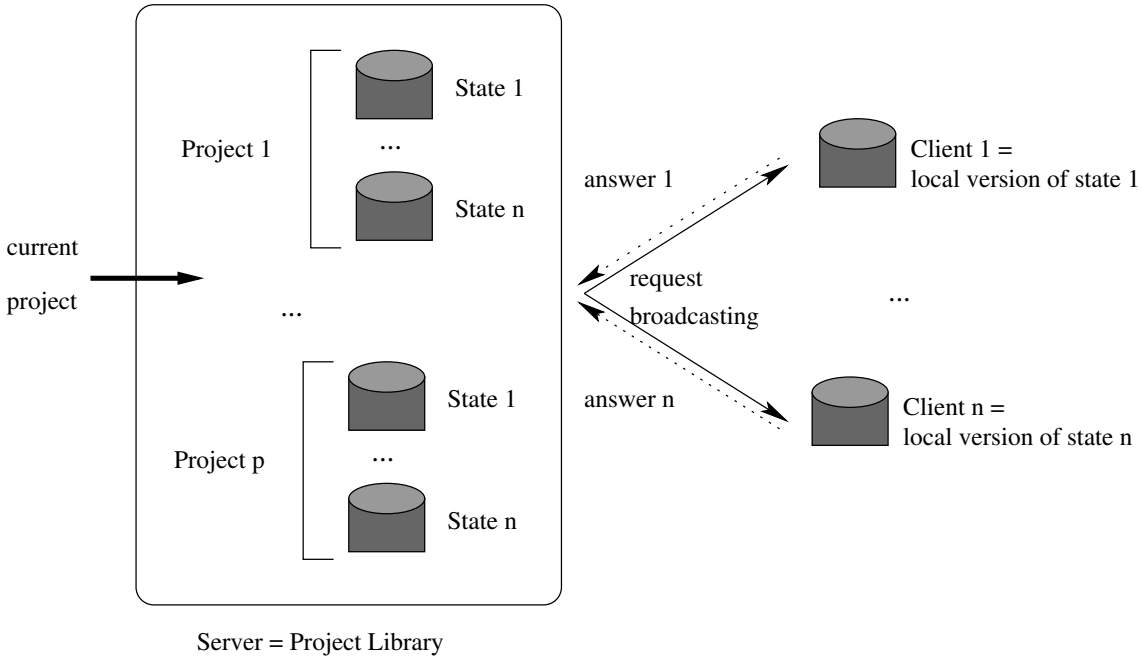


Figure 4.2: Interaction between the project library and your registered global data.

– It is part of the information saved on disk for restoration in a later session.
– It may be part of a *selection* which is a consistent set of states. With such a selection, you can control on which states project operations are consistently applied (see Section 4.9.6). For example, it is possible to clear all the states which depend on value analysis results.

– It is possible to ensure inter-analysis consistency by setting state dependencies. For example, if the entry point of the analyzed program is changed (using `Globals.set_entry_point`), all the results of analyses depending on it (like value analysis' results) are automatically reset. If such a reset were not performed, the results of the value analysis would not be consistent anymore with the current entry point, leading to incorrect results.

Example 4.15

```
Eva.Analysis.compute();
Kernel.feedback "%B" (Eva.Analysis.is_computed ()); (* true *)
Globals.set_entry_point "f" true;
Kernel.feedback "%B" (Eva.Analysis.is_computed ()); (* false *)
```

As the value analysis has been automatically reset when setting the entry point, the above code outputs

```
[kernel] true
[kernel] false
```

State Registration: Overview

For registering a new state, functor `State_builder.Register` is provided. Its use is described in Section 4.9.4 but it is a low-level functor which is usually difficult to apply in a correct way. Higher-level functors are provided to the developer in modules `State_builder` and `Cil_state_builder` that allow the developer to register states in a simpler way. They internally apply the low-level functor in the proper way. Module `State_builder` provides state builders for standard OCaml datastructures like hashtables whereas `Cil_state_builder` does the same for standard Cil datastructures (like hashtables indexed by AST statements)[10]. They are described in Section 4.9.3.

Registering a new state must be performed when the plugin is initialized. Thus, using OCaml `let module` *construct to register the new state is forbidden (except if you really know what you are doing).*

4.9.3 Registering a New State

Here we explain how to register and use a state. Registration through the use of the low-level functor `State_builder.Register` is postponed in Section 4.9.4 because it is more tricky and rarely useful.

In most non-Frama-C applications, a state is a global mutable value. One can use it to store results of analyses. For example, using this mechanism inside Frama-C to create a `state` which would memoize some information attached to statements would result in the following piece of code.

```
open Cil_datatype
type info = Kernel_function.t * Cil_types.varinfo
let state : info Stmt.Hashtbl.t = Stmt.Hashtbl.create 97
let compute_info (kf,vi) = ...
let memoize s =
  try Stmt.Hashtbl.find state s
  with Not_found → Stmt.Hashtbl.add state s (compute_info s)
let run () = ... Eva.Analysis.compute (); ... memoize some_stmt ...
```

However, if one puts this code inside Frama-C, it does not work because this state is not registered as a Frama-C state. For instance, it is never saved on the disk and its value is never changed when setting the current project to a new one. For this purpose, one has to transform the above code into the following one.

[10] These datastructures are only mutable datastructures (like hashtables, arrays and references) because global states are always mutable.

```
module State =
  Cil_state_builder.Stmt_hashtbl
    (Datatype.Pair(Kernel_function)(Cil_datatype.Varinfo))
    (struct
     let size = 97
     let name = "state"
     let dependencies = [ Eva.Analysis.self ]
    end)
let compute_info (kf,vi) = ...
let memoize = State.memo compute_info
let run () = ... Eva.Analysis.compute (); ... memoize some_stmt ...
```

A quick look on this code shows that the declaration of the state itself is more complicated (it uses a functor application) but its use is simpler. Actually what has changed?

1. To declare a new internal state, apply one of the predefined functors in modules `State_builder` or `Cil_state_builder` (see interfaces of these modules for the list of available modules). Here we use `Cil_state_builder.Stmt_hashtbl` which provides a hashtable indexed by statements. The type of values associated to statements is a pair of `Kernel_function.t` and `Cil_types.varinfo`. The first argument of the functor is then the datatype corresponding to this type (see Section 4.7.2). The second argument provides some additional information: the initial size of the hashtable (an integer similar to the argument of `Hashtbl.create`), an unique name for the resulting state and its dependencies. This list of dependencies is built upon values `self` which are called *state kind* (or simply *kind*) and are part of any state's module (part of the signature of the low-level functor `State_builder.Register`). This value represents the state itself as first-class value (like type values for OCaml types, see Section 4.7.1).
2. From outside, a state actually hides its internal representation in order to ensure some invariants: operations on states implementing hashtables do not take hashtables as arguments because they implicitly use the hidden hashtable. In our example, a predefined memo function is used in order to memoize the computation of `compute_info`. This memoization function implicitly operates on the hashtable hidden in the internal representation of `State`.

Postponed dependencies Sometimes, you want to access a state kind before defining it. That is usually the case when you have two mutually-dependent states: the dependencies of the first one provided when registering it must contain the state kind of the second one which is created by registering it. But this second registration also requires a list of dependencies containing the first state kind.

For solving this issue, it is possible to postpone the addition of a state kind to dependencies until all modules have been initialized. However, dependencies must be correct before anything serious is computed by Frama-C. So the right way to do this is the use of the function `Cmdline.run_after_extended_stage` (see Section 4.11 for advanced explanation about the way Frama-C is initialized).

Example 4.16 *Plug-in `from` creates a `State` exposed via its API.*

File ***src/plugins/from/functionwise.ml***

```
module Tbl =
  Kernel_function.Make_Table
    (Function_Froms)
    (struct
      let name = "functionwise_from"
      let size = 97
      let dependencies = [ Eva.Analysis.self ]
    end)
let self = Tbl.self
```

File **src/plugins/from/from.ml**

```
let self = Functionwise.self
```

Plug-in `pdg` *uses* `from` *for computing its own internal state. So it declares this dependency as follows.*

File **src/plugins/pdg/pdg_tbl.ml**

```
module Tbl =
  Kernel_function.Make_Table
    (PdgTypes.Pdg)
    (struct
      let name = "Pdg.State"
      let dependencies = [] (* postponed because From.self may not exist yet *)
      let size = 97
    end)
let self = Tbl.self
```

File **src/plugins/pdg/register.ml**

```
let () =
  Cmdline.run_after_extended_stage
    (fun () →
      State_dependency_graph.add_codependencies
        ~onto:Pdg_tbl.self
        [ From.self ])
```

Dependencies over the AST Most internal states depend directly or indirectly on the AST of the current project. However, the AST plays a special role as a state. Namely, it can be changed in place, bypassing the project mechanism. In particular, it is possible to add globals. Plugins that perform such changes should inform the kernel when they are done using `Ast.mark_as_changed` or `Ast.mark_as_grown`. The latter must be used when the only changes are additions, leaving existing nodes untouched, while the former must be used for more intrusive changes. In addition, it is possible to tell the kernel that a state is "monotonic" with respect to AST changes, in the sense that it does not need to be cleared when nodes are added (the information that should be associated to the new nodes will be computed as needed). This is done with the function `Ast.add_monotonic_state`. `Ast.mark_as_grown` will not touch such a state, while `Ast.mark_as_changed` will clear it.

4.9.4 Direct Use of Low-level Functor `State_builder.Register`

Functor `State_builder.Register` is the only functor which really registers a state. All the others internally use it. In some cases (*e.g.* if you define your own mutable record used as a state), you have to use it. Actually, in the Frama-C kernel, there are only three direct uses of this functor over thousands of state registrations: so you will certainly never use it.

This functor takes three arguments. The first and the third ones respectively correspond to the datatype and to information (name and dependencies) of the states: they are similar to the corresponding arguments of the high-level functors (see Section 4.9.3).

The second argument explains how to handle the *local version* of the state under registration. Indeed here is the key point: from the outside, only this local version is used for efficiency purposes (remember Figure 4.2). It would work even if projects do not exist. Each project knows a *global version*. The project management system *automatically* switches the local version when the current project changes in order to conserve a physical equality between local version and current global version. So, for this purpose, the second argument provides a type `t` (type of values of the state) and five functions `create` (creation of a new fresh state), `clear` (cleaning a state), `get` (getting a state), `set` (setting a state) and `clear_some_projects` (how to clear each value of type `project` in the state if any).

The following invariants must hold:[11]

$$\texttt{create ()}\ \textit{returns a fresh value} \tag{4.1}$$

$$\forall \texttt{p}\ \textit{of type}\ \texttt{t},\ \texttt{create ()} = \texttt{(clear p; set p; get ())} \tag{4.2}$$

$$\forall \texttt{p}\ \textit{of type}\ \texttt{t},\ \texttt{copy p}\ \textit{returns a fresh value} \tag{4.3}$$

$$\forall \texttt{p1}, \texttt{p2}\ \textit{of type}\ \texttt{t}\ \textit{such that}\ \texttt{p1 != p2},\ \texttt{(set p1; get ()) != p2} \tag{4.4}$$

Invariant 4.1 ensures that there is no sharing with any value of a same state: so each new project has got its own fresh state. Invariant 4.2 ensures that cleaning a state resets it to its initial value. Invariant 4.3 ensures that there is no sharing with any copy. Invariant 4.4 is a local independence criterion which ensures that modifying a local version does not affect any other version (different from the global current one) by side effects.

Example 4.17 *To illustrate this, we show how functor* `State_builder.Ref` *(registering a state corresponding to a reference) is implemented.*

```
module Ref
  (Data: Datatype.S)
  (Info: sig include Info val default: unit → Data.t end) =
struct
  type data = Data.t
  let create () = ref Info.default
  let state = ref (create ())
```

Here we use an additional reference: our local version is a reference on the right value. We can use it in order to safely and easily implement `get` *and* `set` *required by the registration.*

```
  include Register
    (Datatype.Ref(Data))
    (struct
     type t = data ref (* we register a reference on the given type *)
     let create = create
     let clear tbl = tbl ← Info.default
     let get () = !state
     let set x = state ← x
     let clear_some_projects f x =
       if Data.mem_project f !x then begin clear x; true end else false
    end)
    (Info)
```

For users of this module, we export "standard" operations which hide the local indirection required by the project management system.

```
  let set v = !state ← v
  let get () = !(!state)
  let clear () = !state ← Info.default
end
```

As you can see, the above implementation is error prone; in particular it uses a double indirection (reference of reference). So be happy that higher-level functors like `State_builder.Ref` *are provided which hide such implementations from you.*

[11] As usual in OCaml, = stands for *structural* equality while == (resp. !=) stands for *physical* equality (resp. disequality).

4.9.5 Using Projects

As said before, all operations are done by default on the current project. But sometimes plug-in developers have to explicitly use another project, for example when the AST is modified (usually through the use of a copy visitor, see Section 4.14) or replaced (*e.g.* if a new one is loaded from disk).

An AST must never be modified inside a project. If such an operation is required, you must either create a new project with a new AST, usually by using `File.init_project_from_cil_file` *or* `File.init_project_from_visitor`*; or write the following line of code (see Section 4.9.6):*

```
let selection = State_selection.only_dependencies Ast.self in
Project.clear ~selection ()
```

Operations over projects are grouped together in module `Project`. A project has type `Project.t`. Function `Project.set_current` sets the current project on which all operations are implicitly performed.

Example 4.18 *Suppose that you saved the current project into file* `foo.sav` *in a previous* Frama-C *session*[12] *thanks to the following instruction.*

```
Project.save "foo.sav"
```

In a new Frama-C *session, executing the following lines of code (assuming the value analysis has never been computed previously)*

```
let print_computed () =
  Kernel.feedback "%B" (Eva.Analysis.is_computed ())
in
print_computed (); (* false *)
let old = Project.current () in
try
  let foo = Project.load ~name:"foo" "foo.sav" in
  Project.set_current foo;
  Eva.Analysis.compute ();
  print_computed (); (* true *)
  Project.set_current old;
  print_computed () (* false *)
with Project.IOError _ →
  Kernel.abort "error while loading"
```

displays

```
[kernel] false
[kernel] true
[kernel] false
```

This example shows that the value analysis has been computed only in project `foo` *and not in project* `old`*.*

An important invariant of Frama-C *is: if p is the current project before running an analysis, then p will be the current project after running it. It is the responsibility of any plug-in developer to enforce this invariant for his/her own analysis.*

[12] A *session* is one execution of Frama-C (through `frama-c` or `frama-c-gui`).

To be sure to enforce the above-mentioned invariant, the project library provides an alternative to the use of `Project.set_current`: `Project.on` applies an operation on a given project without changing the current project (*i.e.* locally switch the current project in order to apply the given operation and, afterwards, restore the initial context).

Example 4.19 *The following code is equivalent to the one given in Example 4.18.*

```
let print_computed () =
  Value_parameters.feedback "%B" (Eva.Analysis.is_computed ())
in
print_computed (); (* false *)
try
  let foo = Project.load ~name:"foo" "foo.sav" in
  Project.on foo
    (fun () → Eva.Analysis.compute (); print_computed () (* true *)) ();
  print_computed () (* false *)
with Project.IOError _ →
  exit 1
```

It displays

```
false
true
false
```

4.9.6 Selections

Most operations working on a single project (*e.g.* `Project.clear` or `Project.on`) have an optional parameter `selection` of type `State_selection.t`. This parameter allows the developer to specify on which states the operation applies. A *selection* is a set of states which allows the developer to consistently handle state dependencies.

Example 4.20 *The following statement clears all the results of the value analysis and all its dependencies in the current project.*

```
let selection = State_selection.with_dependencies Eva.Analysis.self in
Project.clear ~selection ()
```

The selection explicitly indicates that we also want to clear all the states which depend on the value analysis' results.

Use selections carefully: if you apply a function `f` on a selection `s` and `f` handles a state which does not belong to `s`, then the computed result by Frama-C *is potentially incorrect.*

Example 4.21 *The following statement applies a function `f` in the project `p` (which is not the current one). For efficiency purposes, we restrict the considered states to the command line options (see Section 4.10).*

```
Project.on ~selection:(Parameter_state.get_selection ()) p f ()
```

This statement only works if `f` only handles values of the command line options. If it tries to get the value of another state, the result is unspecified and all actions using any state of the current project and of project `p` also become unspecified.

4.10 Command Line Options

Prerequisite: *Knowledge of the* OCaml *module system.*

Values associated with command line options are called *parameters*. The parameters of the Frama-C kernel are stored in module `Kernel` while the plug-in specific ones have to be defined in the plug-in source code.

4.10.1 Definition

In Frama-C, a parameter is represented by a value of type `Typed_parameter.t` and by a module implementing the signature `Parameter_sig.S`. The first representation is a low-level one required by emitters (see Section 4.15) and the GUI. The second one provides a high-level API: each parameter is indeed a state (see Section 4.9.2). Several signatures extending `Parameter_sig.S` are provided in order to deal with the usual parameter types. For example, there are signatures `Parameter_sig.Int` and `Parameter_sig.Bool` for integer and boolean parameters. Mostly, these signatures provide getters and setters for modifying parameter values.

Implementing such an interface is very easy thanks to a set of functors provided by the output module of `Plugin.Register`. Indeed, you have just to choose the right functor according to your option type and potentially the wished default value. Below are some examples of such functors (see the signature `Parameter_sig.Builder` for an exhaustive list).

1. `False` (resp. `True`) builds a boolean option initialized to `false` (resp. `true`).
2. `Int` (resp. `Zero`) builds an integer option initialized to a specified value (resp. to `0`).
3. `String` (resp. `Empty_string`) builds a string option initialized to a specified value (resp. to the empty string `""`).
4. `String_set` builds an option taking a set of strings in argument (initialized to the empty set).
5. `Kernel_function_set` builds an option taking a set of kernel functions in argument (initialized to the empty set).

Each functor takes as argument (at least) the name of the command line option corresponding to the parameter and a short description for this option.

Example 4.22 *The parameter corresponding to the option `-occurrence` of the plug-in `occurrence` is the module `Print` (defined in the file `src/plugins/occurrence/options.ml`). It is implemented as follows.*

```
module Print =
  False
    (struct
      let option_name = "-occurrence"
      let help = "print results of occurrence analysis"
    end)
```

So it is a boolean parameter initialized by default to `false`. The declared interface for this module is simply

```
module Print: Parameter_sig.Bool
```

Another example is the parameter corresponding to the option `-impact-annot` of the plug-in `impact`. This parameter is defined by the module `Annot` (defined in the file `src/plugins/impact/options.ml`). It is implemented as follows.

```
module Annot =
  Kernel_function_set
    (struct
      let option_name = "-impact-annot"
```

```
    let arg_name = "f1, ..., fn"
    let help = "use the impact annotations in the code of functions f1,...,fn"
  end)
```

Thus it is a set of `kernel_functions` *initialized by default to the empty set.* Frama-C *uses the field* `arg_name` *in order to print the name of the argument when displaying help. The field* `help` *is the help message itself. The Interface for this module is simple:*

```
module Annot: Parameter_sig.Kernel_function_set
```

Recommendation 4.1 *Parameters of a same plug-in* `plugin` *should belong to a module called* `Options`*,* `Plugin_options`*,* `Parameters` *or* `Plugin_parameters` *inside the plug-in directory.*

Using a kernel parameters or a parameter of your own plug-in is very simple: you have simply to call the function `get` corresponding to your parameter.

Example 4.23 *To know whether* Frama-C *uses unicode, just write*

```
Kernel.Unicode.get ()
```

Inside the plug-in `From`*, just write*

```
From_parameters.ForceCallDeps.get ()
```

in order to know whether callsite-wise dependencies have been required.

Using a parameter of a plug-in p in another plug-in p' requires the use of module `Dynamic.Parameter`: since the module defining the parameter is not visible from the outside of its plug-in, you have to use the dynamic API of plug-in p in which p's parameters are automatically registered (see Section 4.8.2). The module `Dynamic.Parameter` defines sub-modules which provide easy access to parameters according to their OCaml types.

Example 4.24 *Outside the plug-in* `From`*, just write*

```
Dynamic.Parameter.Bool.get "-calldeps" ()
```

in order to know whether callsite-wise dependencies have been required.

4.10.2 Tuning

It is possible to modify the default behavior of command line options in several ways by applying functions just before or just after applying the functor defining the corresponding parameter.

Functions which can be applied afterwards are defined in the output signature of the applied functor.

Example 4.25 *Here is how the option "-slicing-level" restricts the range of its argument to the interval* $[0; 3]$.

```
module Calls =
  Int
    (struct
      let option_name = "-slicing-level"
      let default = 2
      let arg_name = ""
      let help = "..." (* skipped here *)
      end)
let () = Calls.set_range ~min:0 ~max:3
```

Functions which can be applied before applying the functor are defined in the module `Parameter_customize`.

Example 4.26 *Here is how the opposite of option "-safe-arrays" is renamed into "-unsafe-arrays" (otherwise, by default, it would be "-no-safe-arrays").*

```
let () = Parameter_customize.set_negative_option_name "-unsafe-arrays"
module SafeArrays =
  True
    (struct
      let module_name = "SafeArrays"
      let option_name = "-safe-arrays"
      let help = "for arrays that are fields inside structs, assume that \
                  accesses are in bounds"
     end)
```

4.11 Initialization Steps

Prerequisite: *Knowledge of linking of* OCaml *files.*

In a standard way, Frama-C *modules are initialized in the link order which remains mostly unspecified, so you have to use side-effects at module initialization time carefully.*

This section details the different stages of the Frama-C boot process to help advanced plug-in developers interact more deeply with the kernel process. It can also be useful for debugging initialization problems.

As a general rule, plug-in routines must never be executed at link time. Any useful code, be it for registration, configuration or C-code analysis, should be registered as *function hooks* to be executed at a proper time during the Frama-C boot process. In general, registering and executing a hook is tightly coupled with handling the command line parameters.

The parsing of the command line parameters is performed in several *phases* and *stages* , each one dedicated to specific operations. Following the general rule stated at the beginning of this section, even the kernel services of Frama-C are internally registered as hooks routines to be executed at a specific stage of the initialization process, among plug-ins ones.

From the plug-in developer point of view, the hooks are registered by calling the `run_after_`*xxx*`_stage` routines in `Cmdline` module and `extend` routine in the `Boot.Main` module.

The initialization phases and stages of Frama-C are described below, in their execution order.

A – **The Initialization Stage:** this stage initializes Frama-C compilation units, following some *partially* specified order. More precisely:

1. the architecture dependencies depicted on Figure 3.1 (cf. p. 35) are respected. In particular, the kernel services are linked first, *then* the kernel integrated types for plug-ins, and *finally* the plug-ins are linked in unspecified order;
2. when the GUI is present, for any plug-in p, the non-gui modules of p are always linked *before* the gui modules of p;
3. finally, the module `Boot` is linked at the very end of this stage.

Plug-in developers cannot customize this stage. In particular, the module `Cmdline` (one of the first linked modules, see Figure 3.1) performs a very early configuration stage, such as setting the global verbosity and debugging levels.

B – **The Early Stage:** this stage initializes the kernel services. More precisely:

(a) first, the parsing of command line options registered for the `Cmdline.Early` stage;
(b) then, all functions registered through `Cmdline.run_after_early_stage` are executed in an unspecified order.

C – **The Extending Stage:** the searching and loading of dynamically linked plug-ins, scripts and modules is performed at this stage. More precisely:

(a) the command line options registered for the `Cmdline.Extending` stage are treated, such as `-load-module`
(b) the hooks registered through `Cmdline.run_during_extending_stage` are executed. Such hooks include kernel function calls for searching, loading and linking the various plug-ins and scripts compilation units, with respect to the command line options parsed during stages B – and C –.

D – **The Running Phase:** the command line is split into several groups of command line arguments, each of them separated by an option `-then` or an option `-then-on` p (thus if there is n occurrences of `-then` or `-then-on` p, then there are $n+1$ groups). For each group, the following stages are executed in sequence: all the stages are executed on the first group provided on the command line, then they are executed on the second group, and so on.

1. **The Extended Stage:** this step is reserved for commands which require that all plug-ins are loaded but which must be executed very early. More precisely:

   (a) the command line options registered for the `Cmdline.Extended` stage are treated, such as `-verbose-*` and `-debug-*`;
   (b) the hooks registered through `Cmdline.run_after_extended_stage`. Most of these registered hooks come from postponed internal-state dependencies (see Section 4.9.3).

   Remark that both statically and dynamically linked plug-ins have been loaded at this stage. Verbosity and debug level for each plug-in are determined during this stage.
2. **The Exiting Stage:** this step is reserved for commands that make Frama-C exit before starting any analysis at all, such as printing help information:

   (a) the command line options registered for the `Cmdline.Exiting` stage are treated;
   (b) the hooks registered through `Cmdline.run_after_exiting_stage` are executed in an unspecified order. All these functions should do nothing (using `Cmdline.nop`) or raise `Cmdline.Exit` for stopping Frama-C quickly.
3. **The Loading Stage:** this is where the initial state of Frama-C can be replaced by another one. Typically, it would be loaded from disk through the `-load` option. As for the other stages:

   (a) first, the command line options registered for the `Cmdline.Loading` stage are treated;
   (b) then, the hooks registered through `Cmdline.run_after_loading_stage` are executed in an unspecified order. These functions actually change the initial state of Frama-C with the specified one. The Frama-C kernel verifies as far as possible that only one new-initial state has been specified.

   Normally, plug-ins should never register hooks for this stage unless they actually set a different initial state than the default one. In such a case:

   > *They must call the function `Cmdline.is_going_to_load` while initializing.*
4. **The Configuring Stage:** this is the usual place for plug-ins to perform special initialization routines if necessary, *before* having their main entry points executed. As for previous stages:

   (a) first, the command line options registered for the `Cmdline.Configuring` stage are treated. Command line parameters that do not begin by a hyphen (character `'-'`) are *not* options and are treated as C files. Thus they are added to the list of files to be preprocessed or parsed for building the `AST` (on demand);
   (b) then, the hooks registered through `Cmdline.run_after_configuring_stage` are executed in an unspecified order.
5. **The Setting Files Stage:** this stage sets the C files to analyze according to those indicated

on the command line. More precisely:

(a) first, each argument of the command line which does not begin by a hyphen (character `'-'`) is registered for later analysis;
(b) then, the hooks registered through `Cmdline.run_after_setting_files` are executed in an unspecified order.

6. **The Main Stage:** this is the step where plug-ins actually run their main entry points registered through `Boot.Main.extend`. For all intents and purposes, you should consider that this stage is the one where these hooks are executed.

4.12 Customizing the AST creation

Prerequisite: *None.*

Plug-ins may modify the way source files are transformed into the AST over which the analyses are performed. Customization of the front-end of Frama-C can be done at several stages.

A – **Parsing:** this stage takes care of converting an individual source file into a parsed AST (a.k.a Cabs, which differs from the type-checked AST on which most analyses operate). By default, source files are treated as C files, possibly needing a preprocessing phase. It is possible to tell Frama-C to use another parser for files ending with a given suffix by registering this parser with the `File.new_file_type` function. Suffixes `.h`, `.i`, `.c` and `.ci` are reserved for Frama-C kernel. The registered parser is supposed to return a pair consisting of a type-checked AST (`Cil_types.file`) and a parsed AST (`Cabs.file`). The former can be obtained from the latter with the `Cabs2cil.convFile` function, which guarantees that the resulting `Cil_types.file` respects all invariants expected by the Frama-C kernel.

B – **Type-checking:** a normal `Cabs.file` (*i.e.* not obtained through a custom parsing function) can be transformed before being type-checked. Transformation hooks are registered through `Frontc.add_syntactic_transformation`.

C – **After linking:** Once all source files have been processed, they are all linked together in a single AST. Transformations can be performed on the resulting AST at two stages:

1. before clean-up (*i.e.* removal of useless temporary variables and prototypes that are never called). At that stage, global tables indexing information related to the AST have not yet been filled.
2. after clean-up. At this stage, index tables are filled, and can thus be used. On the other hand, the transformation must take care itself of keeping in sync the AST and the tables

Registering a transformation for this stage is done through the function `File.add_code_transformation_before_cleanup` (respectively `File.add_code_transformation_after_cleanup`). If such a transformation modify the control-flow graph of a function `f`, in particular by adding statements, it must call `File.must_recompute_cfg`, in order to have the graph recomputed afterwards.

4.13 Customizing the machine model

Prerequisite: *None.*

4.13.1 Generating a custom model

Several aspects of the C standard that are implementation-defined, such as the width of standard integer types, endianness, signedness of the `char` type, etc., as well as a few compiler and architecture specific features, can be customized using a `machdep` configuration, defining a new machine model.

Machine models are described as `YAML` files, following the `machdeps/machdep-schema.yaml` schema in frama-c installed files. Predefined machdeps are also located in the `machdeps` directory of Frama-C's `SHARE` directory and can be used as reference for defining new `machdeps` by hand. It is also possible to automatically generate an `YAML` file with the `make_machdep.py` script in the `machdeps/make_machdep` directory. This script requires a C11-compliant cross-compiler for the architecture you want to describe. Its main options are:

`-compiler <c>`: the cross-compiler to be used for generating the machdep.

`-cpp-arch-flags <flag>`: an option given to the compiler for selecting the desired architecture. Multiple occurrences of this option can occur if you want to pass several options.

`-o <file>`: put the generated YAML into the given file (default is to use standard output).

`-help`: outputs the list of all options of the script.

Note that for some compiler setups, notably for non-POSIX compilation targets, the script may fail to find an appropriate value for some fields and will instead put some default value. In that case, warnings will give the names of the problematic fields. Since the issue likely stems from the fact that the corresponding C feature is not supported on the compilation target in the first place, in practice, such feature is not expected to be found in code written for such target. However, users are invited to review the generated YAML and provide more appropriate values for these fields if needed.

In order to communicate machine-related information to the preprocessor (notably the value of standard macros), Frama-C generates a specific header, `__fc_machdep.h`, that is automatically included by the standard headers from Frama-C's standard C library. Field `custom_defs` of the YAML file allows customizing this header (see next section for more information).

4.13.2 Machdep record fields

Each field of the machdep is succintly described in the `machdep-schema.yaml` file. We present below a thorough description of each field.

Meta-data

`version` : human-readable textual description of the machdep.

`machdep_name` : name of the machdep, must only contain alphanumeric characters or underscore (_). If it is e.g. `custom_name`, the generated header will define a macro of the form `__FC_MACHDEP_CUSTOM_NAME`.

`compiler` : defines whether special compiler-specific extensions will be enabled. It should be one of the strings below:

`msvc` : enables `Machine.msvcMode`, that is, MSVC (Visual Studio)-specific extensions;
`gcc` : enables `Machine.gccMode`, that is, GCC-specific extensions;
`generic` (or any other string): no special compiler-specific extensions.

Note that some compiler extensions, such as attributes, are always enabled.

`cpp_arch_flags` : list of arguments used by the compiler to select the corresponding architecture, e.g. `["-m32"]` for a 32-bit machdep. Older versions (up to 26.x - Iron) of Frama-C did pass these flags to the preprocessor, in order for it to define a set of built-in macros related to said

architecture. Current versions do not use this field, and rely on `custom_defs` containing the appropriate definitions. Note that, in practice, very few programs rely on such predefined macros, such as `__x86_64` and `__i386`.

Standard sizes and alignment constraints

`sizeof_short` : size (in bytes) of the `short` type.

`sizeof_int` : size (in bytes) of the `int` type.

`sizeof_long` : size (in bytes) of the `long` type.

`sizeof_longlong` : size (in bytes) of the `long long` type. Note that machdeps (for compiler `"gcc"` in particular) must always have at least one type that is 8 bytes wide, which is typically `long long`.

`sizeof_ptr` : size (in bytes) of an object (non-function) pointer.

`sizeof_float` : size (in bytes) of a single-precision floating point. In implementations compliant with ISO/IEC/IEEE 60559 - IEEE 754, this is always 4.

`sizeof_double` : size (in bytes) of a double-precision floating point. In implementations compliant with ISO/IEC/IEEE 60559 - IEEE 754, this is always 8.

`sizeof_longdouble` : size (in bytes) of a `long double` floating point. Note: type `long double` is currently not supported by existing Frama-C plugins, but this field exists for future expansion, and to compute `sizeof` of aggregates properly.

`sizeof_void` : the result of evaluating `sizeof(void)` by the compiler (or negative if unsupported).

`sizeof_fun` : the result of evaluating `sizeof(f)`, where `f` is a function (*not* a function pointer) by the compiler (or negative if unsupported).

`alignof_short` : the result of evaluating `_Alignof(short)`.

`alignof_int` : the result of evaluating `_Alignof(int)`.

`alignof_long` : the result of evaluating `_Alignof(long)`.

`alignof_longlong` : the result of evaluating `_Alignof(long long)`.

`alignof_ptr` : the result of evaluating `_Alignof(char*)` (or any other pointer, including function pointers).

`alignof_float` : the result of evaluating `_Alignof(float)`.

`alignof_double` : the result of evaluating `_Alignof(double)`.

`alignof_longdouble` : the result of evaluating `_Alignof(long double)`.

`alignof_str` : the result of evaluating `_Alignof("a")` (a literal string).

`alignof_fun` : the result of evaluating `_Alignof(f)`, where `f` is a function (or negative if unsupported).

`alignof_aligned` : the default alignment of a type having the `aligned` attribute (or 1 if unsupported). This corresponds to the default alignment when using `#pragma packed()` without a numeric argument.

Standard types

int_fast8_t : a string containing the actual type that `int_fast8_t` expands to. Usually `signed char`.

int_fast16_t : a string containing the actual type that `int_fast16_t` expands to. Usually `int` or `long`.

int_fast32_t : a string containing the actual type that `int_fast_32_t` expands to. Usually `int` or `long`.

int_fast64_t : a string containing the actual type that `int_fast64_t` expands to. Usually `long` or `long long`.

uint_fast8_t : a string containing the actual type that `uint_fast8_t` expands to. Usually `unsigned char`.

uint_fast16_t : a string containing the actual type that `uint_fast16_t` expands to. Usually `unsigned int` or `unsigned long`.

uint_fast32_t : a string containing the actual type that `uint_fast_32_t` expands to. Usually `unsigned int` or `unsigned long`.

uint_fast64_t : a string containing the actual type that `uint_fast64_t` expands to. Usually `unsigned long` or `unsigned long long`.

intptr_t : a string containing the actual type that `intptr_t` expands to, e.g. `long`

uintptr_t : a string containing the actual type that `uintptr_t` expands to, e.g. `unsigned long`

size_t : a string containing the actual type that `size_t` expands to, e.g. `unsigned long`.

ssize_t : a string containing the actual type that `ssize_t` expands to, e.g. `long`

wchar_t : a string containing the actual type that `wchar_t` expands to. If unsupported, you can use `int`.

ptrdiff_t : a string containing the actual type that `ptrdiff_t` expands to. If unsupported, you can use `int`.

sig_atomic_t : a string containing the actual type that `sig_atomic_t` expands to (i.e. an integer type that can be accessed atomically even in presence of interrupts).

time_t : a string containing the actual type that `time_t` expands to (i.e. an integer type that can hold time values in seconds).

wint_t : a string containing the actual type that `wint_t` expands to (i.e. an integer type capable of holding any `wchar_t` and `WEOF`)

Standard macros Note that all fields described in this paragraph have `string` values, even if they denote numeric constants. In order to avoid errors when loading the YAML file, you can force them to be considered as strings by enclosing them between single or double quotes, as in `eof: '(-1)'` (as a sidenote, negative values should be enclosed in parentheses, in order to ensure safe macro expansions).

bufsiz : value of the `BUFSIZ` macro, i.e. the size of buffers used by I/O functions in `stdio.h`.

eof : value of the `EOF` macro, the value returned by input functions in `stdio.h` to indicate the end of the stream.

errno : list of possible errors with their numeric value. In order to be easier to read, the content of this field is in fact written itself as an object whose fields are the name of the errors. For instance, for a machdep defining only the three errors mandated by the C standard, we would have:

```
errno:
    edom: 1
    eilseq: 2
    erange: 3
```

filename_max : value of the `FILENAME_MAX` macro, which denotes the longest name that is guaranteed to be accepted by `fopen`.

fopen_max : value of the `FOPEN_MAX` macro, which denotes the greatest number of streams that can be simultaneously opened by the program.

host_name_max : value of the `HOST_NAME_MAX` macro, which denotes the maximum length of a hostname (without terminating null byte).

l_tmpnam : value of the `L_tmpnam` macro, which denotes the maximum size of a temporary filename as returned by `tmpnam`.

mb_cur_max : value of the `MB_CUR_MAX` macro, which denotes the maximum number of bytes for a character in the current locale (usually `'1'`).

nsig : number of possible signals (non-standard macro, can be left empty if undefined for the current machdep).

path_max : value of the `PATH_MAX` macro, which denotes the maximum size (including terminating null) that can be stored in a buffer when returning a pathname.

posix_version : value of the `_POSIX_VERSION` macro. Leave empty on non-POSIX machdeps.

rand_max : value of the `RAND_MAX` macro, which denotes the maximum value returned by `rand`.

tmp_max : value of the `TMP_MAX` macro, which denotes the minimum number of temporary filenames returned by `tmpnam` that are guaranteed to be distinct.

tty_name_max : value of the `TTY_NAME_MAX` macro, which denotes the maximum length of a terminal device name (including terminating null byte).

weof : value of the `WEOF` macro, similar to `EOF`, but for wide chars.

wordsize : value of the `__WORDSIZE` macro, which denotes the length of a word on the current architecture.

Other features

char_is_unsigned : whether type `char` is unsigned.

little_endian : whether the machine is little endian or big endian[13].

has__builtin_va_list : whether `__builtin_va_list` is a (built-in) type known by the preprocessor.

__thread_is_keyword : whether `__thread` is a keyword (otherwise, it can be used as a standard identifier).

custom_defs : arbitrary text that will be appended verbatim at the end of the generated header file (this is empty in machdeps that are generated by the `make_machdep.py` script).

[13] More exotic endianness such as mixed-endian are currently unsupported.

4.14 Visitors

Prerequisite: *Knowledge of* OCaml *object programming.*

Module `Cil` offers a visitor, `Cil.cilVisitor`, that allows to traverse (parts of) an AST. It is a class with one method per type of the AST, whose default behavior is simply to call the method corresponding to its children. This is a convenient way to perform local transformations over a whole `Cil_types.file` by inheriting from it and redefining a few methods. However, the original Cil visitor is of course not aware of the internal state of Frama-C itself. Hence, there exists another visitor, `Visitor.generic_frama_c_visitor`, which handles projects in a transparent way for the user. There are very few cases where the plain Cil visitor should be used.

*Basically, as soon as the initial project has been built from the C source files (*i.e. *one of the functions* `File.init_*` *has been applied), only the* Frama-C *visitor should occur.*

There are a few differences between the two (the Frama-C visitor inherits from the Cil one). These differences are summarized in Section 4.14.6, which the reader already familiar with Cil is invited to read carefully.

4.14.1 Entry Points

Module `Cil` offers various entry points for the visitor. They are functions called `Cil.visitCil`*AstType* where *astType* is a node type in the Cil's AST. Such a function takes as argument an instance of a `cilVisitor` and an *astType* and gives back an *astType* transformed according to the visitor. The entry points for visiting a whole `Cil_types.file` (`Cil.visitCilFileCopy`, `Cil.visitCilFile` and `visitCilFileSameGlobals`) are slightly different and do not support all kinds of visitors. See the documentation attached to them in `cil.mli` for more details.

4.14.2 Methods

As said above, there is a method for each type in the Cil AST (including for logic annotation). For a given type *astType*, the method is called `v`*astType*[14], and has type *astType*→*astType'* `visitAction`, where *astType'* is either *astType* or *astType* `list` (for instance, one can transform a `global` into several ones). `visitAction` describes what should be done for the children of the resulting AST node, and is presented in the next section. In addition, some types have two modes of visit: one for the declaration and one for use. This is the case for `varinfo` (`vvdec` and `vvrbl`), `logic_var` (`vlogic_var_decl` and `vlogic_var_use`) `logic_info` (`vlogic_info_decl` and `vlogic_info_use`), `logic_type_info` (`vlogic_type_info_decl` and `vlogic_type_info_use`), and `logic_ctor_info` (`vlogic_ctor_info_decl` and `vlogic_ctor_info_use`). More detailed information can be found in `cil.mli`.

For the Frama-C *visitor, two methods,* `vstmt` *and* `vglob` *take care of maintaining the coherence between the transformed AST and the internal state of* Frama-C *. Thus they must not be redefined. One should redefine* `vstmt_aux` *and* `vglob_aux` *instead.*

4.14.3 Action Performed

The return value of visiting methods indicates what should be done next. There are six possibilities:

[14] This naming convention is not strictly enforced. For instance the method corresponding to `offset` is `voffs`.

- `SkipChildren` the visitor does not visit the children;
- `ChangeTo v` the old node is replaced by `v` and the visit stops;
- `DoChildren` the visit goes on with the children; this is the default behavior;
- `JustCopy` is only meaningful for the copy visitor. Indicates that the visit should go on with the children, but only perform a fresh copy of the nodes
- `ChangeToPost(v,f)` the old node is replaced by `v`, and `f` is applied to the result. This is however not exactly the same thing as returning `ChangeTo(f(v))`. Namely, in the case of `vglob_aux`, `f` will be applied to `v` only *after* the operations needed to maintain the consistency of Frama-C's internal state with respect to the AST have been performed. Thus, `ChangeToPost` should be used with extreme caution, as `f` could break some invariants of the kernel.
- `DoChildrenPost f` visit the children and apply the given function to the result.
- `JustCopyPost(f)` is only meaningful for the copy visitor. Performs a fresh copy of the nodes and all its children and applies `f` to the copy.
- `ChangeDoChildrenPost(v,f)` the old node is replaced by `v`, the visit goes on with the children of `v`, and when it is finished, `f` is applied to the result. In the case of `vstmt_aux`, `f` is called after the annotations in the annotations table have been visited, but *before* they are attached to the new statement, that is, they will be added to the result of `f`. Similarly, `vglob_aux` will consider the result of `f` when filling the table of globals. Note that `ChangeDoChildrenPost(x,f)` where `x` is the current node is *not* equivalent to `DoChildrenPost f`, as in the latter case, the visitor mechanism knows that it still deals with the original node.

4.14.4 Visitors and Projects

Copy visitors (see next section) implicitly take an additional argument, which is the project in which the transformed AST should be put in.

Note that the tables of the new project are not filled immediately. Instead, actions are queued, and performed when a whole `Cil_types.file` has been visited. One can access the queue with the `get_filling_actions` method, and perform the associated actions on the new project with the `fill_global_tables` method.

In-place visitors always operate on the current project (otherwise, two projects would risk sharing the same AST).

4.14.5 In-place and Copy Visitors

The visitors take as argument a `Visitor_behavior.t`, which comes in two flavors: `inplace` and `copy`. In the in-place mode, nodes are visited in place, while in the copy mode, nodes are copied and the visit is done on the copy. For the nodes shared across the AST (`varinfo`, `compinfo`, `enuminfo`, `typeinfo`, `stmt`, `logic_var`, `logic_info` and `fieldinfo`), sharing is of course preserved, and the mapping between the old nodes and their copy can be manipulated explicitly through the following modules, which define a function for each of the types above.

- `Reset` allows to reset the mappings.
- `Get` gets the copy corresponding to an old value. If the given value is not known, it behaves as the identity.
- `Memo` is similar to `Get`, except that if the given value is not known, a new binding is created.
- `Get_orig` gets the original value corresponding to a copy (and behaves as the identity if the given value is not known).
- `Set` sets a copy for a given value. Be sure to use it before any occurrence of the old value has been copied, or sharing will be lost.
- `Set_orig` sets the original value corresponding to a given copy.

Functions from the `Get_orig.`name *modules allow to retrieve additional information tied to the original AST nodes. Its result must not be modified in place (this would defeat the purpose of operating on a copy to leave the original AST untouched). Moreover, note that whenever the index used for* name *is modified in the copy, the internal state of the visitor behavior must be updated accordingly (*via *the* `Set.`name *function) for* `Get_orig.`name *to give correct results.*

The list of such indices is given Figure 4.3.

Type	Index
`varinfo`	`vid`
`compinfo`	`ckey`
`enuminfo`	`ename`
`typeinfo`	`tname`
`stmt`	`sid`
`logic_info`	`l_var_info.lv_id`
`logic_var`	`lv_id`
`fieldinfo`	`fname` and `fcomp.ckey`

Figure 4.3: Indices of AST nodes.

Last, when using a copy visitor, the actions (see previous section) `SkipChildren` *and* `ChangeTo` *must be used with care,* i.e. *one has to ensure that the children are fresh. Otherwise, the new AST will share some nodes with the old one. Even worse, in such a situation the new AST might very well be left in an inconsistent state, with uses of shared node (e.g. a* `varinfo` *for a function* `f` *in a function call) which do not match the corresponding declaration (e.g the* `GFun` *definition of* `f`*).*
When in doubt, a safe solution is to use `JustCopy` *instead of* `SkipChildren` *and* `ChangeDoChildrenPost(x,fun x -> x)` *instead of* `ChangeTo(x)`*.*

4.14.6 Differences Between the Cil and Frama-C Visitors

As said in Section 4.14.2, `vstmt` and `vglob` should not be redefined. Use `vstmt_aux` and `vglob_aux` instead. Be aware that the entries corresponding to statements and globals in Frama-C tables are considered more or less as children of the node. In particular, if the method returns `ChangeTo` action (see Section 4.14.3), it is assumed that it has taken care of updating the tables accordingly, which can be a little tricky when copying a `file` from a project to another one. Prefer `ChangeDoChildrenPost`. On the other hand, a `SkipChildren` action implies that the visit will stop, but the information associated to the old value will be associated to the new one. If the children are to be visited, it is undefined whether the table entries are visited before or after the children in the AST.

4.14.7 Example

Here is a small copy visitor that adds an assertion for each division in the program, stating that the divisor is not zero:

```
open Cil_types
open Cil

module M = Plugin.Register

(* Each annotation in Frama-C has an emitter, for traceability.
```

```
   We create thus our own, and says that it will only be used to emit code
   annotations, and that these annotations do not depend
   on Frama-C's command line parameters.
*)
let syntax_alarm =
  Emitter.create
    "Syntactic check" [ Emitter.Code_annot ] ~correctness:[] ~tuning:[]

class non_zero_divisor prj = object (self)
  inherit Visitor.generic_frama_c_visitor (Visitor_behavior.copy prj)

  (* A division is an expression: we override the vexpr method *)
  method! vexpr e = match e.enode with
    | BinOp((Div|Mod), _, denom, _) →
      (* denom might contain references to variables. Since we haven't visited
        the node yet, they're bound to the varinfo of the original project.
        we perform a plain copy, which will just ensure that they are replaced
        with varinfos of the new project: frama_c_plain_copy is a visitor that
        performs a copy, using the same correspondance tables as self. *)
      let denom = Visitor.visitFramacExpr self#frama_c_plain_copy denom in
      let logic_denom = Logic_utils.expr_to_term ~coerce:false denom in
      let assertion = Logic_const.prel (Rneq, logic_denom, Cil.lzero ()) in
      (* At this point, we have built the assertion we want to insert. It
        remains to attach it to the correct statement. The cil visitor
        maintains the information of which statement and function are
        currently visited in the [current_stmt] and [current_kf] methods,
        which return None when outside of a statement or a function , e.g.
        when visiting a global declaration. Here, it necessarily returns
        [Some]. *)
      let stmt = match self#current_kinstr with
        | Kglobal → assert false
        | Kstmt s → s
      in
      let kf = Option.get self#current_kf in
      (* The above statement and function are related to the original project.
        We need to attach the new assertion to the corresponding statement
        and function of the new project. Cil provides functions to convert a
        statement (function) of the original project to the corresponding
        one of the new project. *)
      let new_stmt = Visitor_behavior.Get.stmt self#behavior stmt in
      let new_kf = Visitor_behavior.Get.kernel_function self#behavior kf in
      (* Since we are copying the file in a new project, we cannot insert
        the annotation into the current table, but in the table of the new
        project. To avoid the cost of switching projects back and forth,
        all operations on the new project are queued until the end of the
        visit, as mentioned above. This is done in the following statement. *)
      Queue.add
        (fun () →
          Annotations.add_assert syntax_alarm ~kf:new_kf new_stmt assertion)
        self#get_filling_actions;
      DoChildren
    | _ → DoChildren
end

(* This function creates a new project initialized with the current file plus
   the annotations related to division. *)
let create_syntactic_check_project () =
```

```
  ignore
    (File.create_project_from_visitor "syntactic check" (new non_zero_divisor))

let () = Boot.Main.extend create_syntactic_check_project
```

4.15 Logical Annotations

Prerequisite: *None.*

Logical annotations set by the users in the analyzed C program are part of the AST. However others annotations (those generated by plug-ins) are not directly in the AST because it would contradict the rule "an AST must never be modified inside a project" (see Section 4.9.5).

So all the logical annotations (including those set by the users) are put in global projectified tables maintained up-to-date by the Frama-C kernel. Anytime a plug-in wants either to access to or to add/delete an annotation, it *must* use the corresponding modules or functions and not the annotations directly stored in the AST. These modules and functions are the following.

- Module `Annotations` which contains the database of annotations related to the AST (global annotations, function contracts and code annotations). Adding or deleting an annotation requires to define an emitter by `Emitter.create` first.
- Module `Property_status` should be used to get or to modify the validity status of logical properties. Modifying a property status requires to define an emitter by `Emitter.create` first. Key concepts and theoretical foundation of this module are described in an associated research paper [5].
- Module `Property` provides access to all logical properties on which property statuses can be emitted. In particular, an ACSL annotation has to be converted into a property if you want to access its property statuses.
- Modules `Logic_const`, `Logic_utils`, `Logic_parse_string` and `Logic_to_c`, contain several operations over annotations.
- Module `Populate_spec` and `Infer_assigns` which provides tools to generate missing specifications in the default behavior.

4.15.1 Specification generation

Sometimes, Frama-C and plug-ins need and use ACSL specifications to improve their performances and/or results. Thus, Frama-C's API offers a way to generate default specification if it is missing (the whole default behavior is missing or only some specific clauses). For this purpose, we can call `Populate_spec.populate_funspec` as follows:

```
Populate_spec.populate_funspec ~do_body:true kf ['Exits; 'Assigns]
```

This code generates specifications in the default behavior for the function `kf` using the selected mode (see the user manual [3] for more details about mode selection). The parameter `do_body` (which defaults to `false`) is used to choose if we want to generate specification only on prototypes or also for functions with a body. We also give the list of clauses that we want to generate. Here we only want to generate `'Assigns` and `'Exits` clauses, but `'Requires`, `'Allocates` and `'Terminates` are also available. This function can take an optional argument `loc:Cil_types.location` which is a location used when emitting missing specification warnings. By default this location is set to the location of `kf`.

The generated specifications are either generated from nothing (using the selected mode) or by combining existing clauses from other behaviors (see the user manual [3]).

4.15.2 Custom mode registration

If none of the available modes behave as needed, it is also possible to create a custom mode. Let us say we want a mode to match these tables:

	Proto	Body
exits	\false	\false
assigns	Auto[a]	\everything
allocates	—	—
terminates	—	—
requires	—	—

	Status
exits	Dont_know
assigns	True
allocates	—
terminates	—
requires	—

[a] Automatically generated using the function parameters.

For this, we need to define generation functions ans status for each clause:

```
(* Generate exits \false clauses. *)
let gen_exits _ _ =
  [ Exits, Logic_const.(new_predicate pfalse) ]

(* Generate assigns for prototypes. *)
let gen_assigns kf _ =
  if Kernel_function.has_definition kf then
    WritesAny
  else Writes (Infer_assigns.from_prototype kf)

(* Do not generate requires. *)
let gen_requires _ _ = [ ]

(* Do not generate allocates. *)
let gen_allocates _ _ = FreeAllocAny

(* Do not generate terminates. *)
let gen_terminates _ _ = None

(* Property status to be emitted for the generated clauses. *)
let status_exits = Property_status.Dont_know
let status_assigns = Property_status.True
```

Each function takes 2 parameters:

- The current `kernel_function` for which we want to generate specifications.
- The original specification of this function, before the generation.

And returns a new clause, which needs to match the type of clause we are currently generating (See `Populate_spec.mli` file for more details). The function `Infer_assigns.from_prototype` is used to generate assigns clauses using the prototype arguments.

Then we need to register the mode using `Populate_spec.register`:

```
let create_mode () =
  Populate_spec.register
     ~gen_exits ~status_exits
     ~gen_assigns ~status_assigns
     ~gen_requires
     ~gen_allocates
     ~gen_terminates
     "mymode"
```

This function registers a new mode `mymode` which can be selected using command line options `-generated-spec-mode mymode`. All parameters are optional and, if a generation function is left unspecified, `Frama_C` mode is used instead to generate the corresponding clauses (emits a warning). It is also possible to specify a property status to be emitted when a clause is generated (emits a warning if omitted). Requires are the only clause for which it is not possible to specify a status, because `Populate_spec` never tries to emit status of requires.

Then we want the mode to be available, and for this we need to register it before Frama-C's main stage (see section 4.11), for example in the configuring stage:

```
let () = Cmdline.run_after_configuring_stage create_mode
```

4.15.3 Example

This example sums up previous sections by showing all steps to register a custom mode and generate default specifications.

```
open Cil_types

(* We start by defining all our generation function. Each function takes 2
   parameters:
   - the current kernel_function for which we want to generate specifications.
   - this kernel function specifications.

   And returns a new clause, which needs to match the type of clause we are
   currently generating (See Populate_spec.mli for more details.)
*)

(* Generate exits \false clauses. *)
let gen_exits _ _ =
  [ Exits, Logic_const.(new_predicate pfalse) ]

(* Generate assigns for prototypes. *)
let gen_assigns kf _ =
  if Kernel_function.has_definition kf then
    Writes []
  else Writes (Infer_assigns.from_prototype kf)

(* Generate requires \false clauses. *)
let gen_requires _ _ = [ Logic_const.(new_predicate pfalse) ]

(* Generate allocates \nothing clauses. *)
let gen_allocates _ _ =
  FreeAlloc([],[])

(* Generate terminates \false for prototypes. *)
let gen_terminates kf _ =
  if Kernel_function.has_definition kf then
    None
  else Some(Logic_const.(new_predicate pfalse))

(* Property status to be emitted for the generated clauses. *)
let status_exits = Property_status.Dont_know
let status_assigns = Property_status.True
let status_allocates = Property_status.Dont_know
let status_terminates = Property_status.Dont_know

(* Main loop, iter on all functions and generate their specification.
```

```
   If [do_body] is false by default, and is used to enable generation for
   function with bodies.
   We also give a list of clauses, which will be generated using the selected
   mode.
*)
let run () =
  let generate_spec kf =
    Populate_spec.populate_funspec ~do_body:true kf
      ['Exits; 'Assigns; 'Requires; 'Allocates; 'Terminates]
  in
  Globals.Functions.iter generate_spec

(* This function registers a new mode "mymode" which can be selected using
   command line options. All parameters are optionnals, and if left unspecified,
   Frama_C mode will be used to generate the corresponding clauses
   (emits a warning). Status are also optionnals, but omitting them will results
   in no emition (emits a warning).
*)
let create_mode () =
  Format.printf "Registering a new spec generation mode@.";
  Populate_spec.register
    ~gen_exits ~status_exits
    ~gen_assigns ~status_assigns
    ~gen_requires
    ~gen_allocates ~status_allocates
    ~gen_terminates ~status_terminates
    "mymode"

(* It is important to register the new mode in an early stage, before our
   main loop and first calls of Populate_spec.populate_funspec done inside
   frama-c. *)
let () = Cmdline.run_after_configuring_stage create_mode

let () = Boot.Main.extend run
```

4.16 Extending ACSL annotations

Prerequisite: *Knowledge of the ACSL specification language.*

Frama-C supports the possibility of adding specific ACSL annotations in the form of special clauses. Such clauses can be of different categories, as described by `Cil_types.ext_category`.

- A contract extension will be stored in the `b_extended` field of `Cil_types.behavior`.
- A global extension will be found as a global ACSL annotation in the form of a `Cil_types.Dextended` constructor.
- A code annotation extension will be stored with the `Cil_types.AExtended` constructor. Such an extension has itself different flavors, determined by the type:
  - it can be meant to be evaluated exactly at the current program point (like an ACSL `assert`), or
  - it can be related to the next statement (or block), like an ACSL statement contract, or
  - it can be a loop extension, or
  - it can be used both as a loop extension or be related to the next (non-loop) statement.

An extension is characterized by its introducing keyword `kw`, or `loop kw` for a loop extension.

Having the same keyword for two distinct extensions is not possible, especially if they belong to different categories, as this would lead to ambiguities in the parser.

Moreover, when writing extended ACSL annotations, it is advised to prefix the keyword with the plug-in that registers it, for example `\wp::strategy`. By doing this, one ensures that when the plug-in that registers the extension is not available, the extension will be ignored (with a warning) and the parsing will continue. Note however that the plug-in name is *not* part of the extension name, so that two different plug-ins cannot register an extension with the same keyword.

Once an extension is registered with keyword `kw`, a clause of the form `kw e1,...,en;`, where each `ei` can be any syntactically valid ACSL term or predicate, will be treated by the parser as belonging to the extension `kw`.

Contract extension clauses must occur after `assumes` and `requires` clauses if any, but can be freely mixed with other behavior clauses (post-conditions, `assigns`, `frees` and `allocates`).

Similarly, in a loop annotation, `loop kw e1, ..., en;` will be treated as belonging to the `kw` extension. In case the loop annotation has a `loop variant`, the extension must occur before. Otherwise, there is no ordering constraint with other loop annotations clauses.

Global extensions can either be a standalone global annotation, or a whole block of global extensions, the latter case following the syntax of `axiomatic` blocks.

Finally, a code annotation extension must appear as a single code annotation, like any code annotation.

Code (and loop) extensions can be made specific to a set of existing behaviors using the standard ACSL `for` construction. Namely, `for bhv: loop kw e1, ..., en;` will indicate that the (loop) extension is supposed to be considered only when behavior `bhv` is active (although it is ultimately up to the plugin to decide what to do with this information).

An `acsl_extension` is a record with:

- `ext_id`: its unique ID, used in annotation tables and generated by `Logic_const.new_acsl_extension`,
- `ext_name`: the keyword that identifies the extension,
- `ext_loc`: the location of the extension in the source file,
- `ext_status`: the fact that a property status is associated to the extension, or not. It is set during extension registration,
- `ext_kind` is an `acsl_extension_kind` that can take three forms:
  - `Ext_id id` with `id` an `int` that the plugin can use to refer to the annotation in its internal state. This identifier is under the full responsibility of the plugin and will never be used by the kernel,
  - `Ext_preds preds` with `preds` a possibly empty list of predicates (traversed normally by the visitor, see section 4.14),
  - `Ext_terms terms` with `terms` a possibly empty list of terms (traversed normally by the visitor, see section 4.14).

In order for the extension to be recognized by the parser, it must be registered by one of the following functions, depending on its category.

- `Acsl_extension.register_behavior`
- `Acsl_extension.register_global`
- `Acsl_extension.register_global_block`
- `Acsl_extension.register_code_annot`
- `Acsl_extension.register_code_annot_next_stmt`
- `Acsl_extension.register_code_annot_next_loop`
- `Acsl_extension.register_code_annot_next_both`
- `Acsl_extension.register_module_importer`

Each function takes the following mandatory arguments:

- `~plugin` the plug-in that registers the extension (`"kernel"` for the kernel),
- `kw` the name of the extension,

– `typer` the type-checking function itself.
– `status`, a boolean flag indicating whether the extended annotation may have a validity status.

The last function has a different behavior and is used to load external ACSL modules with the `import` clause. It is treated at the end of the section (see p. 94).

During type-checking, the list `[e1;...;en]` will be given to `typer`, together with the current typing environment (which allows discriminating between contract and loop extensions and will have the appropriate logic labels set in the local environment). `typer` must return the corresponding `acsl_extension_kind` (possibly adding an entry for key `id` in an internal table if it chooses to return `Ext_id id`).

The first argument of `typer` is a `Logic_typing.typing_context` which provides lookup functions for the various kinds of identifiers that are present in the environment, as well as extensible type-checking functions for predicates, terms, and assigns clauses. Indeed, these functions take themselves as argument a `typing_context ctxt` and will use the functions of `ctxt` to type-check the children of the current node. Extensions can take advantage of this open recursion to recognize only subtrees of an otherwise normal ACSL predicate or term. For instance, the following code will let extension `foo` replace all occurrences of `\foo` by `42`.

```
open Logic_ptree
open Cil_types
open Logic_typing

let type_foo typing_context _loc l =
  let type_term ctxt env expr =
    match expr.lexpr_node with
    | PLvar "\\foo" → Logic_const.tinteger ~loc:expr.lexpr_loc 42
    | _ → typing_context.type_term ctxt env expr
  in
  let typing_context = { typing_context with type_term } in
  let res =
    List.map (typing_context.type_term typing_context (Lenv.empty())) l
  in
  Ext_terms res

let () =
  Acsl_extension.register_behavior ~plugin:"my_plugin" "foo" type_foo false
```

With this extension enabled, Frama-C will interpret the following clause in a given source file:

```
/*@ \my_plugin::foo 84 == \foo + \foo; */
```

as the following type-checked AST fragment:

```
/*@ \my_plugin::foo 84 == 42 + 42; */
```

If the extended clause is of kind `Ext_preds l` or `Ext_terms l`, and all the information of the extension is contained in the list `l`, no function other than the typing function needs to be registered. The parsing will use the standard way to parse untyped predicates and terms. After typing, the visitor will traverse each element of `l` as well as any predicate or term present in the AST. The pretty-printer will output these elements as a comma-separated list preceded by `kw` (or `loop kw` if the extension is a loop annotation).

However, depending on the situation, the following optional functions can be provided to the registration function in order to modify how ACSL extensions are handled by Frama-C:

– `preprocessor` a transformer to apply on the untyped term or predicate read during the parsing phase,
– `visitor` the visitor function to be applied when visiting the extension,
– `printer` the pretty-printing function associated to the extension,
– `short_printer` a function used to provide a brief textual representation of an extension.

The `preprocessor` function is applied just after parsing the extension terms. It takes the list of untyped terms or predicates and can either return the same list (but reading it to do some stuff) or return a new list. By default, this function is the identity.

The `visitor` function is used by the Frama-C visitors. It takes the current visitor, together with the `acsl_extension_kind` of the extended clause and must returns a `Cil.visitAction`. By default, this function just returns `Cil.DoChildren`.

The `printer` function is used by the `Cil_printer.pp_extended` function. It takes the current pretty-printer, the formatter, together with the `acsl_extension_kind` of the extended clause. By default, it prints the list of terms or predicates if the kind is `Ext_preds l` or `Ext_terms l`. If the kind is `Ext_id i`, it only prints the integer `i`.

The `short_printer` function is a function that can be useful for debugging or user-feedback. As an alternative to `Cil_printer.pp_extended`, the `Cil_printer.pp_short_extended` can be used to get brief description of the content of the extension. It is for example used by the GUI to get a more informative name for the extension in the file tree. By default, it does not print anything about the content of the extension, so that the result is `"kwd"` or `"loop kwd"`.

When the extension kind is `Ext_id`, it is common that the plugin defining the extension contains a table that associates some data to this identifier. In such a case, a `printer` might be needed to reconstruct the source code from the data so that a pretty printed code can be parsed again. For the same reason, an extension that registers a `preprocessor` that modifies the AST should probably register a `printer` to recover the original content.

It is also common, when the kind is `Ext_id`, to define a particular visitor for the extension, either to ignore the content of the extension as it is in an internal table of the plugin (thus returning a `SkipChildren` action) or, on the opposite, to give the possibility to a user defined visitor to get an access to this content.

The following code shows a more complete extension example. It provides the user a way to load some types (assumed to be external to Frama-C) so that they can be used in ACSL specification.

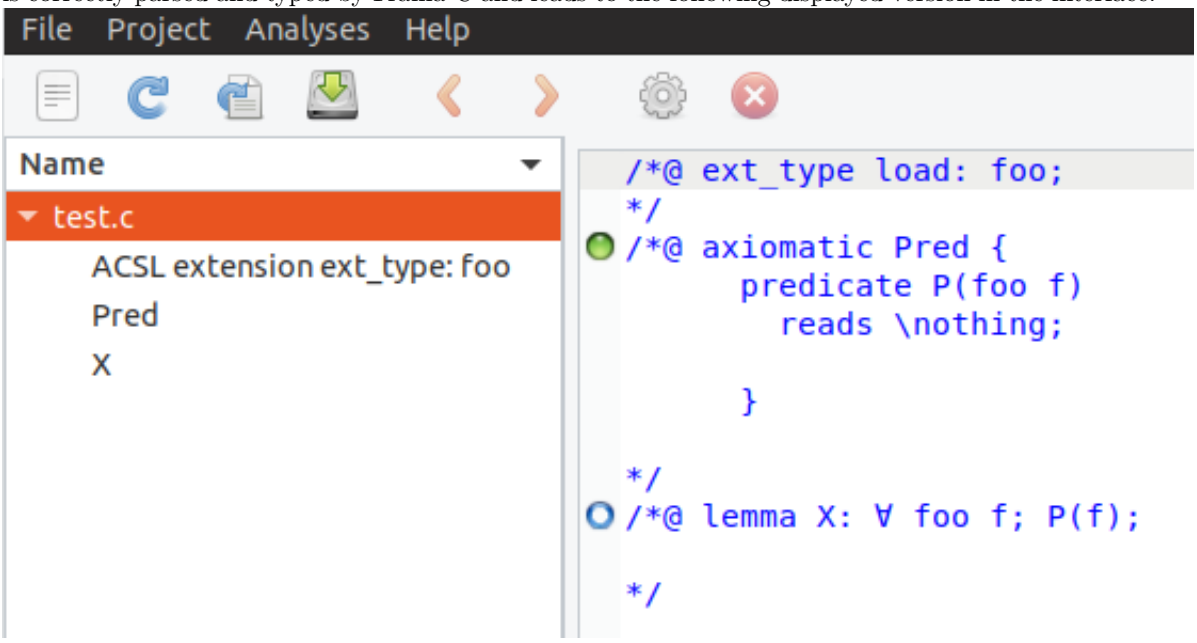
```
open Logic_ptree
open Logic_typing
open Cil_types

let preprocessor =
  List.map (fun e → begin match e with
     | { lexpr_node = PLnamed ("load", { lexpr_node = PLvar s; _ }) ; _ } →
       if not (Logic_env.is_logic_type s) then Logic_env.add_typename s
       else Kernel.error "Type already exists %s" s
     | _ → ()
   end ; e)

module Ts = struct
  let id = ref 0
  let types = Hashtbl.create 5

  let add t = let i = !id in Hashtbl.add types i t ; id ← i + 1 ; i
  let find = Hashtbl.find types
end

let typer ctxt loc = function
  | [ { lexpr_node = PLnamed ("load", { lexpr_node = PLvar s; _ }) ; _ } ] →
    let ti = { lt_name = s ; lt_params = [] ; lt_def = None ; lt_attr = []} in
    ctxt.add_logic_type s ti ;
    Ext_id (Ts.add ti)
  | _ →
    ctxt.error loc "Expected type loader"

let visitor _ _ = Cil.SkipChildren
```

```
let gen_printer s _pp fmt = function
  | Ext_id i →
    Format.fprintf fmt "%s: %s"
      (if s then "ext_type" else "load") (Ts.find i).lt_name
  | _ → assert false

let printer = gen_printer false
let short_printer = gen_printer true

let () =
  Acsl_extension.register_global ~plugin:"tloader"
    "ext_type" ~preprocessor typer ~visitor ~printer ~short_printer false
```

Namely, specification:

```
/*@ \tloader::ext_type load: foo ; */
/*@
  axiomatic Pred {
    predicate P(foo f) reads \nothing ;
  }
*/
/*@ lemma X: \forall foo f ; P(f) ; */
```

is correctly parsed and typed by Frama-C and leads to the following displayed version in the interface:



External Modules. Finally, let us introduce how to extend ACSL modules with external module importers. A typical usage is to import logical definitions from other languages, e.g. Coq or Why3, as regular ACSL module definitions. From the user point the view, this takes the form of an extended `import` clause:

```
//@ import <loader>: <module_name> [ \as <alias-name> ];
```

The syntax is similar to the general ACSL `import` annotation, except that the module name is prefixed with a loader corresponding to the name used when registering the module importer extension. For instance:

```
//@ import Foo: foo::bar::Jazz ;
```

Here, `Foo:` specifies the name of the ACSL extension responsible for importing the module, and `foo::bar::Jazz` is the name of the module to be imported. Like ACSL extensions, an extended syntax with a plugin name (required when registering the extension) can be used to avoid ambiguities if two or more plugins choose the same name for an importer extension:

```
  //@ import \myplugin::Foo: foo::bar::Jazz ;
```

To define such an external module importer, a plug-in shall call the `Acsl_extension.register_module_importer` with a suitable plugin name and importer function. The importer function is responsible for resolving the module name and defining the associated ACSL logic types, functions and predicates. Here is an example of such a loader function:

```
open Cil_types
open Logic_typing

let importer (ctxt: module_builder) (loc: location) (path: string list) =
 begin
   (* in the example above, path is ["foo","bar","Jazz"] *)
   (* ... read some file .. *)
   let p : location = ... in
   let t = Cil.make_logic_type "t" in
   (* ... configure type t ... *)
   let q : location = ... in
   let f = Cil.make_logic_info "f" in
   (* ... configure function f ... *)
   ctxt.add_logic_type p t ;
   ctxt.add_logic_function q f ;
   (* ... *)
 end

let () = Acsl_extension.register_module_importer ~plugin:"myplugin" "Foo" importer
```

Added types and functions shall use local names, as in the example above. After importation, the imported declarations will be accessible through their full-name identifier, for instance `foo::bar::Jazz:f` in the example above.

As explained in the Frama-C source documentation, module importers might be invoked multiple times by the kernel, e.g. when a given module is imported from different source files. Although the importer function itself can use memoization techniques to save time, the module builder shall be populated on every call.

When printing the internal AST from Frama-C command line using `-print` option, externally imported modules are listed with one single clause for each, with no aliasing in order to avoid any ambiguity. For instance:

```
//@ import \myplugin::Foo: foo::bar::Jazz \as _ ;
```

Alternatively, you can debug the logical definitions actually imported by any driver by using `-print` with the `printer:imported-modules` debugging key:

```
> frama-c ... -print -kernel-message-key printer:imported-modules
```

With this option, the contents of the imported modules are printed like regular module definitions, with only a comment to mention the origin of the plug-in:

```
/*@ // import \myplugin::Foo:
    module foo::bar::Jazz {
        ...
```

```
    }
  */
```

Notice that, when using the `printer:imported-modules` message key, the resulting file will still compile and type-check, but the plug-in extension will no more be aware of the external nature of those modules, and it will probably *not* work as with the original specification.

4.17 Locations

Prerequisite: *None.*

In Frama-C, different representations of C locations exist. Section 4.17.1 presents them. Moreover, maps indexed by locations are also provided. Section 4.17.2 introduces them.

4.17.1 Representations

There are four different representations of C locations. Actually only three are really relevant. All of them are defined in module `Locations`. They are introduced below. See the documentation of `src/kernel_services/abstract_interp/locations.mli` for details about the provided operations on these types.

- Type `Location_Bytes.t` is used to represent values of C expressions like `2` or `((int) &a) + 13`. With this representation, there is no way to know the size of a value while it is still possible to join two values. Roughly speaking it is represented by a mapping between C variables and offsets in bytes.
- Type `location`, equivalently `Location.t` is used to represent the right part of a C affectation (including bitfields). It is represented by a `Location_Bits.t` (see below) attached to a size. It is possible to join two locations *if and only if they have the same sizes.*
- Type `Location_Bits.t` is similar to `Location_Bytes.t` with offsets in bits instead of bytes. Actually it should only be used inside a location.
- Type `Zone.t` is a set of bits (without any specific order). It is possible to join two zones *even if they have different sizes.*

Recommendation 4.2 *Roughly speaking, locations and zones have the same purpose. You should use locations as soon as you have no need to join locations of different sizes. If you require to convert locations to zones, use the function* `Locations.enumerate_valid_bits`.

As join operators are provided for these types, they can be easily used in abstract interpretation analyses (which can themselves be implemented thanks to one of functors of module `Dataflow2`.

4.17.2 Map Indexed by Locations

Modules `Lmap` and `Lmap_bitwise` provide functors implementing maps indexed by locations and zones (respectively). The argument of these functors have to implement values attached to indices (resp. locations or zones).

These implementations are quite more complex than simple maps because they automatically handle overlaps of locations (or zones). So such implementations actually require that the structures implementing the values attached to indices are at least semi-lattices; see the corresponding signatures in module `Lattice_type`. For this purpose, functors of the abstract interpretation toolbox can help (see in particular module `Abstract_interp`).

4.18 GUI Extension

Prerequisite: *Knowledge of* Lablgtk3.

Each plug-in can extend the Frama-C graphical user interface (aka *GUI*) in order to support its own functionalities in the Frama-C viewer. For this purpose, a plug-in developer has to register a function of type `Design.main_window_extension_points → unit` thanks to `Design.register_extension`. The input value of type `Design.main_window_extension_points` is an object corresponding to the main window of the Frama-C GUI. It provides accesses to the main widgets of the Frama-C GUI and to several plug-in extension points. The documentation of the class type `Design.main_window_extension_points` is accessible through the source documentation (see Section 2.3.8).

Besides time-consuming computations have to call the function `Async.yield` from time to time in order to keep the GUI reactive.

The GUI implementation uses Lablgtk3 [10]: you can use any Lablgtk3-compatible code in your gui extension. A complete example of a GUI extension may be found in the plug-in `Occurrence` (see file `src/plugins/occurrence/gui/register_gui.ml`).

Potential issues *All the GUI plug-in extensions share the same window and same widgets. So conflicts can occur, especially if you specify some attributes on a predefined object. For example, if a plug-in wants to highlight a statement s in yellow and another one wants to highlight s in red at the same time, the behavior is not specified but it could be quite difficult to understand for an user.*

4.19 Packaging

If you intend to release your plug-in, a possible way is to take advantage of the `opam` integration within `dune`[15]. Basically, the `dune-project` file (see Section 2.3) should contain a stanza `(generate_opam_files true)`, as well as some meta-information (location of the sources, licence, author(s), etc.). It is also possible to provide this information in a file `my-plugin-package.opam.template`, assuming `my-plugin-package` is the name of the package of the plug-in in the `dune-project` file. See the dune documentation for detailed information about the creation of the `opam` file.

4.20 Profiling with Landmarks

Landmarks[16] is a library for "quick and dirty" profiling of OCaml programs. It allows the insertion of annotations in the code to enable profiling of specific parts of it, but also an automatic mode, in which every function call is instrumented. This can help quickly obtain some profiling data for a plug-in or a specific test case.

For quick usage of the library:

– make sure that you have packages `landmarks` and `landmarks-ppx` installed;

[15] https://dune.readthedocs.io/en/stable/opam.html

[16] https://github.com/LexiFi/landmarks

- run `make DUNE_WS=bench` to compile Frama-C using `dev/dune-workspace.bench`, which sets up a specific *Dune context* to enable instrumentation without affecting the default context. This compiles each Frama-C file in a different directory inside `_build`.
- install the instrumented Frama-C: `make install DUNE_WS=bench`.
- enable instrumentation *during execution* of Frama-C, using the `OCAML_LANDMARKS` environment variable:

  ```
  OCAML_LANDMARKS=<options> frama-c [files] [options]
  ```

  The easiest setup is `OCAML_LANDMARKS=auto`, which will instrument everything and output timing information to `stderr` after program termination. You can also use `OCAML_LANDMARKS=auto,output=landmarks.log` to output the timing information to file `landmarks.log`.
- you can also run the instrumented Frama-C without installing it, via `dune exec`:

  ```
  OCAML_LANDMARKS=auto dune exec --workspace dev/dune-workspace.bench \
    --context bench -- frama-c [files] [options]
  ```

  The `--workspace dev/dune-workspace.bench` argument tells Dune to use the workspace in which Landmarks is configured.
  The `--context bench` argument tells Dune to use the instrumented context. This context is different from the default one in order to avoid overwriting non-instrumented files when running `make` without `DUNE_WS`.

To instrument a single file: add `[@@@landmark "auto"]` at the beginning of the file.
To instrument a single function: add `[@landmark]` after the `let`, e.g.:

```
let[@landmark] add_visitor vis =
```

Check https://github.com/LexiFi/landmarks for its documentation.

4.21 Profiling a custom plug-in

To profile your own plug-in using Landmarks, do the following:

1. Add `(instrumentation (backend landmarks))` to your plug-in's `dune` file, inside the `library` stanza, as in the example below:

   ```
   (library
     (name my-plug-in)
     (public_name frama-c-my-plug-in.core)
     ...
     (libraries frama-c.kernel)
     (instrumentation (backend landmarks)) ; <<< ADD THIS LINE
   )
   ```

2. Create a `dune-workspace.bench` file in your plugin's top-level directory:

   File **dune-workspace.bench**

   ```
   (lang dune 3.13)
   (context
    (default
     (name bench)
     (profile bench)
     (instrument_with landmarks)
     (env
   ```

```
      (_ (env-vars ("OCAML_LANDMARKS" "auto")))))
 )
```

3. Add `--workspace=dune-workspace.bench` to the `dune build` and `dune install` commands that you run, e.g.:

   ```
   dune build --workspace=dune-workspace.bench @install
   dune install --workspace=dune-workspace.bench
   ```

   This will compile and install the plug-in with Landmarks instrumentation enabled. Then you just need to set `OCAML_LANDMARKS` as described in the previous section, e.g.:

   ```
   OCAML_LANDMARKS=auto frama-c [my-plugin-options]
   ```

4. For `dune exec`, you need to add the `--workspace` option as well as `--context bench`. Combined with `OCAML_LANDMARKS`, the command-line will resemble the following:

   ```
   OCAML_LANDMARKS=auto dune exec --workspace dev/dune-workspace.bench \
     --context bench -- frama-c [files] [options]
   ```

Changes A

This chapter summarizes the major changes in this documentation between each Frama-C release, from newest to oldest.

30.0 (Zinc)

– **ACSL Extension**: Document new `register_module_importer` extension.

29.0 (Copper)

– **Logging Services**: Document new `Current_loc` module
– There is no more `Db` module:
  – Whole document: `Db.Main.extend` is now `Boot.Main.extend`
  – **Declaring dependencies**: removed section about `Db`
  – **Kernel-integrated Registration and Access**: removed section
  – **Registering a New State**: adapted example on postponed dependencies
  – **GUI Extension**: `Db.yield` is now `Async.yield`

28.0 (Nickel)

– **Logical Annotations**: Add sections about specification generation.

27.0 (Cobalt)

– **Customizing the machine model**: Rewrite section according to new machdep management mechanism

26.0 Iron

– **Makefiles/Dune**: Document the use of `dune` for compiling and testing plug-in, and describe transition from a `Makefile`-based to a `dune`-based setup.

– **Journalisation**: Journalisation has been removed.

25.0 Manganese

– **Testing**: Document new directives (`PLUGIN`, `SCRIPT` and `LIBS`) and new predefined macros for `ptests`.

22.0 Titanium

– **Testing**: Document new directives `TIMEOUT` and `NOFRAMAC`

21.0 Scandium

– **Configure**: Documentation of `configure_pkg`, `plugin_require_pkg` and `plugin_use_pkg` macros.

20.0 Calcium

– **Testing**: Documentation of the new directive `MODULE`.

19.0 Potassium

– **ACSL Extension**: Document new `status` flag for registration functions
– **Testing**: Document usage of `@@` in a directive
– **Profiling with Landmarks**: New section

18.0 Argon

– **Logging Services**: Document `error` and `failure` behaviors.
– **ACSL Extensions**: New extension categories, for global and plain code annotations

Chlorine-20180501

– **Logging Services**: Introduction of warning categories

Sulfur-20171101

- **Tutorial**: Update and complete the Hello plug-in section along with making it available online.
- **Testing**: Explain the appropriate way to handle compilation of `.ml` scripts during tests
- **Makefiles**: Remove references to obsolete `Makefile.plugin` file

Phosphorus-20170501

- **Makefiles**: Update overview of Makefiles.
- **ACSL Extensions**: Update documentation after refactoring of ACSL extensions.
- **Machine model**: fully new section.

Silicon-20161101

- **ACSL Extensions**: Updated documentation for newly introduced loop extensions.

Aluminium-20160501

- **Tutorial**: Plugin `Cfg` renamed to `ViewCfg`; minor fixes.
- **Ptests**: Documentation of the new directive `EXEC`.
- **Ptests**: Documentation for sharing directives amongst ptests configurations
- **Makefiles**: Documentation for `install::` target in dynamic plugins
- **Makefiles**: Documentation of exported `TARGET_*` variables
- **Makefiles**: Documentation of new option `PLUGIN_EXTRA_DIRS`
- **Ptests**: New option `-gui`

Magnesium-20151001

- **License Policy**: remove this section.
- **Ptests**: New configuration directive `LOG` and new macro `PTEST_RESULT`
- **File Tree**: remove this section, now subsumed by the new Chapter on Software Architecture and by the API documentation.
- **File Tree Overview**: remove this useless section.
- **Software Architecture**: rewrite the whole chapter.
- No more `PLUGIN_HAS_MLI`.

Sodium-20150201

- **Type Library**: document `Datatype.Serializable_undefined`.
- **Command Line Options**: document `Parameter_sig.Kernel_function_set`.
- **Configure.in**: warn about using Frama-C macros within conditionals
- **Logical Annotations**: document ACSL extended clauses mechanism (added section 4.16).

– **Tutorial**: fix `hello_world.ml`.

Neon-20140301

– **Reference Manual**: update list of main kernel modules.
– **Logical Annotations**: document module `Property`.
– **Command Line Options**: update according to kernel changes that split the module `Plugin` into several modules.
– **Architecture**, **Plug-in Registration and Access** and **Reference Manual**: document registration of a plug-in through a `.mli` file.
– **Makefiles**: introducing `Makefile.generic`.
– **Testing**: `MACRO` configuration directive.

Fluorine-20130601

– **Tutorial**: fully rewritten.
– **Architecture** and **Reference Manual**: remove references to `Cilutil` module.

Oxygen-20121001

– **Makefile** `WARN_ERROR_ALL` variable.
– **Log**: Debug category (`~dkey` argument).
– **Visitor**: `DoChildrenPost` action.
– **Testing**: document the need for directories to store result and oracles.
– **Project Management System**: Fine tuning of AST dependencies.
– **Testing**: added `PTESTS_OPTS` and `PLUGIN_PTESTS_OPTS` Makefile's variables.
– **Type**: document the `type` library.
– **Logical Annotations**: fully updated.
– **Reference Manual**: update kernel files.
– **Testing**: merge parts in *Advanced Plug-in Development* and in *Reference Manual*.
– **Website**: refer to CEA internal documentation.
– **Command Line Options**: explain how to modify the default behavior of an option.
– **Command Line Options**: fully updated.
– **Project Management System**: fully updated.
– **Plug-in Registration and Access**: `Type` replaced by `Datatype` and document labeled argument `journalize`.
– **Configure.in**: updated.
– **Plug-in General Services**: updated.
– **Software Architecture**: `Type` is now a library, not just a single module.

Nitrogen-20111001

– **Tutorial of the Future**: new chapter for preparing a future tutorial.
– **Types as first class values**: links to articles.
– **Tutorial**: kernel-integrated plug-ins are now deprecated.

– **Visitors**: example is now out-of-date.

Carbon-20110201

Unchanged.

Carbon-20101201-beta1

– **Visitors**: update example to new kernel API.
– **Documentation**: external plugin API documentation.
– **Visitors**: fix bug (replace `DoChildrenPost` by `ChangeDoChildrenPost`), change semantics wrt `vstmt_aux`.

Carbon-20101201-beta1

– **Very Important Preliminary Warning**: adding this very important chapter.
– **Tutorial**: fix bug in the 'Hello World' example.
– **Testing**: updated semantics of `CMD` and `STDOPT` directives.
– **Initialization Steps**: updated according to new options `-then` and `-then-on` and to the new 'Files Setting' stage.
– **Visitors**: example updated

We list changes of previous releases below.

Boron-20100401

– **Configure.in**: updated
– **Tutorial**: the section about kernel-integrated plug-in is out-of-date
– **Project**: no more `rehash` in datatypes
– **Initialisation Steps**: fixed according to the current implementation
– **Plug-in Registration and Access**: updated according to API changes
– **Documentation**: updated and improved
– **Introduction**: is aware of the Frama-C user manual
– **Logical Annotations**: fully new section
– **Tutorial**: fix an efficiency issue with the Makefile of the `Hello` plug-in

Beryllium-20090902

– **Makefiles**: update according to the new `Makefile.kernel`

Beryllium-20090901

- **Makefiles**: update according to the new makefiles hierarchy
- **Writing messages**: fully documented
- **Initialization Steps**: the different stages are more precisely defined. The implementation has been modified to take into account specificities of dynamically linked plug-ins
- **Project Management System**: mention value `descr` in Datatype
- **Makefile.plugin**: add documentation for additional parameters

Beryllium-20090601-beta1

- **Initialization Steps**: update according to the new implementation
- **Command Line Options**: update according to the new implementation
- **Plug-in General Services**: fully new section introducing the new module `Plugin`
- **File Tree**: update according to changes in the kernel
- **Makefiles**: update according to the new file `Makefile.dynamic` and the new file `Makefile.config.in`
- **Architecture**: update according to the recent implementation changes
- **Tutorial**: update according to API changes and the new way of writing plug-ins
- **configure.in**: update according to changes in the way of adding a simple plug-in
- **Plug-in Registration and Access**: update according to the new API of module `Type`

Lithium-20081201

- **Changes**: fully new appendix
- **Command Line Options**: new sub-section *Storing New Dynamic Option Values*
- **Configure.in**: compliant with new implementations of `configure_library` and `configure_tool`
- **Exporting Datatypes**: now embedded in new section *Plug-in Registration and Access*
- **GUI**: update, in particular the full example has been removed
- **Introduction**: improved
- **Plug-in Registration and Access**: fully new section
- **Project**: compliant with the new interface
- **Reference Manual**: integration of dynamic plug-ins
- **Software architecture**: integration of dynamic plug-ins
- **Tutorial**: improve part about dynamic plug-ins
- **Tutorial**: use `Db.Main.extend` to register an entry point of a plug-in.
- **Website**: better highlighting of the directory containing the html pages

Lithium-20081002+beta1

- **GUI**: fully updated
- **Testing**: new sub-section *Alternative testing*
- **Testing**: new directive `STDOPT`
- **Tutorial**: new section *Dynamic plug-ins*
- **Visitor**: `ChangeToPost` in sub-section *Action Performed*

Helium-20080701

- **GUI**: fully updated
- **Makefile**: additional variables of `Makefile.plugin`
- **Project**: new important note about registration of internal states in Sub-section *Internal State: Principle*
- **Testing**: more precise documentation in the reference manual

Hydrogen-20080502

- **Documentation**: new sub-section *Website*
- **Documentation**: new ocamldoc tag *@plugin developer guide*
- **Index**: fully new
- **Project**: new sub-section *Internal State: Principle*
- **Reference manual**: largely extended
- **Software architecture**: fully new chapter

Hydrogen-20080501

- **First public release**

Bibliography

[1] Patrick Baudin, Jean-Christophe Filliâtre, Thierry Hubert, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI C Specification Language. Version 1.8*, March 2014.

[2] Patrick Baudin and Anne Pacalet. Slicing plug-in. http://frama-c.com/slicing.html.

[3] Loïc Correnson, Pascal Cuoq, Florent Kirchner, Armand Puccetti, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. *Frama-C User Manual*, February 2015. http://frama-c.com/download/frama-c-user-manual.pdf.

[4] Loïc Correnson, Zaynah Dargaye, and Anne Pacalet. *Frama-C's WP plug-in*, February 2015. http://frama-c.com/download/frama-c-wp-manual.pdf.

[5] Loïc Correnson and Julien Signoles. Combining Analysis for C Program Verification. In *Formal Methods for Industrial Critical Systems (FMICS)*, August 2012.

[6] Pascal Cuoq, Damien Doligez, and Julien Signoles. Lightweight Typed Customizable Unmarshaling. *ML Workshop'11*, September 2011.

[7] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C, A Program Analysis Perspective. In *the 10th International Conference on Software Engineering and Formal Methods (SEFM 2012)*, volume 7504 of *LNCS*, pages 233–247. Springer, 2012.

[8] Pascal Cuoq and Julien Signoles. Experience Report: OCaml for an industrial-strength static analysis framework. In *Proceedings of International Conference of Functional Programming (ICFP'09)*, pages 281–286, New York, NY, USA, September 2009. ACM Press.

[9] Pascal Cuoq, Boris Yakobowski, and Virgile Prevosto. *Frama-C's value analysis plug-in*, February 2015. http://frama-c.com/download/frama-c-eva-manual.pdf.

[10] Jacques Garrigue, Benjamin Monate, Olivier Andrieu, and Jun Furuse. LablGTK2. http://lablgtk.forge.ocamlcore.org.

[11] Philippe Hermann and Julien Signoles. *Frama-C's RTE plug-in*, April 2013. http://frama-c.com/download/frama-c-rte-manual.pdf.

[12] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, pages 1–37, 2015. Extended version of [7].

[13] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system*. http://caml.inria.fr/pub/docs/manual-ocaml/index.html.

[14] Donald Michie. Memo functions: a language feature with "rote-learning" properties. Research Memorandum MIP-R-29, Department of Machine Intelligence & Perception, Edinburgh, 1967.

[15] Donald Michie. Memo functions and machine learning. *Nature*, 218:19–22, 1968.

[16] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag.

[17] Julien Signoles. Foncteurs impératifs et composés: la notion de projet dans Frama-C. In Hermann, editor, *JFLA 09, Actes des vingtièmes Journées Francophones des Langages Applicatifs*, volume 7.2 of *Studia Informatica Universalis*, pages 245–280, 2009. In French.

[18] Julien Signoles. Une bibliothèque de typage dynamique en OCaml. In Hermann, editor, *JFLA 11, Actes des vingt-deuxièmes Journées Francophones des Langages Applicatifs*, Studia Informatica Universalis, pages 209–242, January 2011. In French.

[19] Nicolas Stouls and Virgile Prevosto. *Frama-C's Aoraï plug-in*, April 2013. http://frama-c.com/download/frama-c-aorai-manual.pdf.

List of Figures

Index