Plug-in Development Guide

Release 20.0 (Calcium)

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

This work is licensed under a Creative Commons “Attribution-ShareAlike 4.0 International” license.

CEA LIST, Software Security Laboratory, Saclay.F-91191

©2009-2019 CEA LIST
## Contents

**Foreword** .............................. 9

**1 Introduction** .......................... 11
  1.1 About this document .................. 11
  1.2 Outline ................................ 12

**2 Tutorial** ............................ 13
  2.1 Quick Setup of a Development Environment ........... 13
     2.1.1 Emacs Configuration Files ............... 13
     2.1.2 Quick Merlin guide ...................... 14
  2.2 What Does a Plug-in Look Like? ................. 14
  2.3 The Hello plug-in ..................... 15
     2.3.1 A Simple Script ....................... 15
     2.3.2 Registering a Script as a Plug-in ......... 16
     2.3.3 Displaying Messages ..................... 17
     2.3.4 Adding Command Line Options ............. 18
     2.3.5 Writing a Makefile ...................... 19
     2.3.6 Testing your Plug-in ................. 22
     2.3.7 Documenting your Source Code ........... 24
  2.4 The ViewCfg plug-in .................. 26
     2.4.1 Visiting the AST ....................... 26
     2.4.2 Plug-In registration and command-line options .... 30
     2.4.3 Interfacing with a kernel-integrated plug-in ...... 31
     2.4.4 Extending the Frama-C GUI .............. 31
     2.4.5 Splitting files and writing a Makefile ........ 33
     2.4.6 Getting your Plug-in Usable by Others ..... 36
     2.4.7 Writing your Plug-in into the Journal ........ 36
     2.4.8 Writing a Configure Script ............ 37
     2.4.9 Getting your plug-in Usable in a Multi Projects Setting .... 37
## 3 Software Architecture

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 General Description</td>
<td>41</td>
</tr>
<tr>
<td>3.2 Plug-ins</td>
<td>43</td>
</tr>
<tr>
<td>3.3 Libraries</td>
<td>43</td>
</tr>
<tr>
<td>3.4 Kernel Services</td>
<td>43</td>
</tr>
<tr>
<td>3.5 Kernel Internals</td>
<td>44</td>
</tr>
</tbody>
</table>

## 4 Advanced Plug-in Development

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Frama-C Configure.in</td>
<td>45</td>
</tr>
<tr>
<td>4.1.1 Principle</td>
<td>46</td>
</tr>
<tr>
<td>4.1.2 Addition of a Simple Plug-in</td>
<td>46</td>
</tr>
<tr>
<td>4.1.3 Configuration of New Libraries or Tools</td>
<td>47</td>
</tr>
<tr>
<td>4.1.4 Addition of Library/Tool Dependencies</td>
<td>48</td>
</tr>
<tr>
<td>4.1.5 Addition of Plug-in Dependencies</td>
<td>49</td>
</tr>
<tr>
<td>4.2 Plug-in Specific Configure.ac</td>
<td>49</td>
</tr>
<tr>
<td>4.3 Frama-C Makefile</td>
<td>50</td>
</tr>
<tr>
<td>4.4 Plug-in Specific Makefile</td>
<td>50</td>
</tr>
<tr>
<td>4.4.1 Using Makefile.dynamic</td>
<td>50</td>
</tr>
<tr>
<td>4.4.2 Compiling Frama-C and external plug-ins at the same time</td>
<td>51</td>
</tr>
<tr>
<td>4.5 Testing</td>
<td>52</td>
</tr>
<tr>
<td>4.5.1 Using ptests</td>
<td>52</td>
</tr>
<tr>
<td>4.5.2 Configuration</td>
<td>54</td>
</tr>
<tr>
<td>4.5.3 Alternative Testing</td>
<td>55</td>
</tr>
<tr>
<td>4.5.4 Detailed options</td>
<td>56</td>
</tr>
<tr>
<td>4.5.5 Detailed directives</td>
<td>57</td>
</tr>
<tr>
<td>4.6 Plug-in General Services</td>
<td>60</td>
</tr>
<tr>
<td>4.7 Logging Services</td>
<td>60</td>
</tr>
<tr>
<td>4.7.1 From printf to Log</td>
<td>61</td>
</tr>
<tr>
<td>4.7.2 Log Quick Reference</td>
<td>62</td>
</tr>
<tr>
<td>4.7.3 Logging Routine Options</td>
<td>63</td>
</tr>
<tr>
<td>4.7.4 Advanced Logging Services</td>
<td>64</td>
</tr>
<tr>
<td>4.8 The Datatype library: Type Values and Datatypes</td>
<td>67</td>
</tr>
<tr>
<td>4.8.1 Type Value</td>
<td>67</td>
</tr>
<tr>
<td>4.8.2 Datatype</td>
<td>67</td>
</tr>
<tr>
<td>4.9 Plug-in Registration and Access</td>
<td>71</td>
</tr>
<tr>
<td>4.9.1 Registration through a .mli File</td>
<td>71</td>
</tr>
<tr>
<td>4.9.2 Kernel-integrated Registration and Access</td>
<td>72</td>
</tr>
<tr>
<td>4.9.3 Dynamic Registration and Access</td>
<td>73</td>
</tr>
</tbody>
</table>
4.10 Journalization ............................................................... 76
4.11 Project Management System ........................................... 76
   4.11.1 Overview and Key Notions ........................................ 77
   4.11.2 State: Principle .................................................. 77
   4.11.3 Registering a New State .......................................... 79
   4.11.4 Direct Use of Low-level Functor State_builder.Register ....... 81
   4.11.5 Using Projects .................................................... 83
   4.11.6 Selections .......................................................... 84
4.12 Command Line Options .................................................. 85
   4.12.1 Definition .......................................................... 85
   4.12.2 Tuning ............................................................. 87
4.13 Initialization Steps ....................................................... 87
4.14 Customizing the AST creation ........................................... 90
4.15 Customizing the machine model ....................................... 91
4.16 Machdep record fields .................................................. 97
4.17 Visitors ................................................................. 99
   4.17.1 Entry Points ....................................................... 99
   4.17.2 Methods ........................................................... 99
   4.17.3 Action Performed ................................................ 100
   4.17.4 Visitors and Projects ............................................. 100
   4.17.5 In-place and Copy Visitors ..................................... 100
   4.17.6 Differences Between the Cil and Frama-C Visitors .......... 101
   4.17.7 Example .......................................................... 102
4.18 Logical Annotations ..................................................... 103
4.19 Extending ACSL annotations ............................................ 103
4.20 Locations .................................................................. 106
   4.20.1 Representations ................................................... 106
   4.20.2 Map Indexed by Locations ....................................... 107
4.21 GUI Extension ............................................................ 107
4.22 Documentation ........................................................... 108
   4.22.1 General Overview ................................................ 108
   4.22.2 Source Documentation .......................................... 108
5 Reference Manual ............................................................ 111
   5.1 Configure.in ........................................................... 111
   5.2 Makefiles ................................................................ 112
      5.2.1 Overview ......................................................... 112
Foreword

This is the documentation of the Frama-C implementation\(^1\) which aims at helping developers integrate new plug-ins inside this platform. It started as a deliverable of the task 2.3 of the ANR RNTL project CAT\(^2\).

The content of this document corresponds to the version 20.0 (Calcium), released on December 4, 2019, 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: Gergö Barany, Patrick Baudin, Richard Bonichon, Pascal Cuq, Zaynah Dargaye, Florent Garnier, Pierre-Loïc Garoche, Philippe Herrmann, Boris Hollas, Nikolaï Kosmatov, Jean-Christophe Léchenet, André Maroneze, Benjamin Monate, Yannick Moy, Anne Pacalet, Armand Puccetti, Muriel Roger and Boris Yakobowski. We also thank Johannes Kanig for his Mlpost support\(^3\), the tool formerly used for making figures of this document.

\(^1\)http://frama-c.com
\(^2\)http://www.rntl.org/projet/resume2005/cat.htm
\(^3\)http://mlpost.lri.fr
Chapter 1

Introduction

Frama-C (Framework for Modular Analyses of C) is a software platform which helps the development of static analysis tools for C programs thanks to a plug-ins 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, a set of Frama-C-specific programming rules 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 API is documented in the html source code generated by make doc (see Section 4.22.1 for additional details about this documentation).

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 his/her 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. In the Frama-C source code, the ocamlproc documentation of such a value contains the special tag @plugin development guide while, in the html documentation of the Frama-C API, the note “Consult the Plugin Development Guide for additional details” is attached the value name.

The most important paragraphs are displayed inside gray 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 four 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.

Chapter 5 is a reference manual with complete documentation for some particular points of the Frama-C platform.
Chapter 2
Tutorial

Target readers: beginners.

This chapter aims at helping a developer to write his 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.

First, Section 2.1 describes how to quickly setup 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 Quick Setup of a Development Environment

This setup is based on Emacs as the IDE, with OPAM for OCaml package installation and Merlin for OCaml source code navigation. Similar environments can be setup using other editors, such as Vim and Sublime Text, but they are not detailed here; please refer to the Merlin documentation for further details on how to setup other editors for OCaml development.

2.1.1 Emacs Configuration Files

The Frama-C source distribution includes, in its share directory, some frama-c-*.el files (Emacs-lisp scripts) with default settings for an OCaml development environment based on Emacs + OPAM + Merlin. To set it up, do the following:

1. Install Emacs, OPAM, and OPAM packages merlin, tuareg and ocp-indent;
2. Copy share/emacs/frama-c-*.el to a directory in your Emacs load-path;
3. Add (load-library "frama-c-recommended") to your .emacs init file.

For instance, run these commands (after having installed OPAM):
opam install merlin tuareg ocp-indent
mkdir ~/.emacs.d/frama-c
cp <path to frama-c sources>/share/emacs/frama-c-*.el ~/.emacs.d/frama-c/

And then add this in the beginning of your ~/.emacs:

(add-to-list 'load-path "~/.emacs.d/frama-c")
(load-library "frama-c-recommended")

You can replace frama-c-recommended with frama-c-dev in the above line. The former contains some extra, optional settings, while the latter contains only the most important settings for OCaml development. We recommend you review the definitions in frama-c-recommended and remove those you find unnecessary.

2.1.2 Quick Merlin guide

Merlin\(^1\) is a tool for OCaml type information, source code navigation and auto-completion. Detailed instructions for using Merlin with Emacs are available at https://github.com/the-lambda-church/merlin/wiki/emacs-from-scratch.

The Frama-C Makefile contains a target merlin that generates an appropriate .merlin file to be used when editing Frama-C .ml/.mli files. Run make merlin after compiling Frama-C, and the generated .merlin file will be automatically used when opening Frama-C OCaml sources.

Here is a quick summary of the most useful commands when editing .ml/.mli files:

- **Ctrl+c Ctrl+t**: display type information (repeat it to further expand types)
- **Ctrl+c Ctrl+l**: jump to definition (for variables, types, modules, etc.)
- **Ctrl+c Ctrl+x**: jump to next error in current buffer

Merlin also includes an auto-complete feature. Check its website for further documentation.

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:

- extend the Frama-C entry point thanks to the function Db.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.

\(^1\)Merlin is available on Github (https://github.com/the-lambda-church/merlin) and as an OPAM package.
2.3 The Hello plug-in

This simple plug-in explain how to make your plug-in interact basically with several aspects of the Frama-C framework: registration, getting command-line options, compilation and installation, 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 A Simple Script

The easiest way to extend Frama-C is to write a simple script. A simple 'Hello World' script consists of a single OCaml file:

```
let run () =
  try
```

Figure 2.1: Plug-in Integration Overview.
CHAPTER 2. TUTORIAL

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

The script is compiled, loaded and run with the command `frama-c -load-script hello_world.ml`. Executing this command creates a `hello.out` file in the current directory.

### 2.3.2 Registering a Script as a Plug-in

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

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

```ocaml
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 "%s\n" msg
  let () = Db.Main.extend run
```

The argument for this 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, usually used as a prefix for plug-in options. No space is allowed in that string.
2.3. THE HELLO PLUG-IN

- help is a string containing free-form text, containing a description of the module.

Visible results of the registration include:

- “hello world” appears in the list of available plug-ins (consultable with frama-c -load-script hello_world.ml -plugins);
- default options for the plug-in work, including the inline help (available with frama-c -load-script hello_world.ml -hello-help).

2.3.3 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.

```
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
```

Running this script yields the following output:

```
$ frama-c -load-script hello_world.ml
[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 takes the same arguments than the OCaml function Format.printf.

The function feedback outputs messages that show progress to the user. In this example, we gave to feedback a log level of 2, because we estimated that in most case 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:

3However writing to a new file using standard OCaml primitives is OK.
2.3.4 Adding Command Line Options

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

If the plug-in is installed (with make install), it will be loaded and executed on every invocation of frama-c, which is surely not what you want. To avoid 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 the option is to take the short name of the module with no 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.

```ml
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;
```

For a comprehensive list of the logging routines and options, see Section 4.7.
2.3. THE HELLO PLUG-IN

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 () = Db.Main.extend run

Registering these new options is done by calling the \texttt{Self.False} and \texttt{Self.String} functors, which respectively creates 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 \textit{via} \texttt{Enabled.get()} and \texttt{Output_file.get()}. With this change, the hello message is displayed only if \texttt{Frama-C} is executed with the \texttt{-hello} option.

\begin{verbatim}
$ frama-c
$ frama-c -load-script hello_world.ml -hello
Hello, world!
$ frama-c -load-script hello_world.ml -hello -hello-output hello.out
$ ls hello.out
hello.out
\end{verbatim}

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

\begin{verbatim}$ frama-c -load-script hello_world.ml -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)

\end{verbatim}

2.3.5 Writing a Makefile

The use of \texttt{load-script} is ideal for small experimentations, or when writing very specific extensions. When a plug-in becomes larger, or more general-purpose, and must be split into several files, it is a good idea to build and install it properly. \texttt{Frama-C} provides means to simplify this through the use of \texttt{Makefiles}.

First, let us create a \texttt{hello} directory that will contain all of our plug-in files. We put \texttt{hello_world.ml} inside it, and then create our \texttt{Makefile} there.

\textbf{A simple Makefile}

We write a simple \texttt{./Makefile} for our \texttt{./hello_world.ml} plug-in:

\begin{verbatim}
File ./Makefile

FRAMAC_SHARE := $(shell frama-c-config -print-share-path)
FRAMAC_LIBDIR := $(shell frama-c-config -print-libpath)
PLUGIN_NAME = Hello
\end{verbatim}
CHAPTER 2. TUTORIAL

PLUGIN_CMO = hello_world
include $(FRAMAC_SHARE)/Makefile.dynamic

This Makefile sets some variables before including the generic Makefile.dynamic which is installed within Frama-C. It may be customized in several ways to help building a plug-in (see Section 4.4 for details).

The name of each compilation unit (here hello_world) must be different from the plug-in name set by the Makefile (here Hello), from any other plug-in names (e.g. value) and from any other Frama-C kernel OCaml files (e.g. plug-in).

The plug-in also needs an interface file. Indeed, thanks to Makefile.dynamic, each plug-in is packed into a single module $(PLUGIN_NAME) (here Hello) which needs an interface. Here we simply export an empty interface in order to hide the whole implementation to other developers.

File ./Hello.mli

(* * Hello World plug-in.
   No function is exported. *)

Note the unusual capitalization of the filename Hello.mli which is required for compilation purposes.

Inside the plug-in’s directory, run make to compile it. Note that the compiled files are copied into a top (for top-level) subdirectory (if our plug-in had GUI-dependent modules, they would be placed in a gui subdirectory). The module can then be loaded and executed by using frama-c -load-module top/Hello.

Then run make install to install the plug-in (you need to have write access to the $(FRAMAC_LIBDIR)/plugins directory).

Just launch frama-c (without any option): the Hello plug-in is now always loaded, without the need to pass other options to the command line. Check it with the command frama-c -hello-help.

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 Db.Main.extend should be at the top, as in the following example. The PLUGIN_CMO variable must contain the list of file names, in the correct OCaml build order.

File ./Makefile

FRAMAC_SHARE := $(shell frama-c-config -print-share-path)
FRAMAC_LIBDIR := $(shell frama-c-config -print-libpath)
PLUGIN_NAME = Hello
PLUGIN_CMO = hello_options hello_print hello_run
include $(FRAMAC_SHARE)/Makefile.dynamic

20
2.3. THE HELLO PLUG-IN

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**

```ocaml
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**

```ocaml
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.writeln chan "%s" 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**

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

The plug-in can be tested again by running:

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

### 2.3.6 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*.

To build these tests, the location of the subdirectories containing them must be indicated in the `Makefile` through the variable `PLUGIN_TESTS_DIRS` set at `hello` as this will be the name of the subdirectory of `.tests/` where the plug-in’s tests will be located:

```
File ./Makefile

FRAMAC_SHARE := $(shell frama-c-config -print-share-path)
FRAMAC_LIBDIR := $(shell frama-c-config -print-libpath)
PLUGIN_NAME = Hello
PLUGIN_CMO = hello_options hello_print hello_run
PLUGIN_TESTS_DIRS := hello
include $(FRAMAC_SHARE)/Makefile.dynamic
```

This enables the creation of a `./tests/ptests_config` file holding the environment needed by *ptests* to run the plug-in’s tests by running:

```
$ make
<...>
Generating tests/ptests_config
```

For non-regression testing, the current behaviour 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 directory must contain a `run.config` comment with the test directives and the C source code used for the test. (There are other ways to declare and control tests as developed in Section 4.5.2.) For this tutorial, there will be no such source code. A file `./tests/hello/hello_test.c` is then created:

```
File ./tests/hello/hello_test.c

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

In this file, there is only one directive `OPT: -hello` which requires to run Frama-C on this test with the `-hello` option. A look at Section 4.5.5 gives you an idea of the kind of directives which can be used to test plug-ins.

Once the `run.config` has been configured, it becomes possible to get the output generated by the plug-in:
$ ptests.opt -show
Env:
<...>
Command:
/usr/local/bin/frama-c.byte tests/hello/hello_test.c -check -hello \
2>tests/hello/result/hello_test.err.log >tests/hello/result/hello_test.res.log
[kernel] Parsing FRAMAC_SHARE/libc/__fc_builtin_for_normalization.i (no preprocessing)
[kernel] Parsing tests/hello/hello_test.c (with preprocessing)
[hello] Hello, world!
Env:
<...>
Command:
<...>

The option -show runs the tests and shows the output (but not their examination). Other options are detailed in Section 4.5.4 to give a better idea of the extent to which ptests is configurable. Under Env is displayed the context (coming from the file ./hello/ptests_config). Then, Command shows the executed command for this test case followed by bash pipes to control the dataflow. (Note the -hello option which has been passed to Frama-C as requested by OPT: -hello in the run.config.) Two outputs are considered as results for each test: an error output and a result output. These outputs are logged in the two following files: ./tests/hello/result/hello_test.err.log and ./tests/hello/result/hello_test.res.log. The three lines beginning by [kernel] and [hello] are the actual non-erroneous outputs made by Frama-C: two of them are generated by the kernel (which could be muted by changing the verbosity) while the third one is generated by our plug-in Hello. Env and Command parts appear once again because some test strategies need it, which is not the case in the simple setting of this tutorial.

Once you have verified the output is as expected, set it as an oracle to be used for later non-regression tests by running:

```
$ ptests.opt -update
```

This command copies the two log files to ./tests/hello/oracle/hello_test.err.oracle and ./tests/hello/oracle/hello_test.res.oracle.

The setting of this test case is now finished. Let’s now assume the plug-in is later erroneously modified as follows:

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

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

where Hello, world! is incorrectly changed to Hello world!. Running the command:

```
$ make
<...>
$ make tests
TESTING PLUG-IN Hello
% Dispatch finished, waiting for workers to complete
Env:
<...>
Command:
```
% Comparisons finished, waiting for diffs to complete
--- tests/hello/oracle/hello_test.res.oracle 2017-06-02 14:39:49.407624816 +0200
+++ tests/hello/result/hello_test.res.log 2017-06-02 14:40:03.483624679 +0200
@@ -1,3 +1,3 @@
- [kernel] Parsing FRAMAC_SHARE/libc/__fc_builtin_for_normalization.i (no preprocessing)
+ [kernel] Parsing tests/hello/hello_test.c (with preprocessing)
- [hello] Hello, world!
+ [hello] Hello world!
% Diffs finished. Summary:
Run = 1
Ok = 1 of 2
Time = 0.380000 s.
real 0.45
user 0.38
sys 0.03
displays the differences (à la diff) between the current executions 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. One test is marked as being successful: that is the test comparing stderr (which contains an empty string). After fixing the OCaml code, running make && make tests again shows that all test cases are successful.

You may use other Frama-C’s 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) for instance. Additional information about plug-in testing is available in Sections 4.5 and 5.3.

2.3.7 Documenting your Source Code

Frama-C automatically generates the documentation of plug-ins when the command make doc is run. This relies on ocamldoc and requires the plug-in to be documented following the ocamldoc guidelines (please refer to the corresponding chapter in [13]).

We show here how the Hello plug-in could be slightly documented and use ocamldoc features such as @-tags and cross references:

```ocaml
File ./hello_options.ml

(/**
 ** 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.printf chan "%s\n" msg;
  flush chan;
  close_out chan
with Sys.error exc as exc →
  let msg = Printexc.to_string exc in
  Printf.eprintf "There was an error: %s\n" msg
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\(^7\), to hook a plug-in, to interface a plug-in with other plug-ins, to extend the GUI\(^8\), to make a plug-in usable by others, to make write a plug-in into the journal, to configure a script, and to make a plug-in usable in a multi-projects setting.

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.

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 to visit 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.

\(^6\)The direct link is: http://frama-c.com/download.html.
\(^7\)Abstract Syntax Tree
\(^8\)Graphical User Interface
2.4. THE VIEWCFG PLUG-IN

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:

```plaintext
open Cil_types

let print_stmt out
    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 prints the different Cil types. The documentation of these module is available on the Frama-C website9, 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 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.

```plaintext
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 4.17.4).

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

---

Cil.DoChildrenPost is one of the possible visitAction, that tells the visitor what to do after the function is executed. With DoChildrenPost func, the func argument is called once the children have been executed: here we close the parenthesis once that 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.

Cil.SkipChildren tells the visitor not to visit the children nodes, which makes it more efficient\textsuperscript{10}.

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

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:

Hooking into Frama-C

It just remains to hook this script into Frama-C.

\textsuperscript{10}In a copying visitor, Cil.JustCopy should have been used instead.
Assuming the script is called `cfg_print.ml`, it can then be run with:

```
frama-c -load-script cfg_print.ml [other_options] file.c [file2.c]
```

And the graph can be visualized with

```
dotty cfg.out
```

This produces a graph like in Figure 2.2

```c
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.

**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")`) such statements must be protected using the "%S" Format directive;

- The script could be transformed into a regular plug-in, by registering into Frama-C, and taking options from the command line; for instance to compute the control flow graph of a single function given as an argument;

- The graphs could be fancier, in particular by distinguishing between branching nodes and plain ones, or showing exit of blocks as well as their beginning; or linking a call with the called function.

We will concentrate on another extension, which is to reuse the analysis of the value Frama-C 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.
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.

```plaintext
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
  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 () = Db.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 2.4.9 of this tutorial and section 4.11 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 value plug-in.
2.4. THE VIEWCFG PLUG-IN

2.4.3 Interfacing with a kernel-integrated plug-in

Kernel-integrated plug-ins, such as `value`, use a special mechanism to statically register their APIs for other plug-ins that wish to access them. This mechanism is the `Db` module of the Frama-C kernel, the entry point for all kernel-integrated plug-ins. By using the functions exported through the `Db.Value` module, our plug-in will obtain reachability information computed by `value`.

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:

```plaintext
method vstmt_aux s =
  let color =
    if Db.Value.is_computed () then
      let state = Db.Value.get_stmt_state s in
      let reachable = Db.Value.is_reachable state in
      if reachable then "fillcolor="#ccffcc" style=filled"
      else "fillcolor=pink style=filled"
    else ""
  in
  Format.fprintf out "@[s%d [label=%S %s]@]@;@

  s.sid (Pretty_utils.to_string print_stmt s.skind) color;
  List.iter
    (fun succ → Format.fprintf out "@[s%d → s%d;@]@" 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, frama-c should be launched with:

```
frama-c test.c -load-script cfg_print.ml -val -then -cfg && dotty cfg.out
```

Note that the relative order of the parameters `-load-script` and `-val` in this example is not relevant (see Section 4.13 for details). However, it is important to ensure that `-cfg` is separated from `-val` by a `then`; otherwise, it is not guaranteed that the `value` plug-in will run before the ViewCfg plug-in, which might lead to a non-colored graph in some cases, and colored in others.

The resulting graph is shown in Figure 2.3.

2.4.4 Extending the Frama-C GUI

In this section, we will extend our plug-in so that the control flow graph can be displayed interactively. For that, we will extend the Frama-C GUI so that when you right-click on a function in the code, a new “Show CFG” item appears, that displays the control flow graph of the function in a dialog box. This is achieved just by appending the following pieces of code at the end of the `cfg_print.ml` file.

Currently, we used a visitor that outputs a DOT file with the CFG of all functions of all files. We use `dump_function` to output the CFG of a single function instead.

```plaintext
let dump_function fundec fmt =
  Format.fprintf fmt "@[< hov 2> digraph cfg { @ ";
  ignore (Visitor.visitFramacFunction (new print_cfg fmt) fundec);
  Format.fprintf fmt "}@@@\n"
```

The resulting graph is shown in Figure 2.3.
File `test.c`

```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.

We reused the `print_cfg` visitor, but we selected a different starting point. The argument `fundec` gets type `Cil_types.fundec`, which is the CIL type representing a function definition.

Now we write the GUI extension code:

```c
let cfg_selector
     (popup_factory:GMenu.menu GMenu.factory) main_ui ~button:_ localizable =
match localizable with
(* Matches global declarations that are functions. *)
| Pretty_source.PVDecl(_, _, (_vtype = TFun(_,_,_,_,_)) as vi)) ->
  let callback () =
    let kf = Globals.Functions.get vi in
    let fundec = Kernel_function.get_definition kf in
    let window:GWindow.window = main_ui#main_window in
    Dgraph_helper.graph_window_through_dot
    ~parent:window ~title:"Control flow graph"
    (dump_function fundec)
    in
    ignore (popup_factory#add_item "Show _CFG" ~callback)
| -> ()

let main_gui main_ui = main_ui#register_source_selector cfg_selector
let () = Design.register_extension main_gui
```

Let us explain this code from the end. `Design.register_extension` is the entry point for extending the GUI. Its argument is a function which takes as argument an object corresponding to the main window of the Frama-C GUI. This object provides access to the main widgets of the window, and several extension points.
2.4. THE VIEWCFG PLUG-IN

Here we have implemented a single extension, the “source selector”, that allows to add entries to menu obtained when right-clicking on the source. This is implemented by the 
\texttt{cfg\_selector} function.

This function takes a \texttt{localizable} argument, which gives information on where the user clicks on the source. Here we do something only if the user clicks on the declaration of a variable whose type is a function (\textit{i.e.} when the user clicked on a function declaration or definition). In that case, we add an item to the popup menu, that calls the \texttt{callback} function if clicked. The \texttt{callback} function calls a Frama-C GUI function that displays a graph from DOT printing functions. It uses several important Frama-C APIs: \texttt{Globals} and \texttt{Kernel\_function}, which contain several functions for manipulating globals and functions.

Note that this GUI extension could also have been done through a script (instead of a plug-in), but it would have been less than ideal. In particular, the GUI OCaml modules are available only when a script is loaded with \texttt{frama-c\_gui}, and not when loaded with \texttt{frama-c}. When the user wants to view the CFG from the GUI, outputting the CFG of all functions in \texttt{cfg\_out} is useless. A better architectural solution is to split our plug-in in several files, with its own Makefile, to better manage its functionalities.

2.4.5 Splitting files and writing a Makefile

The Frama-C plug-in development environment allows to split GUI-related and non-GUI related modules, so that GUI-related modules are loaded and run only if Frama-C is executed with \texttt{frama-c\_gui}. This requires splitting the module into several files. We choose the following architecture:

- \texttt{cfg\_options\_ml} implements plug-in registration and configuration options;
- \texttt{cfg\_core\_ml} implements the main functions for computing the CFG;
- \texttt{cfg\_register\_ml} implements “global” computation of the CFG using the \texttt{-cfg} option, and hooking into the Frama-C main loop;
- \texttt{cfg\_gui\_ml} implements GUI registration.

Dependencies between the modules\textsuperscript{11} is presented on Figure 2.4.

To break recursive dependencies between OCaml modules, it is typical that plug-in registration is done at the bottom of the module hierarchy, while definition of the \texttt{run} function is at the top. The GUI is also at the top of the hierarchy: the Frama-C \texttt{Makefile} requires that normal plug-in modules do not depend on GUI modules. Note that currently, the dependency from \texttt{Cfg\_core} and \texttt{Cfg\_gui} to \texttt{Cfg\_register} is artificial, but in future evolutions the GUI could depend on configuration options.

File \texttt{Makefile}

```
FRAMAC\_SHARE := $(shell frama-c\-config -print-share\-path)
FRAMAC\_LIBDIR := $(shell frama-c\-config -print-libpath)
PLUGIN\_NAME = ViewCfg
PLUGIN\_CMD0 = cfg\_options cfg\_core cfg\_register
PLUGIN\_GUI\_CMD0 = cfg\_gui
include $(FRAMAC\_SHARE)/Makefile\_dynamic
```

\textsuperscript{11}This graphic is generated in file \texttt{doc/code/modules\_dot} after running \texttt{make doc}.
In the Makefile, the PLUGIN_CMO variable must contain the list of file names of the ml files, in the correct OCaml build order. Modules in PLUGIN_CMO must not depend on modules in PLUGIN_GUI_CMO.

We also need to add an interface for the whole plug-in:

```ml
(** ViewCfg plug-in.

   No function is exported. *)
```

Here is the listing for the different modules:

**File cfg_options.ml**

```ml
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
    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)
```

**File cfg_core.ml**

```ml
module Options = Cfg_options
open Cil_types

let print_stmt out = function
```
2.4. THE VIEWCFG PLUG-IN

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

    method! vfile _ =
        Format.printf out "@[< hov 2> digraph cfg {@ 
            Cil.DoChildrenPost (fun f -> Format.printf out "}@[0]@."; f)
        }

    method! vglob_aux g =
        match g with
        | GFun(f, _) ->
            Format.printf out "@[< hov 2> subgraph cluster_%a {@ 
                Printer.pp_varinfo f.svar
                Printer.pp_varinfo f.svar;
                Cil.DoChildrenPost(fun g -> Format.printf out "}@[0]@."; g)
        |
        method! vstmt_aux s =
            let color =
                if Db.Value.is_computed () then
                    let state = Db.Value.get_stmt_state s in
                    let reachable = Db.Value.is_reachable state in
                    if reachable then "fillcolor=\"#ccffcc\" style=filled"
                    else "fillcolor=pink style=filled"
                else ""
            in
            Format.printf out "@[s%d@ [label=%S %s]@]@ "
            s.sid (Pretty_utils.to_string print_stmt s.skkind) color;
            List.iter
                (fun succ -> Format.printf out "@[s%d -> s%d]@ " s.sid succ.sid)
                s.succs;
            Cil.DoChildren

        end

    let dump_function fundec fmt =
        Format.printf fmt "@[< hov 2> digraph cfg {@ 
            ignore (Visitor.visitFramacFunction (new print_cfg fmt) fundec);
        }
```

```
CHAPTER 2. TUTORIAL

File cfg_register.ml

```ocaml
open Cfg_options
open Cfg_core

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 () = Db.Main.extend run
```

File cfg_gui.ml

```ocaml
open Cil_types
open Cfg_core
module Options = Cfg_options

let cfg_selector
  (popup_factory:GMenu.menu GMenu.factory) main_ui ~button:_ localizable =
match localizable with
(* Matches global declarations that are functions. *)
| Pretty_source.PVDecl(_, _, ({vtype = TFun(_,_,_,_) as vi})) →
  let callback () =
    let kf = Globals.Functions.get vi in
    let fundec = Kernel_function.get_definition kf in
    let window:GWindow.window = main_ui#main_window in
    Dgraph_helper.graph_window_through_dot ~parent:window ~title:"Control flow graph"
    (dump_function fundec)
  in
  ignore (popup_factory#add_item "Show _CFG" ~callback)
  | _ → ()

let main_gui main_ui = main_ui#register_source_selector cfg_selector

let () = Design.register_extension main_gui
```

2.4.6 Getting your Plug-in Usable by Others

*Not written yet*: please report as “feature request” on [http://bts.frama-c.com](http://bts.frama-c.com) if you really need this section.

2.4.7 Writing your Plug-in into the Journal

*Not written yet*: please report as “feature request” on [http://bts.frama-c.com](http://bts.frama-c.com) if you really need this section.
2.4. THE VIEWCFG PLUG-IN

2.4.8 Writing a Configure Script

_Not written yet:_ please report as “feature request” on http://bts.frama-c.com if you really need this section.

2.4.9 Getting your plug-in Usable in a Multi Projects 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, the value plug-in computes, for each statement, a table associating to each AST’s 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.

In this tutorial, we will store the file representing the DOT output of the control flow graph of a function (as needed by `dump_function`) as a string, by using a hashtable from `fundec` to `string`. Storing this string will allow us to memoize 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 hashtable as a Frama-C state is _mandatory_ to ensure Frama-C consistency: for instance, by using a standard OCaml hashtable, a user that would have loaded several session through the 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:

```ocaml
let name = "Data_for_cfg.Cfg_graph_state"
let dependencies = [ Ast.self; Db.Value.self ]
let size = 17
end);
```

The _State_builder_ module provides several functors that help registering states. _State_builder.Hashtbl_ allows the developer to create a hashtable. It is parameterized by a module describing the hashtable and its key, a module describing the data associated to keys, and other informations.

The _Datatype_ and _Cil_datatype_ modules describe the hashtable 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 4.8. _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 hashtable, a name (mainly used for internal debugging), and a list of _dependencies_. Here we expressed that our hashtable
depends on the Ast and the results of the Value plug-in. For instance, whenever the Frama-C
kernel updates one of these states, it will automatically reset our hashtable. 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.

```ocaml
let dump_to_string fundec =
  Self.feedback "Computing CFG for function %s" (fundec.svar.vorig_name);
ignore
  (Visitor.visitFramacFunction (new print_cfg Format.str_formatter) fundec);
Format.flush_str_formatter ()
let dump_to_string_memoized = Cfg_graph_state.memo dump_to_string
let dump_function fundec fmt =
  Format.fprintf fmt "@\[digraph cfg {\%s}@\]@\n"
  (dump_to_string_memoized fundec)
```

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.

If you launch frama-c-gui with the above code, click on functions to view their CFG, and
inspect the console, you will observe that the string “Computing CFG for function ...” is
displayed only once per function.

One can see the effects of the dependency on the Value plug-in by first launching the value
analysis, inspecting the CFG for the f function, then changing the entry point to f in the CFG
and re-running the value analysis. The console indicates that the CFG have been recomputed.
Indeed the state of the Value plug-in, and of its dependencies, was reset when the entry point
was changed.

Another way to observe how Frama-C automatically handles states is to display a CFG, save
the session, close and restart Frama-C, and then reload the session: the control flow graph is
not recomputed, which means that Frama-C has automatically saved the Cfg_graph_state
with the rest of the session. Everything should also work properly when loading several
sessions.

Clearing states, selection and projects

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

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

```ocaml
module Value_is_computed = State_builder.Ref
  (Datatype.Bool)
(struct
  let name = "Data_for_cfg.Value_computed"
  let dependencies = []
```

2.4. The VIEWCFG Plug-in

```ml
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.

```ml
let dump_function fundec fmt =
  if not (Value_is_computed.get () && Db.Value.is_computed () then begin
    Value_is_computed.set true;
    let selection = State_selection.with_dependencies Cfg_graph_state.self in
    Project.clear ~selection () ;
  end;
Format.fprintf fmt "@digraph cfg {%s}@." (dump_to_string_memoized fundec)
```

The only part that need to be explained is the notion of `selection` and `project`. A selection is just a set of states; here we selected the state `Cfg_graph_state` with all its dependencies, as resetting this state would also impact states that would depend on it (even if there is 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 analysis (e.g. with different entry points). Projects consistently groups a version of the AST of the program, with the states related to this AST.

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 to not 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 multiple projects 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 much time 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.
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.
Figure 3.1: Frama-C Architecture Design.
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/plugin_name. Each plug-in is an extension point of Frama-C which has to be registered through Plugin.Register (see Section 4.6). 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 modules Dynamic or Db. The first method is the preferred one, the second one (through module Dynamic) is the only possible one to define mutually dependent plug-ins while the third one (through module Db) is now fully deprecated even if most of the older Frama-C plug-ins are still defined this way.

Plug-ins are usually dynamically linked when Frama-C is booting, even if some older Frama-C plug-ins are still statically linked. However it is still possible to statically link a dynamic plug-in if wanted or required. In particular, OCaml does not support dynamic linking on some hardware architecture [13]: in this case, you have to statically link all 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 FCmodule_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 4.8 and 4.11.

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/visitor` (see Section 4.17). 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

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 4.11.2);
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 4.11.5).

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). Static plug-in development requires some knowledge of autoconf and make. Each section summarizes its own prerequisites at its beginning (if any).

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 chapter to another one. Pointers to reference manuals (Chapter 5) are also provided for readers who want full details about specific parts.

4.1 Frama-C Configure.in

Target readers: not for standard plug-ins developers.

Prerequisite: Knowledge of autoconf and shell programming.

\(^1\)It is fortunately quite difficult (but not impossible) to fall into the worst situation by mistake if you are not a kernel developer.
CHAPTER 4. ADVANCED PLUG-IN DEVELOPMENT

In this Section, we detail how to modify the file `configure.in` in order to configure plug-ins (Frama-C configuration has been introduced in Section 2.4.8 and 2.4.5).

First Section 4.1.1 introduces the general principle and organisation of `configure.in`. Then Section 4.1.2 explains how to configure a new simple plug-in without any dependency. Next we show how to exhibit dependencies with external libraries and tools (Section 4.1.4) and with other plug-ins (Section 4.1.5). Finally Section 4.1.3 presents the configuration of external libraries and tools needed by a new plug-in but not used anywhere else in Frama-C.

4.1.1 Principle

When you execute `autoconf`, file `configure.in` is used to generate the `configure` script. Each Frama-C user executes this script to check his/her system and determine the most appropriate configuration: at the end of this configuration (if successful), the script summarizes the status of each plug-in, which can be:

- **available** (everything is fine with this plug-in);
- **partially available**: either an optional dependency of the plug-in is not fully available, or a mandatory dependency of the plug-in is only partially available; or
- **not available**: either the plug-in itself is not provided by default, or a mandatory dependency of the plug-in is not available.

The important notion in the above definitions is dependency. A dependency of a plug-in `p` is either an external library/tool or another Frama-C plug-in. It is either mandatory or optional. A mandatory dependency must be present in order to build `p`, whereas an optional dependency provides features to `p` that are additional but not highly required (especially `p` must be compilable without any optional dependency).

Hence, for the plug-in developer, the main role of `configure.in` is to define the optional and mandatory dependencies of each plug-in. Another standard job of `configure.in` is the addition of options `--enable-p` and `--disable-p` to `configure` for a plug-in `p`. These options respectively forces `p` to be available and disables `p` (its status is automatically “not available”).

Indeed `configure.in` is organised in different sections specialized in different configuration checks. Each of them begins with a title delimited by comments and it is highlighted when `configure` is executed. These sections are described in Section 5.1. Now we focus on the modifications to perform in order to integrate a new plug-in in Frama-C.

4.1.2 Addition of a Simple Plug-in

In order to add a new plug-in, you have to add a new subsection for the new plug-in to Section Plug-in wished. This action is usually very easy to perform by copying/pasting from another existing plug-in (e.g. occurrence) and by replacing the plug-in name (here occurrence) by the new plug-in name in the pasted part. In these sections, plug-ins are sorted according to a lexicographic ordering.

For instance, Section Wished Plug-in introduces a new sub-section for the plug-in occurrence in the following way.
4.1. FRAMA-C CONFIGURE.IN

```c
# occurrence
 check_plugin(occurrence, src/plugins/occurrence, [support for occurrence analysis], yes)
```

- The first argument is the plug-in name,
- the second one is the name of directory containing the source files of the plug-in (usually a sub-directory of src/plugins),
- the third one is an help message for the -enable-occurrence option of configure,
- the last one indicates if the plug-in is enabled by default (yes/no).

The plug-in name must contain only alphanumeric characters and underscores. It must be the same as the name value given as argument to the functor Plugin.Register of section 4.6 (with spaces replaced by underscore). It must also be the same (modulo upper/lower case) as the PLUGIN_NAME variable given in the plugin’s Makefile presented in section 4.4.

The macro check_plugin sets the following variables: FORCE_OCCURRENCE, REQUIRE_OCCURRENCE, USE_OCCURRENCE, and ENABLE_OCCURRENCE.

The first variable indicates if the user explicitly requires the availability of occurrence via setting the option --enable-occurrence. The second and third variables are used by others plug-ins in order to handle their dependencies (see Section 4.1.5). The fourth variable ENABLE_OCCURRENCE indicates the plug-in status (available, partially available or not available). If --enable-occurrence is set, then ENABLE_OCCURRENCE is yes (plug-in available); if --disable-occurrence is set, then its value is no (plug-in not available). If no option is specified on the command line of configure, its value is set to the default one (according to the value of the fourth argument of check_plugin).

4.1.3 Configuration of New Libraries or Tools

Some plug-ins need additional tools or libraries to be fully functional. The configure script takes care of these in two steps. First, it checks that an appropriate version of each external dependency exists on the system. Second, it verifies for each plug-in that its dependencies are met. Section 4.1.4 explains how to make a plug-in depend on a given library (or tool). The present section deals with the first part, that is how to check for a given library or tool on a system. Configuration of new libraries and configuration of new tools are similar. In this section, we therefore choose to focus on the configuration of new libraries. This is done by calling a predefined macro called configure_library. The configure_library macro takes three arguments. The first one is the (uppercase) name of the library, the second one is a filename which is used by the script to check the availability of the library. In case there are multiple locations possible for the library, this argument can be a list of filenames. In this case, the argument must be properly quoted (i.e. enclosed in a [ , ] pair). Each name is checked in turn. The first one which corresponds to an existing file is selected. If no name in the list corresponds to an existing file, the library is considered to be unavailable. The

---

For tools, there is a macro configure_tool which works in the same way as configure_library.
last argument is a warning message to display if a configuration problem appears (usually because the library does not exist). Using these arguments, the script checks the availability of the library.

Results of this macro are available through two variables which are substituted in the files generated by configure.

- **HAS_library** is set to **yes** or **no** depending on the availability of the library
- **SELECTED_library** contains the name of the version selected as described above.

When checking for OCaml libraries and object files, remember that they come in two flavors: bytecode and native code, which have distinct suffixes. Therefore, you should use the variables **LIB_SUFFIX** (for libraries) and **OBJ_SUFFIX** (for object files) to check the presence of a given file. These variables are initialized at the beginning of the configure script depending on the availability of a native-code compiler on the current installation.

**Example 4.1** The library **Lablgtksourceview2** (used to have a better rendering of C sources in the GUI) is part of **Lablgtk2** [10]. This is checked through the following command, where **LABLGTRPATH_FOR_CONFIGURE** is the path where configure has found Lablgtk2 itself.

```configure
configure_library( [GTKSOURCEVIEW],
  [$LABLGTRPATH_FOR_CONFIGURE/lablgtksourceview2.$LIB_SUFFIX],
  [lablgtksourceview not found])
```

### 4.1.4 Addition of Library/Tool Dependencies

Dependencies upon external tools and libraries are governed by two macros:

- **plugin_require_external(plugin,library)** indicates that **plugin** requires **library** in order to be compiled.

- **plugin_use_external(plugin,library)** indicates that **plugin** uses **library**, but can nevertheless be compiled if **library** is not installed (potentially offering reduced functionality).

**Recommendation 4.1** The best place to perform such extensions is just after the addition of $p$ which sets the value of **ENABLE_p**.

**Example 4.2** Plug-in **gui** requires **Lablgtk2** and **GnomeCanvas**. It also optionally uses **DOT** for displaying graphs (graph cannot be displayed without this tool). So, just after its declaration, there are the following lines in **configure.in**.

```configure
plugin_require_external(gui,lablgtk)
plugin_require_external(gui,gnomecanvas)
plugin_use_external(gui,dot)
```
4.2. PLUG-IN SPECIFIC CONFIGURE.AC

4.1.5 Addition of Plug-in Dependencies

Adding a dependency with another plug-in is quite the same as adding a dependency with an external library or tool (see Section 4.1.4). For this purpose, configure.in uses two macros

- `plugin_require(plugin1, plugin2)` states that `plugin1` needs `plugin2`.
- `plugin_use(plugin1, plugin2)` states that `plugin1` can be used in absence of `plugin2`, but requires `plugin2` for full functionality.

There can be mutual dependencies between plug-ins. This is for instance the case for plug-ins value and from.

4.2 Plug-in Specific Configure.ac

**Target readers:** standard plug-ins developers.

**Prerequisite:** Knowledge of autoconf and shell programming.

External plug-ins can have their own configuration file, and can rely on the macros defined for Frama-C. In addition, as mentioned in section 4.4.2, those plug-ins can be compiled directly from Frama-C’s own Makefile. In order for them to integrate well in this setting, they should follow a particular layout, described below. First, they need to be able to refer to the auxiliary configure.ac file defining Frama-C-specific macros when they are used as stand-alone plug-ins. This can be done by the following code

```makefile
m4_define([plugin_file],Makefile)

m4_define([FRAMAC_SHARE_ENV],
    [m4_normalize(m4_esyscmd([echo $FRAMAC_SHARE]))])

m4_define([FRAMAC_SHARE],
    [m4_ifval(FRAMAC_SHARE_ENV,[FRAMAC_SHARE_ENV],
        [m4_esyscmd(frama-c -print-path)])])

m4_ifndef([FRAMAC_M4_MACROS],
    [m4_include(FRAMAC_SHARE/configure.ac)])
```

`plugin_file` is the file which must be present to ensure that autoconf is called in the appropriate directory (see documentation for the AC_INIT macro of autoconf). configure.ac can be found in two ways: either by relying on the FRAMAC_SHARE shell variable (when Frama-C is not installed, i.e. when configuring the plug-in together with the main Frama-C), or by calling an installed Frama-C (when installing the plug-in separately). The inclusion of configure.ac needs to be guarded to prevent multiple inclusions, as the configuration file of the plug-in might itself be included by configure.in (see section 4.4.2 for more details).

The configuration of the plug-in itself or related libraries and tools can then proceed as described in Sections 4.1.2 and 4.1.3. References to specific files in the plug-in source directory should be guarded with the following macro:

```makefile
PLUGIN_RELATIVE_PATH(file)
```
If the external plug-in has some dependencies as described in sections 4.1.4 and 4.1.5, the configure script `configure` must check that all dependencies are met. This is done with the following macro:

```bash
check_plugin_dependencies
```

An external plug-in can have dependencies upon previously installed plug-ins. However two separately installed plug-ins can not be mutually dependent on each other. Nevertheless, they can be compiled together with the main Frama-C sources using the `--enable-external` option of `configure` (see section 4.4.2 for more details).

Finally, the configuration must end with the following command:

```bash
write_plugin_config(files)
```

where `files` are the files that must be processed by configure (as in `AC_CONFIG_FILES` macro). `PLUGIN_RELATIVE_PATH` is unneeded here.

For technical reasons, the macros `configure_library`, `configure_tool`, `check_plugin_dependencies`, and `write_plugin_config` must not be inside a conditional part of the `configure` script. More precisely, they are using the diversion mechanism of `autoconf` in order to ensure that the tests are performed after all dependencies information has been gathered from all existing plugins. Diversion is a very primitive, text-to-text transformation. Using those macros within a conditional (or anything that alters the control-flow of the script) is likely to result in putting some unrelated part of the script in the same branch of the conditional.

### 4.3 Frama-C Makefile

**Target readers:** *not for standard plug-in developers.*

**Prerequisite:** *Knowledge of make.*

In this section, we detail the use of `Makefile` dedicated to Frama-C compilation. This file is split in several sections which are described in Section 5.2.2. By default, executing `make` only displays an overview of commands. For example, here is the output of the compilation of source file `src/kernel_services/plugin_entry_points/db.cmo`.

```
$ make src/kernel_services/plugin_entry_points/db.cmo
Ocamlc src/kernel_services/plugin_entry_points/db.cmo
```

If you wish the exact command line, you have to set variable `VERBOSEMAKE` to `yes` like below.

```
$ make VERBOSEMAKE=yes src/kernel_services/plugin_entry_points/db.cmo
```

### 4.4 Plug-in Specific Makefile

**Prerequisite:** *Knowledge of make.*
4.4. PLUG-IN SPECIFIC MAKEFILE

4.4.1 Using Makefile.dynamic

In this section, we detail how to write a Makefile for a given plug-in. Even if it is still possible to write such a Makefile from scratch, Frama-C provides a generic Makefile, called Makefile.dynamic, which helps the plug-in developer in this task. This file is installed in the Frama-C share directory. So for writing your plug-in specific Makefile, you have to:

1. set some variables for customizing your plug-in;
2. include Makefile.dynamic.

Example 4.3 A minimal Makefile is shown below. That is the Makefile of the plug-in Hello World presented in the tutorial (see Section 2.3). Each variable set in this example has to be set by any plug-in.

```
FRAMAC_SHARE := $(shell frama-c-config -print-share-path)
FRAMAC_LIBDIR := $(shell frama-c-config -print-libpath)
PLUGIN_NAME = Hello
PLUGIN_CMO = hello_options hello_print hello_run
include $(FRAMAC_SHARE)/Makefile.dynamic
```

FRAMAC_SHARE must be set to the Frama-C share directory while FRAMAC_LIBDIR must be set to the Frama-C lib directory. PLUGIN_NAME is the capitalized name of your plug-in while PLUGIN_CMO is the list of the files .cmo generated from your OCaml sources. Note that, by default, during compilation of your plug-in all warnings are disabled; it is recommended to define environment variable DEVELOPMENT=yes to enable the standard set of compilation warnings.

To run your specific Makefile, you must have properly installed Frama-C before, or compile your plugin together with Frama-C, as described in section 4.4.2.

You may possibly need to do make depend before running make.

Which variable can be set and how they are useful is explained Section 5.2.3. Furthermore, Section 5.2.5 explains the specific features of Makefile.dynamic.

4.4.2 Compiling Frama-C and external plug-ins at the same time

Target readers: plug-in developers using the git repository of Frama-C.

It is also possible to get a completely independent plug-in recompiled and tested together with Frama-C’s kernel. For that, Frama-C must be aware of the existence of the plug-in. This can be done in two ways:

- All sub-directories of src/plugins directory in Frama-C sources which are not known to Frama-C’s kernel are assumed to be external plug-ins.
- One can use the --enable-external option of configure which takes as argument the path to the plug-in
In the first case, the plug-in behaves as any other built-in plug-in: `autoconf` run in Frama-C’s main directory will take care of it and it can be enabled or disabled in the same way as the others. If the plug-in has its own `configure.in` or `configure.ac` file, the configuration instructions contained in it (in particular additional dependencies) will be read as well.

In the second case, the plug-in is added to the list of external plug-ins at configure time. If the plug-in has its own `configure`, it is run as well.

Provided it properly uses the variables set by `Makefile.dynamic`, the plug-in’s `Makefile` does not require specific adaptations depending on whether it is compiled together with the kernel or with respect to an already-existing Frama-C installation. It is however possible to check the compilation mode with the `FRAMAC_INTERNAL` variable, which is set to `yes` when compiling together with Frama-C kernel and to `no` otherwise.

In addition, if a plug-in wishes to install custom files to `FRAMAC_LIBDIR` through the `install::` target, this target must depend on `clean-install`. Indeed, Frama-C’s main `Makefile` removes all existing files in this directory before performing a fresh installation, in order to avoid potential interference with an obsolete (and usually incompatible) module from a previous installation. Adding the dependency thus ensures that the removal will take place before any new file has been installed in the directory.

**Example 4.4** If a plug-in wants to install `external/my_lib.cm*` in addition to the normal plugin files, it should use the following code:

```plaintext
install:: clean-install
$(PRINT_INSTALL) "My beautiful library"
$(MKDIR) $(FRAMAC_LIBDIR)
$(CP) external/my_lib.cm* $(FRAMAC_LIBDIR)
```

### 4.5 Testing

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

p tests runs the Frama-C toplevel on each specified test (which are usually C files). Specific directives can be used for each test. Each result of the execution is compared from 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 that the number of tests because standard and error outputs are compared separately.

First Section 4.5.1 shows how to use ptests. Next Section 4.5.2 introduces how to use pre-defined directives to configure tests, while Section 4.5.3 explains how to set up various testing goals for the same test base. Last Section 4.5.4 details ptests’ options, while Section 4.5.5 describes ptests’ directive.

#### 4.5.1 Using ptests

If you’re using a `Makefile` written following the principles given in section 4.4, the simplest way of using ptests is through `make tests` which is roughly equivalent to

```sh
$ time ./bin/ptests.opt
```

or
4.5. TESTING

depending on whether you’re inside Frama-C’s sources or compiling a plug-in against an already installed Frama-C distribution.

A specific target \$(PLUGIN_NAME)_TESTS will specifically run the tests of the plugin. One can add new tests as dependencies of this target. The default tests are run by the target \$(PLUGIN_NAME)_DEFAULT_TESTS.

ptests.opt runs tests belonging to a sub-directory of directory tests that is specified in ptests configuration file. This configuration file, tests/ptests_config, is automatically generated by Frama-C’s Makefile from the options you set in your plugin’s Makefile. ptests also accepts specific test suites in arguments. A test suite is either the name of a sub-directory in directory tests or a filename (with its path relative to the current directory).

Example 4.5 If you want to test plug-in sparecode and specific test tests/pdg/variadic.c, just run

| $ time ptests.opt
| Depending on whether you’re inside Frama-C’s sources or compiling a plug-in against an already installed Frama-C distribution.

A specific target \$(PLUGIN_NAME)_TESTS will specifically run the tests of the plugin. One can add new tests as dependencies of this target. The default tests are run by the target \$(PLUGIN_NAME)_DEFAULT_TESTS.

ptests.opt runs tests belonging to a sub-directory of directory tests that is specified in ptests configuration file. This configuration file, tests/ptests_config, is automatically generated by Frama-C’s Makefile from the options you set in your plugin’s Makefile. ptests also accepts specific test suites in arguments. A test suite is either the name of a sub-directory in directory tests or a filename (with its path relative to the current directory).

Example 4.5 If you want to test plug-in sparecode and specific test tests/pdg/variadic.c, just run

| $ time ptests.opt

which should display (if there are 7 tests in directory tests/sparecode)

| % Dispatch finished, waiting for workers to complete
| % Comparisons finished, waiting for diffs to complete
| % Diffs finished. Summary:
| Run = 8
| Ok = 16 of 16

ptests accepts different options which are used to customize test sequences. These options are detailed in Section 4.5.4.

Example 4.6 If the code of plug-in plug-in has changed, a typical sequence of tests is the following one.

| $ ./bin/ptests.opt plug-in
| $ ./bin/ptests.opt -update plug-in
| $ make tests

So we first run the tests suite corresponding to plug-in in order to display what tests have been modified by the changes. After checking the displayed differences, we validate the changes by updating the oracles. Finally we run all the test suites in order to ensure that the changes do not break anything else in Frama-C.

Example 4.7 For adding a new test, the typical sequence of command is the following.

| $ ./bin/ptests.opt -show tests/plug-in/new_test.c
| $ ./bin/ptests.opt -update tests/plug-in/new_test.c
| $ make tests

We first ask ptests to print the output of the test on the command line, check that it corresponds to what we expect, and then take it as the initial oracle. If some changes have been made to the code in order to let new_test.c pass, we must of course launch the whole test suite and check that all existing tests are alright.

If you’re creating a whole new test suite suite, don’t forget to create the sub-directories suite/result and suite/oracle where ptests will store the current results and the oracles for all the tests in suite.
4.5.2 Configuration

In order to exactly perform the test that you wish, some directives can be set in three different places. We indicate first these places and next the possible directives.

The places are:

- inside file tests/test_config;
- inside file tests/subdir/test_config (for each sub-directory subdir of tests); or
- inside each test file, in a special comment of the form

```c
/* run.config
   ... directives ...
*/
```

In each of the above case, the configuration is done by a list of directives. Each directive has to be on one line and to have the form

`CONFIG_OPTION: value`

There is exactly one directive by line. The different directives (i.e. possibilities for `CONFIG_OPTION`) are detailed in Section 4.5.5.

Note that some specific configurations require dynamic linking, which is not available on all platforms for native code. `ptests` takes care of reverting to bytecode when it detects that the `OPT`, `EXECNOW`, or `EXEC` options of a test require dynamic linking. This occurs currently in the following cases:

- `OPT` contains the option `-load-script`
- `OPT` contains the option `-load-module`
- `EXECNOW` and `EXEC` use `make` to create a `.cmxs`

**Concurrency issues:** tests using compiled modules (`-load-script` or `-load-module`) may lead to concurrency issues when the same module is used in different test files, or in different test cases within the same file. One way to avoid issues is to serialize tests via `EXECNOW` directives, e.g. by using `make` to compile a `.cmxs` from the `.ml` file, and then loading the `.cmxs` in the test cases, as in the example below.

```bash
EXECNOW: make -s @PTEST_DIR@/@PTEST_NAME@.cmxs
STDOPT: #"-load-module @PTEST_DIR@/@PTEST_NAME.cmxs" ...
STDOPT: #"-load-module @PTEST_DIR@/@PTEST_NAME.cmxs" ...
```

In addition, if the same script `tests/suite/script.ml` is shared by several test files, the `EXECNOW` directive should be put into `tests/suite/test_config`.

**Example 4.8** Test `tests/sparecode/calls.c` declares the following directives.

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

These directives state that we want to test sparecode and slicing analyses on this file. Thus running the following instruction executes two test cases.

```
$ ./bin/ptests.opt tests/sparecode/calls.c
% Dispatch finished, waiting for workers to complete
% Comparisons finished, waiting for diffs to complete
% Diffs finished. Summary:
Run = 2
Ok = 4 of 4
```

4.5.3 Alternative Testing

You may want to set up different testing goals for the same test base. Common cases include:

- checking the result of an analysis with or without an option;
- checking a preliminary result of an analysis, in particular if the complete analysis is costly;
- checking separately different results of an analysis.

This is possible with option `-config` of `ptests`, which takes as argument the name of a special test configuration, as in

```
$ ./bin/ptests.opt -config <special_name> plug-in
```

Then, the directives for this test can be found:

- inside file `tests/test_config_<special_name>`;
- inside file `tests/subdir/test_config_<special_name>` (for each sub-directory `subdir` of `tests`); or
- inside each test file, in a special comment of the form

  ```
  /* run.config_<special_name>
  ... directives ...
  */
  ```

Multiple configurations may share the same set of directives:

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

The following wildcard is also supported, and accepts any configuration:

```/* run.config* .```

All operations for this test configuration should take option `-config` in argument, as in

```
$ ./bin/ptests.opt -update -config <special_name> plug-in
```

In addition, option `-config <special_name>` requires subdirectories `result_<special_name>` and `oracle_<special_name>` to store results and oracle of the specific configuration.
4.5.4 Detailed options

Figure 4.1 details the options of `ptests`.

<table>
<thead>
<tr>
<th>kind</th>
<th>Name</th>
<th>Specification</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toplevel</td>
<td>-add-options</td>
<td>Additional options appended to the normal toplevel command-line</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-add-options-pre</td>
<td>Additional options prepended to the normal toplevel command line</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-byte</td>
<td>Use bytecode toplevel</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>-opt</td>
<td>Use native toplevel</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>-gui</td>
<td>Use GUI instead of console-based toplevel</td>
<td>no</td>
</tr>
<tr>
<td>Behavior</td>
<td>-run</td>
<td>Delete current results; run tests and examine results</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>-examine</td>
<td>Only examine current results; do not run tests</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>-show</td>
<td>Run tests and show results, but do not examine them; implies -byte</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>-update</td>
<td>Take current results as new oracles; do not run tests</td>
<td>no</td>
</tr>
<tr>
<td>Misc.</td>
<td>-exclude suite</td>
<td>Do not consider the given suite</td>
<td>diff -u</td>
</tr>
<tr>
<td></td>
<td>-diff cmd</td>
<td>Use <code>cmd</code> to show differences between results and oracles when examining results</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-cmp cmd</td>
<td>Use <code>cmd</code> to compare results against oracles when examining results</td>
<td>cmp -s</td>
</tr>
<tr>
<td></td>
<td>-use-diff-as-cmp</td>
<td>Use the same command for <code>diff</code> and <code>cmp</code></td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>-j n</td>
<td>Set level of parallelism to <code>n</code></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>-v</td>
<td>Increase verbosity (up to twice)</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>-help</td>
<td>Display helps</td>
<td>no</td>
</tr>
</tbody>
</table>

Figure 4.1: `ptests` options.

The commands provided through the `-diff` and `-cmp` options play two related but distinct roles. `cmp` is always used for each test (in fact it is used twice: one for the standard output and one for the error output). Only its exit code is taken into account by `ptests` and the output of `cmp` is discarded. An exit code of 1 means that the two files have differences. The two files will then be analyzed by `diff`, whose role is to show the differences between the files. An exit code of 0 means that the two files are identical. Thus, they won’t be processed by `diff`. An exit code of 2 indicates an error during the comparison (for instance because the corresponding oracle does not exist). Any other exit code results in a fatal error. It is possible to use the same command for both `cmp` and `diff` with the `-use-diff-as-cmp` option, which will take as `cmp` command the command used for `diff`.

The `-exclude` option can take as argument a whole suite or an individual test. It can be used with any behavior.

The `-gui` option will launch Frama-C’s GUI instead of the console-based toplevel. It can be combined with `-byte` to launch the bytecode GUI. In this mode the default level of parallelism
is reduced to 1.

### 4.5.5 Detailed directives

Figure 4.2 shows all the directives that can be used in the configuration header of a test (or a test suite). Any directive can identify a file using a relative path. The default directory

<table>
<thead>
<tr>
<th>Kind</th>
<th>Name</th>
<th>Specification</th>
<th>default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command</td>
<td>CMD</td>
<td>Program to run</td>
<td>./bin/toplevel.opt</td>
</tr>
<tr>
<td></td>
<td>OPT</td>
<td>Options given to the program</td>
<td>-val -out -input -deps</td>
</tr>
<tr>
<td></td>
<td>STD OPT</td>
<td>Add and remove options from the default set</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>LOG</td>
<td>Add an additional file to compare against an oracle</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>EXECNOW</td>
<td>Run a command before the following commands. When specified in a configuration file, run it only once.</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>EXEC</td>
<td>Similar to EXECNOW, but run it once per testing file.</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>MACRO</td>
<td>Define a new macro</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>FILTER</td>
<td>Command used to filter results</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>MODULE</td>
<td>Register a dynamic module to be built and to be loaded with each subsequent test</td>
<td>None</td>
</tr>
<tr>
<td>Test suite</td>
<td>DONTRUN</td>
<td>Do not execute this test</td>
<td>None</td>
</tr>
</tbody>
</table>
|           | FILEREG | selects the files to test                          | .*/\.(c|i|)
| Miscellaneous | COMMENT | Comment in the configuration                       | None    |
|            | GCC   | Unused (compatibility only)                         | None    |

Figure 4.2: Directives in configuration headers of test files.

considered for . is always the parent directory of directory tests. The DONTRUN directive does not need to have any content, but it is useful to provide an explanation of why the test should not be run (e.g. test of a feature that is currently developed and not fully operational yet). If a test file is explicitly given on the command line of ptests, it is always executed, regardless of the presence of a DONTRUN directive.

As said in Section 4.5.2, these directives can be found in different places:

1. default value of the directive (as specified in Fig. 4.2);
2. inside file tests/test_config;
3. inside file tests/subdir/test_config (for each sub-directory subdir of tests); or
4. inside each test file

As presented in Section 4.5.3, alternative directives for test configuration <special_name> can be found in slightly different places:
CHAPTER 4. ADVANCED PLUG-IN DEVELOPMENT

- default value of the directive (as specified in Fig. 4.2);
- inside file tests/test_config_<special_name>;
- inside file tests/subdir/test_config_<special_name> (for each sub-directory subdir of tests); or
- inside each test file.

For a given test tests/suite/test.c, each existing file in the sequence above is read in order and defines a configuration level (the default configuration level always exists).

- 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. At 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 directive. Several files can be monitored from a single OPT 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.
- If there are several directives OPT in the same configuration level, they correspond to different test cases. The OPT directive(s) of a given configuration level replace(s) 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.

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

options are always given between quotes. An option following a + (resp. #) is added to the end (resp. start) of 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 directives EXECNOW and EXEC is the following.

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

or
4.5. TESTING

EXEC: \([ [ \text{LOG file} \mid \text{BIN file} ] \ldots ] \text{cmd} \)

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 from oracles. Full access path to these files have to be specified only in cmd. All the commands described by directives EXECNOW or EXEC are executed in order and before running any of the other directives. If the execution of one EXECNOW or EXEC directive fails (i.e. has a non-zero return code), the remaining actions are not executed. EXECNOW or EXEC directives from a given level are added to the directives of the following levels.

The distinction between EXECNOW and EXEC only occurs when the command is put in a test configuration file: EXECNOW executes the command only once for the test suite, while EXEC executes it once per test file of the test suite.

- 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 pre-defined macros are listed in section 5.3.1.

- 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 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.

@ 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 guaranteed
order between test executions is the following:

1. EXEC and EXECNOW commands are executed sequentially, from top to bottom.
2. Then, all OPT/STDOPT commands execute in an unspecified order (possibly simulta-
neously).

A consequence of this ordering is that, if you need a test to produce output that will be
consumed by another test, the producer must be defined via EXEC/EXECNOW (e.g. using
@frama-c@ and explicitly giving its arguments), while the consumer can be either in a
later EXEC/EXECNOW, or in a OPT/STDOPT directive.

4.6 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 4.9 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 4.7). Second it allows to define
plug-in specific parameters available as options on the Frama-C command line to the end-user
(see Section 4.12).

4.7 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 4.6 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 `Pervasives.stdout` and `Pervasives.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 4.10** A minimal example of a plug-in using the logging services:

```ocaml
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 () = Db.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.

### 4.7.1 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:

```ocaml
let print_one v l =  
    Format.printf "variable %s (%d):@
    v.vname v.vid;
List.iter  
    (fun (ki, lv) →  
        Format.printf " sid %a: %a@
        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 \texttt{Format.formatter} parameter, and you call \texttt{fprintf} instead of \texttt{printf}:

```ocaml
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 \texttt{printf} towards an appropriate logging routine, with a formatting string containing the necessary "\%t" and "\%a" formatters:

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

### 4.7.2 Log Quick Reference

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

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

- **feedback** <\texttt{options}> "\ldots"
  - 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** <\texttt{options}> "\ldots"
  - To be used for plug-in development messages and internal error diagnosis. You may specify \texttt{~level:n} option to discard too detailed messages in conjunction with the \texttt{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** <\texttt{options}> "\ldots"
  - 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, \textit{etc.} Typical options include \texttt{~current:true} to localize the message on the current source location.

- **error** <\texttt{options}> "\ldots"
  - 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 \textit{not} be used for some not-yet implemented feature, however.

  The \texttt{abort} routine is a variant that raises an exception and thus immediately aborts the computation\textsuperscript{3}. If you use \texttt{error}, execution will continue until the end of current stage or current group of the running phase (see section 4.13).

\textsuperscript{3}The raised exception is not supposed to be caught by anything else than the main entry point of Frama-C.
4.7. LOGGING SERVICES

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) |

4.7.3 Logging Routine Options

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

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. You can define
subcategories by putting colons in the registered name. For instance a:b:c defines a subcat-
egory c of a:b, itself a subcategory of a. 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 de-
 bugging 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 \(~\text{wkey}\) optional argument that takes a value of abstract type \texttt{warn\_category}. Warning categories are distinct from plain categories, and must be registered with the \texttt{register\_warn\_category} function. As explained in the user manual \cite{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 \texttt{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 \texttt{AST} visitor.

\begin{itemize}
\item \texttt{\textasciitilde source:s} use the source location \texttt{s} (see \texttt{Log.mli})
\item \texttt{\textasciitilde current:}\texttt{true} use the current source location managed by \texttt{Cil.CurrentLoc}.
\end{itemize}

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 4.7.4 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:

\begin{itemize}
\item \texttt{\textasciitilde emitwith:f} suppresses the echo and sends the emitted event only to the callback function \texttt{f}. Listeners are not fired at all.
\item \texttt{\textasciitilde once:}\texttt{true} finally discards the message if the same one was already emitted before with the \texttt{\textasciitilde once} option.
\end{itemize}

Append Option.  All logging routines have the \texttt{\textasciitilde append:f} optional parameter, where \texttt{f} is function taking a \texttt{Format.formatter} as parameter and returning unit. This function \texttt{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 \texttt{Format.kfprintf}-like functions.

4.7.4  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 \texttt{\textasciitilde once} has been used. Upon message completion, the message is emitted and sent to all globally registered hook functions, unless the \texttt{\textasciitilde emitwith} option has been used.

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

\begin{verbatim}
val set_echo : ?plugin:string \rightarrow ?kinds:kind list \rightarrow bool \rightarrow unit
val add_listener : ?plugin:string \rightarrow ?kinds:kind list \rightarrow (event \rightarrow unit) \rightarrow unit
\end{verbatim}
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 \( \text{event} \rightarrow \alpha \), and the log routine returns \( \alpha \).

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

```ocaml
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:

\[
\begin{align*}
\text{log} \ ?\text{kind} \ ?\text{verbose} \ ?\text{debug} <\text{options}> "..." \\
\text{with_log} \ f \ ?\text{kind} <\text{options}> "..."
\end{align*}
\]

Emits a message with the given kind, when the verbosity and/or debugging level are sufficient.

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:

- \( \text{log} "..." \) : verbosity is at least 1
- \( \text{log} -\text{verbose}:n \) : verbosity is at least \( n \)
- \( \text{log} -\text{debug}:n \) : debugging is at least \( n \)
- \( \text{log} -\text{verbose}:v -\text{debug}:d \) : 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 Pervasives.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_log_channel channel f ?kind ?prefix

This routine is similar to the with_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 Pervasives.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. In this case, your text is written to the specified file.
4.8 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 4.8.1 introduces type values. Then Section 4.8.2 introduces datatypes built on top of type values.

4.8.1 Type Value

A type value is an OCaml value which dynamically represents a static monomorphic OCaml type $\tau$. It gets the type $\tau$ $\text{Type}.t$. There is at most one type value which represents the type $\tau$. 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 4.9.3).

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

Example 4.11 The type value for type int is Datatype.int while the one for type string is Datatype.string. The former has type int $\text{Type}.t$ while the latter has type string $\text{Type}.t$.

Type values are created when building datatypes (see Section 4.8.2). 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 module Datatype.

Example 4.12 The type value for type int list is Datatype.list Datatype.int while the one for type string $\rightarrow$ char $\rightarrow$ bool is Datatype.func2 Datatype.string Datatype.char Datatype.bool. The former has type int list $\text{Type}.t$ while the latter has type (string $\rightarrow$ char $\rightarrow$ bool) $\text{Type}.t$.

4.8.2 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 or journalizing). Whenever possible, a datatype implements an extensible version of Datatype.S, namely Datatype.S_with_collections. For a type $\tau$, this extended signature additionally provides modules Set, Map and Hashtbl respectively implementing sets over $\tau$, maps and hashtables indexed by elements of $\tau$.

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 4.13 The following line of code pretty prints whether two statements are equal.
(assuming the type of [stmt1] and [stmt2] is Cil_types.stmt)

```plaintext
Format.printf
    (* a formatter previously defined somewhere *)
  "statements %a and %a are %sequal"
Cil_datatypeStmt.pretty stmt1
Cil_datatypeStmt.pretty stmt2
(if Cil_datatypeStmt.equal stmt1 stmt2 then "" else "not ")
```

**Example 4.14**  Module *Datatype.String* implements *Datatype.S_with_collections.*
Thus you can initialize a set of strings in the following way.

```plaintext
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 docu-
menced in the *Frama-C* API. The following example introduces them in a practical way.

**Example 4.15**  Here is how to define in the more precise way the datatype corresponding to
a simple sum type.

```plaintext
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_descr.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 = Pervasives.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 *)
  end)
```
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)
(* printer which must produce a valid OCaml value in a given context. It is used when journalising. *)

let internal_pretty_code prec_caller fmt = function
| A → Type.par prec_caller Type.Basic fmt
  (fun fmt → Format.pp_print_string fmt "A")
| B n → Type.par prec_caller Type.Call fmt
  (fun fmt → Format.fprintf fmt "B %d" n)
(* A good prefix name to use for an OCaml variable of this type. *)

let varname v = "ab" ^ (match v with A → "_a_" | B → "_b_")

end)

Only providing an effective implementation for the values name and reprs is mandatory. For instance, if you know that you never journalize a value of a type t, you can define the function internal_pretty_code equal to the predefined function Datatype.pp_fail. Similarly, if 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 4.16 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.Undef
    let equal (x:t) y = x = y
  end)

One weakness of Datatype.Undef is that it cannot be used in a projectified state (see Section 4.11.2) because its values cannot be serializable. In such a case, you can use the very useful predefined structure Datatype.Serializable_Undef which behaves as Datatype.Undef 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. These functors take as argument how to build the datatype corresponding each monomorphic instance.

Example 4.17 Here is how to apply Datatype.Polymorphic corresponding to the type 'a t below.

```
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 → Pervasives.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
        f
        (fun fmt → Format.Printf fmt "A %a" (f Type.Call) x)
      | B n →
        Type.par prec_caller
        Type.Call
        f
        (fun fmt → Format.Printf fmt "B %d" n)
  let mk_pretty f fmt x =
    mk_internal_pretty_code (fun _ → f) Type.Basic fmt x
  let mk_varname _ = "ab"
  let mk_mem_project mem f = function
    | A x → mem f x
    | B _ → false
end)
module Ab = Poly_AB.Make
```

Polymorphic3 and Polymorphic4 also exist in case of polymorphic types with 3 or 4 type parameters.
4.9. 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 4.9.1 indicates how to register and access a plug-in through a .mli file. Section 4.9.2 indicates how to register and access a kernel-integrated plug-in while Section 4.9.3 details how to register and access a standard plug-in.

4.9.1 Registration through a .mli File

Target readers: plug-in developers.

Prerequisite: Basic knowledge of make.

Each plug-in is compiled into a module of name indicated by the variable PLUGIN_NAME of its Makefile (say Plugin_A). Its developer has to provide a .mli for this plug-in (following the previous example, a file Plugin_A.mli). This .mli file may thus contains the API of the plug-in.

Another plug-in may then access to Plugin_A as it accesses any other OCaml module, but it has to declare in its Makefile that it depends on Plugin_A through the special variable PLUGIN_DEPENDENCIES.

Example 4.18 Plugin_A declares and provides access to a function compute in the following way.

```
File File plugin_a/my_analysis_a.ml
| let compute () = ... |
```

```
File File plugin_a/Plugin_A.mli
| module My_analysis_a: sig val compute: unit → unit |
```

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).
CHAPTER 4. ADVANCED PLUG-IN DEVELOPMENT

PLUGIN_NAME:=Plugin_A
PLUGIN_CMO:=... my_analysis_a ...
...
include Makefile.dynamic

Then, Plugin_B may use this function Compute as follows.

File File plugin_b/my_analysis_b.ml

| let compute () = ... Plugin_A.My_analysis_a.compute () ...

File File plugin_b/Makefile

| PLUGIN_NAME:=Plugin_B
| PLUGIN_CMO:=... my_analysis_b ...
| PLUGIN_DEPENDENCIES:=Plugin_A
| ...
| include Makefile.dynamic

4.9.2 Kernel-integrated Registration and Access

Target readers: kernel-integrated plug-in developers.

Prerequisite: Accepting to modify the Frama-C kernel. Otherwise, you can still register your plug-in as any standard plug-in (see Section 4.9.3 for details).

A database, called Db (in directory src/kernel_services/plugin_entry_points), groups together the API of all kernel-integrated plug-ins. So it permits easy plug-in collaborations. Each kernel-integrated plug-in is only visible through Db. For example, if a plug-in A wants to know the results of another plug-in B, it uses the part of Db corresponding to B. A consequence of this design is that each plug-in has to register in Db by setting a function pointer to the right value in order to be usable from others plug-ins.

Example 4.19 Plug-in Impact registers function compute_pragmas in the following way.

File src/plugins/impact/register.ml

| let compute_pragmas () = ...
| let () = Db.Impact.compute_pragmas ← compute_pragmas

Thus each developer who wants to use this function calls it by pointer dereferencing like this.

| let () = !Db.Impact.compute_pragmas ()

If a kernel-integrated plug-in has to export some datatypes usable by other plug-ins, such datatypes have to be visible from module Db. Thus they cannot be declared in the plug-in implementation itself like any other plug-in declaration because postponed type declarations are not possible in OCaml.

Such datatypes are called plug-in types. The solution is to put these plug-ins types in some files linked before Db; hence you have to put them in another directory than the plug-in directory. The best way is to create a directory dedicated to types.
4.9. PLUG-IN REGISTRATION AND ACCESS

Recommendation 4.2 The suggested name for this directory is \texttt{p\_types} for a plug-in \texttt{p}.

If you add such a directory, you also have to modify \texttt{Makefile} by extending variable \texttt{FRAMAC\_SRC\_DIRS} (see Section 5.2.3).

Example 4.20 Suppose you are writing a plug-in \texttt{p} which exports a specific type \texttt{t} corresponding to the result of the plug-in analysis. The standard way to proceed is the following.

File \texttt{src/plugins/p\_types/p\_types.mli}

\begin{verbatim}
type \texttt{t} = ...
\end{verbatim}

File \texttt{src/kernel\_services/plugin\_entry\_points/db.mli}

\begin{verbatim}
module \texttt{P} : sig
  val \texttt{run\_and\_get} : (unit \rightarrow \texttt{P\_types.t}) ref
    (** Run plugin analysis (if it was never launched before).
     \texttt{@return} result of the analysis. *)
end
\end{verbatim}

File \texttt{share/Makefile.common}

\begin{verbatim}
FRAMAC\_SRC\_DIRS= ... plugins/p\_types
  # Extend this variable with the new directory
\end{verbatim}

This design choice has a side effect: it reveals exported types. You can always hide them using a module to encapsulate the types (and provide corresponding getters and setters to access them).

At this point, part of the plug-in code is outside the plug-in implementation. This code should be linked before \texttt{Db} \footnote{A direct consequence is that you cannot use the whole \texttt{Frama-C} functionalities, such as module \texttt{Db}, inside this code.}.

To this effect, the files containing the external plug-in code must be added to the \texttt{Makefile} variable \texttt{PLUGIN\_TYPES\_CMO} (see Section 5.2.3).

4.9.3 Dynamic Registration and Access

Target readers: standard plug-ins developers.

Registration of kernel-integrated plug-ins requires to modify module \texttt{Db} which belongs to the \texttt{Frama-C} kernel. Such a modification is not possible for standard plug-ins which are fully independent of \texttt{Frama-C}. Consequently, the \texttt{Frama-C} kernel provides another way for registering a plug-in through the module \texttt{Dynamic}.

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

Registering a value

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

```
val register : plugin : string \to string \to \alpha \ Type.t \to journalize : bool \to \alpha \to
```

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 4.8.1). It is required for safety reasons when accessing to the registered value (see the next paragraph). The labeled fourth argument `journalize` indicates whether a total call to this function must be written in the journal (see also Section 4.10). The usual value for this argument is `true`. The fifth argument is the value to register.

**Example 4.21** Here is how the function `run` of the plug-in `hello` of the tutorial is registered. The type of this function is `unit \to unit`.

```
let run () : unit = ...
let () =
Dynamic.register
~plugin:"Hello"
"run"
(Datatype.func Datatype.unit Datatype.unit)
~journalize : true
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 4.13).

The function call `Datatype.func Datatype.unit Datatype.unit` returns the type value representing `unit \to 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 \to unit`.

Accessing to a registered value

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

```
val get : plugin : string \to string \to \alpha \ Type.t \to \alpha
```

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 4.22** 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 4.23 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.

```ocaml
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.t
let stmt_ty = Cil_datatype.Stmt.t

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" ~journalize:false name ty
```
You have to use the functor \texttt{Type.Abstract} to access to the type value corresponding to the type of callstacks (and thus to access to the above dynamically registered functions).

\begin{verbatim}
(* 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 ... *)
\end{verbatim}

4.10 Journalization

\textit{Not written yet: please report as “feature request” on \url{http://bts.frama-c.com} if you really need this section.}

4.11 Project Management System

\textbf{Prerequisite:} \textit{Knowledge of the OCaml module system and labels.}

In Frama-C, a key notion detailed in this section is the one of \textit{project}. An overview as well as technical details may also be found in a related article in French [17]. Section 4.11.1 first introduces the general principle of project. Section 4.11.2 introduces the notion of \textit{states}. State registration is detailed in Sections 4.11.3 and 4.11.4. The former is dedicated to
4.11. PROJECT MANAGEMENT SYSTEM

standard (high-level) registration, while the latter is dedicated to low-level registration. Then
Section 4.11.5 explains how to use projects. Finally Section 4.11.6 details state selections.

4.11.1 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 4.12) 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 4.11.3 and 4.11.4.
Relations between states and projects are summarized in Figure 4.3.

<table>
<thead>
<tr>
<th>States</th>
<th>Projects</th>
<th>Project (p_1)</th>
<th>(\ldots)</th>
<th>Project (p_n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AST (a)</td>
<td>value of (a) in (p_1)</td>
<td>(\ldots)</td>
<td>value of (a) in (p_n)</td>
<td></td>
</tr>
<tr>
<td>data (d_1)</td>
<td>value of (d_1) in (p_1)</td>
<td>(\ldots)</td>
<td>value of (d_1) in (p_n)</td>
<td></td>
</tr>
<tr>
<td>(\ldots)</td>
<td>(\ldots)</td>
<td>(\ldots)</td>
<td>(\ldots)</td>
<td></td>
</tr>
<tr>
<td>data (d_m)</td>
<td>value of (d_m) in (p_1)</td>
<td>(\ldots)</td>
<td>value of (d_m) in (p_n)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.3: Representation of the Frama-C State.

To ease development, Frama-C maintains a current project (Project.current ()): all op-
erations 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 4.11.5.

4.11.2 State: Principle

If some data should be part of the state of Frama-C, you must register it in the project
library (see Sections 4.11.3 and 4.11.4).

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 4.4 summarizes these interactions between the project library and your state.
Figure 4.4: 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 4.11.6). 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 \texttt{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.

\textbf{Example 4.24}

\begin{verbatim}
!Db.Value.compute();
Kernel.feedback "%B" (Db.Value.is_computed()); (* true *)
Globals.set_entry_point "f" true;
Kernel.feedback "%B" (Db.Value.is_computed()); (* false *)
\end{verbatim}

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

\begin{verbatim}
[kernel] true
[kernel] false
\end{verbatim}

\textbf{State Registration: Overview}

For registering a new state, functor \texttt{State-builder.Register} is provided. Its use is described in Section 4.11.4 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). They are described in Section 4.11.3.

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).

### 4.11.3 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 4.11.4 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.

```ocaml
classic 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 () = ... !Db.Value.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.

```ocaml
module State =
  Cil_state_builder.Stmt_hashtbl
  (Datatype.Pair(Kernel_function)(Cil_datatype.Varinfo))
  (struct
    let size = 97
    let name = "state"
    let dependencies = [ Db.Value.self ]
  end)
let compute_info (kf,vi) = ...
let memoize = State.memo compute_info
let run () = ... !Db.Value.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?

---

6 These datastructures are only mutable datastructures (like hashtables, arrays and references) because global states are always mutable.
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 4.8.2). 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 4.8.1).

2. From outside, a state actually hides its internal representation in order to ensure some invariants: operations on states implementing hashables do not take hashables 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 4.13 for advanced explanation about the way Frama-C is initialized).

**Example 4.25** Plug-in from puts a reference to its state kind in the following way. This reference is initialized at module initialization time.

```ocaml
File src/kernel_services/plugin_entry_points/db.mli
module From = struct
  ...
  val self : State.t ref
end

File src/kernel_services/plugin_entry_points/db.ml
module From = struct
  ...
  val self = ref State.dummy (* postponed *)
end

File src/plugins/from/functionwise.ml
```
4.11. PROJECT MANAGEMENT SYSTEM

```ocaml
module Tbl = Kernel_function.Make_Table (Function_Froms)
(struct
  let name = "functionwise_from"
  let size = 97
  let dependencies = [ Db.Value.self ]
)'

let () =
  (* performed at module initialization runtime. *)
Db.From.self ← Tbl.self
```

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

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

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

let () =
Cmdline.run_after_extended_stage (fun () →
  State_dependency_graph.add_codependencies ~onto:Tbl.self
  [ !Db.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.

4.11.4 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 4.11.3).

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 4.4). 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:\(^7\)

\[
\text{create}() \text{ returns a fresh value (4.1)} \\
\forall p \text{ of type } t, \text{ create}() = (\text{clear } p; \text{ set } p; \text{ get }()) (4.2) \\
\forall p \text{ of type } t, \text{ copy } p \text{ returns a fresh value (4.3)} \\
\forall p_1, p_2 \text{ of type } t \text{ such that } p_1 \neq p_2, (\text{set } p_1; \text{ get }()) \neq p_2 (4.4)
\]

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

```ocaml
module Ref = struct
  type data = Data.t
  let create () = ref Info.default
  let state = ref (create ())
end
```

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.

```ocaml
include Register
(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
)```

\(^7\)As usual in OCaml, $=$ stands for structural equality while $==$ (resp. $\neq$) stands for physical equality (resp. disequality).
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
```

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.

### 4.11.5 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 4.17) 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 4.11.6):

```
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 4.27** Suppose that you saved the current project into file `foo.sav` in a previous Frama-C session 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" (Db.Value.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;
  !Db.Value.compute ();
  print_computed (); (* true *)
```

---

*A session is one execution of Frama-C (through frama-c or frama-c-gui).*
CHAPTER 4. ADVANCED PLUG-IN DEVELOPMENT

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.

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 4.28** The following code is equivalent to the one given in Example 4.27.

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

It displays
| false |
| true |
| false |

4.11.6 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 4.29** The following statement clears all the results of the value analysis and all its dependencies in the current project.

```plaintext
let selection = State_selection.with_dependencies Db.Value.self in
Project.clear ~selection ()
```
4.12. Command Line Options

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 4.30** 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 4.12).

```ocaml
| 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.

---

4.12 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.

4.12.1 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 4.18) and the GUI. The second one provides a high-level API: each parameter is indeed a state (see Section 4.11.2). 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 4.31** 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.

```ocaml
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

```ocaml
module Print: Parameter_sig.Int
```

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

```ocaml
module Pragma = Kernel_function_set
  (struct
    let option_name = "-impact-pragma"
    let arg_name = "f1, ..., fn"
    let help = "use the impact pragmas 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:

```ocaml
module Pragma: Parameter_sig.Kernel_function_set
```

**Recommendation 4.3** 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 4.32** To know whether Frama-C uses unicode, just write

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

Inside the plug-in From, just write

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

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

Using a parameter of a plug-in $p$ in another plug-in $p'$ requires the use of module
\texttt{Dynamic.Parameter}: since the module defining the parameter is not visible from the out-
side 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 4.9.3). The module \texttt{Dynamic.Parameter} defines
sub-modules which provide easy access to parameters according to their \texttt{OCaml} types.

\begin{example}
\exampleheader{Example 4.33}{Outside the plug-in From, just write}
\begin{verbatim}
| Dynamic.Parameter.Bool.get "-calldeps" ()
\end{verbatim}
in order to know whether callsite-wise dependencies have been required.
\end{example}

4.12.2 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.

\begin{example}
\exampleheader{Example 4.34}{Here is how the option "-slicing-level" restricts the range of its argument to the interval $[0; 3]$.}
\begin{verbatim}
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
\end{verbatim}
\end{example}

Functions which can be applied before applying the functor are defined in the module
\texttt{Parameter_customize}.

\begin{example}
\exampleheader{Example 4.35}{Here is how the opposite of option "-safe-arrays" is renamed into "-unsafe-
arrays" (otherwise, by default, it would be "-no-safe-arrays").}
\begin{verbatim}
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)
\end{verbatim}
\end{example}

4.13 Initialization Steps

\textbf{Prerequisite:} Knowledge of linking of \texttt{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. For instance, journal replays should be performed after loading dynamic plug-ins, and so on. 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 Db.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. 41) 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 checking if journalization has to be activated (cf. Section 4.10), or setting the global verbosity and debugging levels.

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

(a) first, the journal name is set to its right value (according to the option -journal-name) and the default project is created;
(b) then, the parsing of command line options registered for the Cmdline.Early stage;
(c) finally, 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, of journal, 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-script and -add-path;
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, journal 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 4.11.3).

   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 makes 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 or computed by running a journal (see Section 4.10). 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 an 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 an 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 `Db.Main.extend`. For all intents and purposes, you should consider that this stage is the one where these hooks are executed.

### 4.14 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 pre-processing 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.
4.15 Customizing the machine model

**Prerequisite:** None.

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.

To create a new machine model, define an instance of `Cil_types.mach`. You can base it on the examples available in `tests/misc/custom_machdep/custom_machdep.ml` and `src/kernel_internals/runtime/machdeps.ml`. The new definition can be added to Frama-C’s database using `File.new_machdep`.

**Example 4.36** A custom machine description may be implemented as follows (the meaning of each field is presented later in this section):

```ocaml
let my_machine =
{
    version = "generic C compiler for my machine";
    compiler = "generic";
    cpp_arch_flags = []; 
    sizeof_short = 2;
    sizeof_int = 4;
    sizeof_long = 4;
    sizeof_longlong = 8;
    sizeof_ptr = 4;
    sizeof_float = 4;
    sizeof_double = 8;
    sizeof_longdouble = 12;
    sizeof_void = 1;
    sizeof_fun = 1;
    size_t = "unsigned long";
    wchar_t = "int";
    ptdiff_t = "int";
    alignof_short = 2;
    alignof_int = 4;
    alignof_long = 4;
    alignof_longlong = 4;
    alignof_ptr = 4;
    alignof_float = 4;
    alignof_double = 4;
    alignof_longdouble = 4;
    alignof_str = 1;
    char_is_unsigned = false;
    const_string_literals = true;
    little_endian = true;
    underscore_name = false;
    __thread_is_keyword = true;
    __thread___is_keyword = true;
}
let () = File.new_machdep "my_machine" my_machine
```
After this code is loaded, Frama-C can be instructed to use the new machine model using the -machdep command line option.

If you intend to use Frama-C’s standard library headers, you must also do the following:

- define constant __FC_MACHDEP_<CUSTOM>, replacing <CUSTOM> with the name (in uppercase letters) of your created machdep; this can be done via -cpp-extra-args="-D__FC_MACHDEP_<CUSTOM>";
- provide a header file with macro definitions corresponding to your OCaml definitions. For the most part, these are macros prefixed by __FC__, corresponding to standard C macro definitions, e.g., __FC_UCHAR_MAX. These definitions are used by Frama-C's <limits.h> and other headers to provide the standard C definitions. The test file tests/misc/custom_machdep/__fc_machdep_custom.h (reproduced below) contains a complete example of the required definitions. Other examples can be found in share/libc/__fc_machdep.h.

Make sure that your custom header defines the __FC_MACHDEP include guard, and that the program you are analyzing includes this header before all other headers. One way to ensure this without having to modify any source files is to use an option such as -include in GCC.

Example 4.37 Contents of tests/misc/custom_machdep/__fc_machdep_custom.h, used as example for creating custom machdeps. Notice the unusual size for int (3 bytes), selected for testing purposes only, and inconsistent with the chosen values for INT_MIN and INT_MAX, which do not fit in 3 bytes.

```c
/* skeleton of a real custom machdep header. 
   Note: the values provided here are merely for illustrative purposes
   and are not necessarily consistent between them. */
#ifndef __FC_MACHDEP
#define __FC_MACHDEP

#ifdef __FC_MACHDEP_CUSTOM

/* Constants required by the C standard */
#undef __CHAR_UNSIGNED_
#define __WORDSIZED 32
#define __SIZEOF_SHORT 2
#define __SIZEOF_INT 3
#define __SIZEOF_LONG 4
#define __SIZEOF_LONGLONG 8
#define __CHAR_BIT 8
#define __PTRDIFF_T int
#define __SIZE_T unsigned int

#define __FC_SCHAR_MIN (-128)
#define __FC_SCHAR_MAX 127
#define __FC_UCHAR_MAX 255
#define __FC_SHRT_MIN (-32768)
#define __FC_SHRT_MAX 32767
#define __FC_USHRT_MAX 65535
#define __FC_INT_MIN (-2147483647 - 1)
#define __FC_INT_MAX 2147483647
#define __FC_UINT_MAX 4294967295U
```

92
#define __FC_LONG_MIN (-2147483647L -1L)
#define __FC_LONG_MAX 2147483647L
#define __FC_ULONG_MAX 4294967295UL
#define __FC_LLONG_MIN (-9223372036854775807LL -1LL)
#define __FC_LLONG_MAX 9223372036854775807LL
#define __FC_UULLONG_MAX 18446744073709551615ULL
#define __INT_MAX_T signed long long
#define __UINT_MAX_T unsigned long long
#define __FC_PATH_MAX 256
#define __FC_SIZE_MAX __FC_UULLONG_MAX

/* Optional constants */
#define __INT8_T signed char
#define __UINT8_T unsigned char
#define __INT16_T signed short
#define __UINT16_T unsigned short
#define __INT32_T signed int
#define __UINT32_T unsigned int
#define __INT64_T signed long long
#define __UINT64_T unsigned long long

/* Required constants */
#define __INT_LEAST8_T signed char
#define __UINT_LEAST8_T unsigned char
#define __INT_LEAST16_T signed short
#define __UINT_LEAST16_T unsigned short
#define __INT_LEAST32_T signed int
#define __UINT_LEAST32_T unsigned int
#define __INT_LEAST64_T signed long long
#define __UINT_LEAST64_T unsigned long long
#define __INT_FAST8_T signed char
#define __UINT_FAST8_T unsigned char
#define __INT_FAST16_T signed int
#define __UINT_FAST16_T unsigned int
#define __INT_FAST32_T signed long
#define __UINT_FAST32_T unsigned int
#define __INT_FAST64_T signed long long
#define __UINT_FAST64_T unsigned long long

/* POSIX */
#define __SSIZE_T int
/* stdio.h */
#define __FC_L_tmpnam 1024
/* stdint.h */
#define __FC_PTRDIFF_MIN __FC_INT_MIN
#define __FC_PTRDIFF_MAX __FC_INT_MAX
#define __FC_INTMAX_MIN (-9223372036854775807LL -1LL)
define __FC_INTMAX_MAX 9223372036854775807LL
#define __FC_UINTMAX_MAX 18446744073709551615ULL
#define __FC_EOF (-1)
#define __FC_FOPEN_MAX 20
#define __FC_RAND_MAX 32767
#define __WCHAR_T unsigned short

/* for stdarg.h */
#define __FC_VA_LIST_T char *

/* for time.h */
#define __FC_TIME_T long

/* for wchar.h */
#define __WINT_T unsigned int
#define __FC_WEOF (0xFFFFFFFFU)
#define __FC_WINT_MIN 0
#define __FC_WINT_MAX __FC_UINT_MAX

/* for errno.h */
#define __FC_EPERM 1
#define __FC_ENOENT 2
#define __FC_ESRCH 3
#define __FC_EINTR 4
#define __FC_EIO 5
#define __FC_ENXIO 6
#define __FC_E2BIG 7
#define __FC_ENOEXEC 8
#define __FC_EBADF 9
#define __FC_ECHILD 10
#define __FC_EAGAIN 11
#define __FC_ENOMEM 12
#define __FC_EACCES 13
#define __FCEFAULT 14
#define __FC_ENOTBLK 15
#define __FC_EBUSY 16
#define __FC_EEXIST 17
#define __FC_EXDEV 18
#define __FC_ENODEV 19
#define __FC_ENOTDIR 20
#define __FC_EISDIR 21
#define __FC EINVAL 22
#define __FC_ENFILE 23
#define __FC_EMFILE 24
#define __FC_ENOTTY 25
#define __FC_ETXTBSY 26
#define __FC_EFBIG 27
#define __FC_ENOSPC 28
#define __FC_ESPIPE 29
#define __FC_EROFS 30
#define __FC_EMLINK 31
#define __FC_EPIPE 32
#define __FC_EDOM 33
#define __FC_ERANGE 34
#define __FC_EDALEADLK 35
#define __FC_ENAMETOOLONG 36
#define __FC_ENOLCK 37
#define __FC_ENOSYS 38
#define __FC_ENOTEMPTY 39
#define __FC_ELOOP 40
#define __FC_EWOULDBLOCK EAGAIN
#define __FC_ENOMSG 42
#define __FC_EIDRM 43
#define __FC_ECHRNG 44
#define __FC_EL2NSYNC 45
#define __FC_EL3HLT 46
#define __FC_EL3RST 47
#define __FC_ELNRNG 48
#define __FC_EUNATCH 49
#define __FC_ENOSCI 50
#define __FC_EL2HLT 51
#define __FC_EBADE 52
#define __FC_EBADR 53
#define __FC_EXFULL 54
#define __FC_ENGANO 55
#define __FC_EBADRQC 56
#define __FC_EBADSLT 57
#define __FC_EDEADLOCK EDEADLK
#define __FC_EBFONT 59
#define __FC_ENOSTR 60
#define __FC_ENODATA 61
#define __FC_ETIME 62
#define __FC_ENOSR 63
#define __FC_ENOPKG 64
#define __FC_EILSEQ 65
#define __FC_EREMOTE 66
#define __FC_ENOSEQ 67
#define __FC_EADV 68
#define __FC_ESRMNT 69
#define __FC_ECOMM 70
#define __FC_EPROTO 71
#define __FC_EMULTIHOP 72
#define __FC_EDOTDOT 73
#define __FC_EBADMSG 74
#define __FC_EOVERFLOW 75
#define __FC_ENOTUNIQ 76
#define __FC_EBADFD 77
#define __FC_EREMCHG 78
#define __FC_ELIBACC 79
#define __FC_ELIBBAD 80
#define __FC_ELIBSCN 81
#define __FC_ELIBMAX 82
#define __FC_ELIBEXEC 83
#define __FC_EILSEQ 84
#define __FC_ERESTART 85
#define __FC_ESTRPIPE 86
#define __FC_EUSER 87
#define __FC_ENOTSOCK 88
#define __FC_EDESTADDRREQ 89
#define __FC_EMSGSIZE 90
#define __FC_EPROMPTYPE 91
#define __FC_ENOPROTOOPT 92
CHAPTER 4. ADVANCED PLUG-IN DEVELOPMENT

#define __FC_EPROTONOSUPPORT 93
#define __FC_ESOCKTNOSUPPORT 94
#define __FC_ENOTSUP 95
#define __FC_EQPNOTSUPP 96
#define __FC_EPFNOSUPPORT 97
#define __FC_EDADDINUE 98
#define __FC_EDADDRNOTAVAIL 99
#define __FC_ENETDOWN 100
#define __FC_ENETUNREACH 101
#define __FC_ENETRESET 102
#define __FC_ECONNABORTED 103
#define __FC_ECONNRESET 104
#define __FC_ENOBUFS 105
#define __FC_EISCONN 106
#define __FC_ENOTCONN 107
#define __FC_ESHUTDOWN 108
#define __FC_EISNAM 109
#define __FC_ENOTNAM 110
#define __FC_EUCLEAN 111
#define __FC_EINPROGRESS 112
#define __FC_ETIMEDOUT 113
#define __FC_ECONNREFUSED 114
#define __FC_EHOSTDOWN 115
#define __FC_EHOSTUNREACH 116
#define __FC_EALREADY 117
#define __FC_EINPROGRESS 118
#define __FC_ESTALE 119
#define __FC_EISNAM 120
#define __FC_EISNAM 121
#define __FC_EDQUOT 122
#define __FC_ENOMEDIUM 123
#define __FC_EMEDIUMTYPE 124
#define __FC_ECANCELED 125
#define __FC_ENOKEY 126
#define __FC_EKEYEXPIRED 127
#define __FC_EKEYREVOKED 128
#define __FC_EKEYREJECTED 129
#define __FC_EOWNERDEAD 130
#define __FC_ENOTRECOVERABLE 131
#define __FC_ERFKILL 132
#define __FC_EHWPOISON 133

#else
error "I'm supposed to be called with __FC_MACHDEP_CUSTOM macro defined"
#endif
#endif

An example of the complete command-line is presented below, for a custom machdep called myarch, defined in file my_machdep.ml and with stdlib constants defined in machdep_myarch.h:

```
frama-c -load-script my_machdep.ml -machdep myarch \
    -cpp-extra-args="-D__FC_MACHDEP_MYARCH -include machdep_myarch.h"
```
4.16 Machdep record fields

Each field in the machdep record is succinctly described in the Cil_types module. We present below a thorough description of each field.

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

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

- **msvc**: enables Cil.msvcMode, that is, MSVC (Visual Studio)-specific extensions;
- **gcc**: enables Cil.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 to be added to the command-line when invoking the C preprocessor. Typically used to ensure that multiarch compilers apply the appropriate predefined macros. E.g. use `"-m32"` for a 32-bit machdep when preprocessing with a 64-bit multiarch GCC. Note that, in practice, very few programs rely on such predefined macros, such as `__x86_64` and `__i386`.

**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 0 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).

**size_t**: a string containing the actual type that size_t expands to, e.g. "unsigned long".

---

9Note that the sizes of standard integer types are already defined in the machdep, so they do not depend on these flags.
\begin{verbatim}
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.

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_longsdouble : 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.

char_is_unsigned : whether type char is unsigned.

const_string_literals : whether string literals have const chars, or are writable.
If true, the following code has undefined behavior, otherwise it is defined:
char *s = "no"; s[0] = 'g';

little_endian : whether the machine is little endian.

underscore_name : whether the compiler generates assembly labels by prepending _ to the identifier. That is, will function foo() have the label foo, or _foo?

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).

Writing a new machdep Writing a machdep for a new architecture is not trivial, due to the fact that some steps are hard to automate. If you have a working compiler for the target architecture, you can use it to produce an executable that will print the contents of expressions such as sizeof(long), _Alignof(int), etc. You can also use the compiler to test for unsupported features. In case printf is not available, you can use the exit code of the program (return code of main). In case you can only preprocess, but not compile and run the program, the assembly code may provide some useful data.
\end{verbatim}
4.17 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 4.17.6, which the reader already familiar with Cil is invited to read carefully.

4.17.1 Entry Points

Module Cil offers various entry points for the visitor. They are functions called Cil.visitCilAstType 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.

4.17.2 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 vastType, 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.

10This naming convention is not strictly enforced. For instance the method corresponding to offset is voffs.
4.17.3 Action Performed

The return value of visiting methods indicates what should be done next. There are six possibilities:

- **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.

4.17.4 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).

4.17.5 In-place and Copy Visitors

The visitors take as argument a **visitor_behavior**, which comes in two flavors: **inplace_visit** and **copy_visit**. 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 functions:

- **reset_behavior_name** resets the mapping corresponding to the type name.
- **get_original_name** gets the original value corresponding to a copy (and behaves as the identity if the given value is not known).
- **get_name** gets the copy corresponding to an old value. If the given value is not known, it behaves as the identity.
- **set_name** 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.

**get_original_name** functions 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_original_name to give correct results.

The list of such indices is given Figure 4.5.

<table>
<thead>
<tr>
<th>Type</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>varinfo</td>
<td>vid</td>
</tr>
<tr>
<td>compinfo</td>
<td>ckey</td>
</tr>
<tr>
<td>enuminfo</td>
<td>ename</td>
</tr>
<tr>
<td>typeinfo</td>
<td>tname</td>
</tr>
<tr>
<td>stmt</td>
<td>sid</td>
</tr>
<tr>
<td>logic_info</td>
<td>l_var_info.lv_id</td>
</tr>
<tr>
<td>logic_var</td>
<td>lv_id</td>
</tr>
<tr>
<td>fieldinfo</td>
<td>fname and fcomp.ckey</td>
</tr>
</tbody>
</table>

**Figure 4.5: 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).

### 4.17.6 Differences Between the CIL and Frama-C Visitors

As said in Section 4.17.2, 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 4.17.3), 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.

### 4.17.7 Example

Here is a small copy visitor that adds an assertion for each division in the program, stating that the divisor is not zero:

```ocaml
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, _) →
    let logic_denom = Logic_utils.expr_to_term ~cast:true 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 = Extlib.the 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}
```
4.18 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 4.11.5). 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 and Db.Properties contain several operations over annotations.

4.19 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 an 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. It is not possible to have the same keyword for two distinct extensions, especially if they belong to different categories, as this would lead to ambiguities in the parser.

An acsl_extension is a triple (id, kw, ext) where id is its unique ID, used in annotation tables and generated by Logic_const.new_acsl_extension, kw identifies the extension, and ext is an acsl_extension_kind and 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.
- Ext_terms terms with terms a possibly empty list of terms.

For the latter two cases, the corresponding list is traversed normally by the visitor (see section 4.17).

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.

- Logic_typing.register_behavior_extension
- Logic_typing.register_global_extension
- Logic_typing.register_code_annot_extension
- Logic_typing.register_code_annot_next_stmt_extension
- Logic_typing.register_code_annot_next_loop_extension
- Logic_typing.register_code_annot_next_both_extension
4.19. EXTENDING AC admirable ANNOTATIONS

Each function takes three arguments:

- **kw** the name of the extension,
- **status**, a boolean flag indicating whether the extended annotation may have a validity status, and
- **f** the type-checking function itself.

After a call to the appropriate registration function, 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 appear either alone in a global annotation, or as part of an axiomatic with a set of other global annotations.

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).

During type-checking, the list `[e1;...;en]` will be given to `f`, 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). `f` 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 `f` 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`.

```plaintext
open Logic_ptree
open Cil_types
open Logic_typing

let type_foo ~typing_context ~loc:_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
```
With this extension enabled, Frama-C will interpret the following clause in a given source file:

\[ /\star \text{foo } 84 \equiv \text{\textbackslash foo } + \text{\textbackslash foo};\ast / \]

as the following type-checked AST fragment:

\[ /\star \text{foo } 84 \equiv 42 + 42;\ast / \]

If all the information of the extended clause is contained in the predicate list \texttt{preds}, no other registration is needed beyond the parsing and type-checking: the pretty-printer will output \texttt{preds} as a comma-separated list preceded by \texttt{kw} (or \texttt{loop kw} if the extension is a loop annotation), and the visitor will traverse each \texttt{preds} as well as any predicate present in the AST. However, if some information is present in the internal state of the plugin, two more functions may be required for pretty-printing and visiting the extending clause respectively.

First, \texttt{Cil_printer.register_behavior_extension} registers a new pretty-printer \texttt{pp} for a given extension \texttt{kw}. Together with the \texttt{acsl_extension_kind} of the extended clause, \texttt{pp} is given the current pretty-printer and the formatter where to output the result.

Second, \texttt{Cil.register_behavior_extension} registers a custom visitor \texttt{vext} for a given extension \texttt{kw}. \texttt{vext} is given the content of the extended clause and the current visitor, and must return a \texttt{Cil.visitAction} (if all the information is in the plugin’s own table, it can return \texttt{SkipChildren}).

### 4.20 Locations

**Prerequisite:** None.

In Frama-C, different representations of C locations exist. Section 4.20.1 presents them. Moreover, maps indexed by locations are also provided. Section 4.20.2 introduces them.

#### 4.20.1 Representations

There are four different representations of C locations. Actually only three are really relevant. All of them are defined in module \texttt{Locations}. They are introduced below. See the documentation of \texttt{src/kernel_services/abstract_interp/locations.mli} for details about the provided operations on these types.

- Type \texttt{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 \texttt{location}, equivalently \texttt{Location.t} is used to represent the right part of a C affectation (including bitfields). It is represented by a \texttt{Location_BITS.t} (see below) attached to a size. It is possible to join two locations \textit{if and only if they have the same sizes}. 
4.21 GUI EXTENSION

- 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 4.4** 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.

4.20.2 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).

4.21 GUI Extension

**Prerequisite:** Knowledge of Lablgtk2.

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 4.22).

The GUI plug-in code has to be put in separate files into the plug-in directory. Furthermore, in the Makefile, the variable PLUGIN_GUI_CMO has to be set in order to compile the GUI plug-in code (see Section 5.2.3).

Besides time-consuming computations have to call the function !Db.progress from time to time in order to keep the GUI reactive.

The GUI implementation uses Lablgtk2 [10]: you can use any Lablgtk2-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/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.

## 4.22 Documentation

**Prerequisite:** Knowledge of ocamldoc.

Here we present some hints on the way to document your plug-in. First Section 4.22.1 introduces a quick general overview about the documentation process. Next Section 4.22.2 focuses on the plug-in source documentation.

### 4.22.1 General Overview

Command `make doc` produces the whole Frama-C source documentation in HTML format. The generated index file is `doc/code/html/index.html`. A more general purpose index is `doc/index.html` (from which the previous index is accessible).

The previous command takes times. So command `make doc-kernel` only generates the kernel documentation (i.e. Frama-C without any plug-in) while `make $(PLUGIN_NAME)_DOC` (by substituting the right value for `$(PLUGIN_NAME)`) generates the documentation for a single plug-in.

### 4.22.2 Source Documentation

Each plug-in should be properly documented. Frama-C uses ocamldoc and so you can write any valid ocamldoc comments.

**ocamldoc tags for Frama-C**  The tag `@since version` should document any element introduced after the very first release, in order to easily know the required version of the Frama-C kernel or specific plug-ins. In the same way, the Frama-C documentation generator provides a custom tag `@modify version description` which should be used to document any element which semantics have changed since its introduction.

Furthermore, the special tag `@plugin developer guide` must be attached to each function used in this document.

**Plug-in API**  A plug-in should export functions in its plug-in interface or through modules Db or Dynamic as explained in Section 4.9.

The interface name of a plug-in plugin must be `Plugin.mli`. Be careful to capitalization of the filename which is unusual in OCaml but required here for compilation purposes.
Internal Documentation for Kernel Integrated Plug-ins  The Frama-C documentation generator also produces an internal plug-in documentation which may be useful for the plug-in developer itself. This internal documentation is available via file doc/code/plugin/index.html for each plug-in plugin. You can add an introduction to this documentation into a file. This file has to be assigned into variable PLUGIN_INTRO of the Makefile (see Section 5.2.3).

In order to ease access to this internal documentation, you have to manually edit the file doc/index.html in order to add an entry for your plug-in in the plug-in list.

Internal Documentation for External Plug-ins  External plug-ins can be documented in the same way as plug-ins that are compiled together with Frama-C. However, in order to be able to compile the documentation with make doc, you must have generated the documentation of Frama-C’s kernel (make doc, see above) and installed it with the make install-doc-code command.
Chapter 5

Reference Manual

**Target readers:** *Developers who would like to have a deep understanding of Frama-C.*

This chapter is a reference manual for Frama-C developers. It provides details completing the previous chapters.

### 5.1 Configure.in

Figure 5.1 presents the different parts of configure.in in the order in which they are introduced in the file. The second column of the table indicates whether the given part might need to be modified by a kernel-integrated plug-in developer. More details are provided below.

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Mod.</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Configuration of make</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Configuration of OCaml</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Configuration of mandatory tools/libraries</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Configuration of non-mandatory tools/libraries</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Platform configuration</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Wished Frama-C plug-in</td>
<td>yes</td>
<td>Sections 4.1.2 and 4.1.4</td>
</tr>
<tr>
<td>7</td>
<td>Configuration of plug-in tools/libraries</td>
<td>yes</td>
<td>Section 4.1.3</td>
</tr>
<tr>
<td>8</td>
<td>Checking plug-in dependencies</td>
<td>yes</td>
<td>Section 4.1.5</td>
</tr>
<tr>
<td>9</td>
<td>Makefile creation</td>
<td>yes</td>
<td>Section 4.1.2</td>
</tr>
<tr>
<td>10</td>
<td>Summary</td>
<td>yes</td>
<td>Section 4.1.2</td>
</tr>
</tbody>
</table>

Figure 5.1: Sections of configure.in.

1. **Configuration of make** checks whether the version of make is correct. Some useful option is `--enable-verbosemake` (resp. `--disable-verbosemake`) which set (resp. unset) the verbose mode for make. In verbose mode, full make commands are displayed on the user console: it is useful for debugging the makefile. In non-verbose mode, only command shortcuts are displayed for user readability.

2. **Configuration of OCaml** checks whether the version of OCaml is correct.

3. **Configuration of other mandatory tools/libraries** checks whether all the external mandatory tools and libraries required by the Frama-C kernel are present.
4. **Configuration of other non-mandatory tools/libraries** checks which external non-mandatory tools and libraries used by the Frama-C kernel are present.

5. **Platform Configuration** sets the necessary platform characteristics (operating system, specific features of gcc, etc) for compiling Frama-C.

6. **Wished Frama-C Plug-ins** sets which Frama-C plug-ins the user wants to compile.

7. **Configuration of plug-in tools/libraries** checks the availability of external tools and libraries required by plug-ins for compilation and execution.

8. **Checking Plug-in Dependencies** sets which plug-ins have to be disabled (at least partially) because they depend on others plug-ins which are not available (at least partially).

9. **Makefile Creation** creates *Makefile.config* from *Makefile.config.in* including information provided by this configuration.

10. **Summary** displays summary of each plug-in availability.

### 5.2 Makefiles

In this section, we detail the organization of the different Makefiles existing in Frama-C. First Section 5.2.1 presents a general overview. Next Section 5.2.2 details the different sections of *Makefile.config.in, Makefile.common, Makefile.generic, Makefile.generating* and *Makefile*. Next Section 5.2.3 introduces the variables customizing *Makefile.dynamic*. Finally Section 5.2.5 shows specific details of *Makefile.dynamic*.

#### 5.2.1 Overview

Frama-C uses different Makefiles (plus the plug-in specific ones). They are:

- **Makefile**: the general Makefile of Frama-C;
- **Makefile.generating**: it contains the complex rules that generate files. It is not directly in the general Makefile in order to reduce the dependencies of these rules to *Makefile.generating*;
- **Makefile.config.in**: the Makefile configuring some general variables (especially the ones coming from *configure*);
- **Makefile.common**: the Makefile providing some other general variables;
- **Makefile.generic**: the Makefile providing generic rules for compiling source files
- **.Makefile.plugin.generated**: the Makefile introducing specific stuff for plug-in compilation which is generated from *Makefile.plugin.template*;
- **Makefile.dynamic**: the Makefile usable by plug-in specific Makefiles.
- **Makefile.dynamic_config**: this Makefile is automatically generated either from *Makefile.dynamic_config.internal* or *Makefile.dynamic_config.external*. It sets variables which automatically configure *Makefile.dynamic*. 
5.2. MAKEFILES

- **Makefile.clean** contains specific targets for cleaning which are separated from **Makefile** for performance reasons.

- **.Makefile.user** is a per-user Makefile that can be used to override some variables. If it is not present, the default values of **Makefile** variables will be used.

**Makefile** and **.Makefile.user** are part of the root directory of the Frama-C distribution while the other ones are part of directory **share**. Each Makefile either includes or is included into at least another one. Figure 5.2 shows these relationships. **Makefile**

![Diagram showing the relationship between the Makefiles]

Caption:

1. **Makefile** \( m_1 \) is included in **Makefile** \( m_2 \)
2. **Makefile** \( m_1 \) is generated from **Makefile** \( m_2 \)

- **Orange boxes**: Plug-in Makefiles
- **Red boxes**: Generated Makefiles
- **Green boxes**: Other kernel Makefiles
- **Light green boxes**: Optional kernel Makefiles

Figure 5.2: Relationship between the Makefiles

and **Makefile.dynamic** are independent: the first one is used to compile the Frama-C kernel while the second one is used to compile the Frama-C plug-ins. Their common variables are defined in **Makefile.common** (which includes **Makefile.config.in**). They also include **Makefile.generic**, that defines default compilation rules for various kinds of source files. **Makefile.plugin.template** defines generic rules and variables for compiling plug-ins. It is used to generate **.Makefile.plugin.generated** for each plugin.
.Makefile.plugin.generated is included either by Makefile for kernel-specific plug-ins integrated in Frama-C Makefile and by Makefile.dynamic for plug-ins with their own Makefiles. .Makefile.user is included by Makefile when the former exists. It is only used when compiling Frama-C itself, and has no effect for external plugins.

### 5.2.2 Sections of Makefile, Makefile.generating, Makefile.config.in, Makefile.common and Makefile.generic

Figure 5.3 presents the different parts of Makefile.config.in, Makefile.common, Makefile.generic, Makefile.generating and Makefile in the order that they are introduced in these files. The third row of the tabular says whether the given part may be modified by a kernel-integrated plug-in developer. More details are provided below.

1. **Working directories** (split between Makefile.config.in and Makefile.common defines the main directories of Frama-C. In particular, it declares the variable FRAMAC_SRC_DIRS which should be extended by a plug-in developer if he uses files which do not belong to the plug-in directory (that is if variable PLUGIN_TYPES_CMO is set, see Section 5.2.3).  
2. **Installation paths** defines where Frama-C has to be installed.  
3. **OCaml stuff** defines the OCaml compilers and specific related flags.  
4. **Libraries** defines variables for libraries required by Frama-C.  
5. **Miscellaneous commands** defines some additional commands.  
6. **Miscellaneous variables** defines some additional variables.  
7. **Variables for plug-ins** defines some variables used by plug-ins distributed within Frama-C (and using the configure of Frama-C).  
8. **Flags** defines some variables setting compilation flags.  
9. **Verbosing** sets how make prints the command. In particular, it defines the variable VERBOSEMAKE which must be set yes in order to see the full make commands in the user console. The typical use is
   
   ```bash
   $ make VERBOSEMAKE=yes
   ```

10. **Shell commands** sets all the shell commands eventually executed while calling make.  
11. **Command pretty printing** sets all the commands to be used for pretty printing.

#### Example 5.1 Consider the following target foo in a plug-in specific Makefile.

```make
foo: bar
   $(PRINT_CP) $@
   $(CP) $< $@
```

** Executing

```bash
$ make foo
```

prints
## 5.2. MAKEFILES

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>File</th>
<th>Mod.</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Working directories</td>
<td>Makefile.config.in</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Installation paths</td>
<td>Makefile.config.in</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Ocaml stuff</td>
<td>Makefile.config.in</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Libraries</td>
<td>Makefile.config.in</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Miscellaneous commands</td>
<td>Makefile.config.in</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Miscellaneous variables</td>
<td>Makefile.config.in</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Variables for plug-ins</td>
<td>Makefile.config.in</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>1 (bis)</td>
<td>Working directories</td>
<td>Makefile.common</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Flags</td>
<td>Makefile.common</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Verbosing</td>
<td>Makefile.common</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Shell commands</td>
<td>Makefile.common</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Command pretty printing</td>
<td>Makefile.common</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Tests</td>
<td>Makefile.common</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Generic rules</td>
<td>Makefile.generic</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Source files generation</td>
<td>Makefile.generating</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Global plug-in variables</td>
<td>Makefile</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Additional global variables</td>
<td>Makefile</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Main targets</td>
<td>Makefile</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Coverage</td>
<td>Makefile</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Ocamlgraph</td>
<td>Makefile</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Frama-C Kernel</td>
<td>Makefile</td>
<td>yes</td>
<td>Section 4.3</td>
</tr>
<tr>
<td>21</td>
<td>Plug-in sections</td>
<td>Makefile</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Generic variables</td>
<td>Makefile</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>Toplevel</td>
<td>Makefile</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>GUI</td>
<td>Makefile</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>Standalone obfuscator</td>
<td>Makefile</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>Tests</td>
<td>Makefile</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>Emacs tags</td>
<td>Makefile</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Documentation</td>
<td>Makefile</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>Installation</td>
<td>Makefile</td>
<td>yes</td>
<td>Not written yet.</td>
</tr>
<tr>
<td>30</td>
<td>File headers: license policy</td>
<td>Makefile</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Makefile rebuilding</td>
<td>Makefile</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>Cleaning</td>
<td>Makefile</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>Depend</td>
<td>Makefile</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>ptests</td>
<td>Makefile</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>Source distribution</td>
<td>Makefile</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.3: Sections of Makefile.config.in, Makefile.common and Makefile.
CHAPTER 5. REFERENCE MANUAL

| Copying to foo

while executing

| $ make foo VERBOSEMBACE=yes

prints

| cp -f bar foo

If one of the two commands is missing for the target foo, either make foo or make foo
VERBOSEMBACE=yes will not work as expected.

12. **Tests** defines a generic template for testing plug-ins.

13. **Generic rules** contains rules in order to automatically produces different kinds of files
(e.g. .cm[iox] from .ml or .mli for OCaml files)

14. **Source files generation** contains rules for generating files that depend on the configuration and on which the main **Makefile** depends on. They are put in an auxiliary **Makefile.generate** to avoid unnecessary rebuilds.

15. **Global plug-in variables** declares some plug-in specific variables used throughout the makefile.

16. **Additional global variables** declares some other variables used throughout the makefile.

17. **Main targets** provides the main rules of the makefile. The most important ones are **top**, **byte** and **opt** which respectively build the **Frama-C** interactive, bytecode and native toplevels.

18. **Coverage** defines how compile the eponymous library.

19. **Ocamlgraph** defines how compile the eponymous library.

20. **Frama-C Kernel** provides variables and rules for the **Frama-C** kernel. Each part is described in specific sub-sections.

21. After Section “Kernel”, there are several sections corresponding to **plug-ins** (see Section 5.2.3). This is the part that a plug-in developer has to modify in order to add compilation directives for its plug-in.

22. **Generic variables** provides variables containing files to be linked in different contexts.

23. **Toplevel** provides rules for building the files of the form **bin/toplevel.***

24. **GUI** provides rules for building the files of the form **bin/viewer.***

25. **Standalone obfuscator** provides rules for building the **Frama-C** obfuscator.

26. **Tests** provides rules to execute tests. **make tests** takes care of generating the appropriate environment and launching **ptests** (see Section 4.5) for all test suites of the kernel and enabled plugins. It is possible to pass options to **ptests** through the **PTESTS_OPTS** environment variable.
5.2. MAKEFILES

27. **Emacs tags** provides rules which generate emacs tags (useful for a quick search of OCaml definitions).

28. **Documentation** provides rules generating Frama-C source documentation (see Section 4.22).

29. **Installation** provides rules for installing different parts of Frama-C.

30. **File headers: license policy** provides variables and rules to manage the Frama-C license policy.

31. **Makefile rebuilding** provides rules in order to automatically rebuild Makefile and configure when required.

32. **Cleaning** provides rules in order to remove files generated by makefile rules.

33. **Depend** provides rules which compute Frama-C source dependencies.

34. **Ptests** provides rules in order to build ptests (see Section 4.5).

35. **Source distribution** provides rules usable for distributing Frama-C.

### 5.2.3 Variables of Makefile.dynamic

Figures 5.4 and 5.5 presents all the variables that can be set before including Makefile.dynamic (see Section 4.4). Details are provided below.

- Variable **PLUGIN_NAME** is the module name of the plug-in.
  
  
  This name must be capitalized (as is each OCaml module name). It must be distinct from all other visible modules in the plugin directory, or in the Frama-C kernel.

- Variable **PLUGIN_DIR** is the directory containing plug-in source files. It is usually set to `src/plugins/plugin` where `plugin` is the plug-in name.

- Variable **PLUGIN_ENABLE** must be set to `yes` if the plug-in has to be compiled. It is usually set to `@plugin_ENABLE@` provided by configure.in where `plugin` is the plug-in name.

- Variable **PLUGIN_HAS_META** must be set to `yes` if plug-in `plugin` gets a META file describing its packaging. Unless **PLUGIN_HAS_META** is `yes`, the following variables are used to generate a suitable META:

  - **PLUGIN_VERSION** short text with `plugin` version number. Default value is Frama-C's version number.
  
  - **PLUGIN.Requires** packages that must be loaded before the `plugin`. By default, there is no such package.
  
  - **PLUGIN_DEPENDENCIES** other plug-ins that must be loaded before the `plugin`. By default, there is no dependency.

  **Remark:** the `plugin` package name is defined to be `frama-c-plugin` with lowercased plugin name. You can refer to it directly with `ocamlfind`. 
<table>
<thead>
<tr>
<th>Kind</th>
<th>Name</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plug-in Declaration</td>
<td>PLUGIN_NAME</td>
<td>Module name of the plug-in</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_DIR</td>
<td>Directory containing plug-in source files</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_ENABLE</td>
<td>Whether to compile the plug-in</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_HAS_META</td>
<td>Provided META (default: no)</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_DESCRIPTION</td>
<td>Short description (defaults to name)</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_VERSION</td>
<td>Version number (defaults to Frama-C’s)</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_DEPENDENCIES</td>
<td>Dependencies to other plug-ins</td>
</tr>
<tr>
<td></td>
<td>PLUGIN.Requires</td>
<td>Dependencies to ocamlfind packages</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_CMO</td>
<td>Object files (without .cmo)</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_CMI</td>
<td>Standalone interfaces (without .cmi)</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_GUI_CMO</td>
<td>Additional objects files for the GUI</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_TYPES_CMO</td>
<td>External .cmo files</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_BFLAGS</td>
<td>Plug-in specific flags for ocamlc</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_OFLAGS</td>
<td>Plug-in specific flags for ocamlopt</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_EXTRA_BYTE</td>
<td>Additional bytecode files to link against</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_EXTRA_OPT</td>
<td>Additional native files to link against</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_EXTRA_DIRS</td>
<td>Additional directories containing source files, relative to the root directory of the plugin (i.e. PLUGIN_DIR)</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_LINK_BFLAGS</td>
<td>Plug-in specific flags for linking with ocamlc</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_LINK_OFLAGS</td>
<td>Plug-in specific flags for linking with ocamlopt</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_LINK_GUI_BFLAGS</td>
<td>Plug-in specific flags for linking a GUI with ocamlc</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_LINK_GUI_OFLAGS</td>
<td>Plug-in specific flags for linking a GUI with ocamlopt</td>
</tr>
</tbody>
</table>

Figure 5.4: Standard parameters of Makefile.dynamic.
5.2. MAKEFILES

- Variables PLUGIN_CMO and PLUGIN_CMI are respectively .cmo plug-in files and .cmi files without corresponding .cmo plug-in files. For each of them, do not write their file path nor their file extension: they are automatically added (${PLUGIN_DIR}/f.cmo[io]) for a file f).

- Variable PLUGIN_TYPES_CMO is the .cmo plug-in files which do not belong to ${PLUGIN_DIR}. They usually belong to src/plugins/plugin_types where plugin is the plug-in name (see Section 4.9.2). Do not write file extension (which is .cmo): it is automatically added.

- Variable PLUGIN_GUI_CMO is the .cmo plug-in files which have to be linked with the GUI (see Section 4.21). As for variable PLUGIN_CMO, do not write their file path nor their file extension.

- Variables of the form PLUGIN_*_FLAGS are plug-in specific flags for ocamlc, ocamlopt, ocamldep or ocamldoc.

- Variable PLUGIN_GENERATED is files which must be generated before computing plug-in dependencies. In particular, this is where .ml files generated by ocamlyacc and ocamllex must be placed if needed.

- Variable PLUGIN_DEPENDS is the other plug-ins which must be compiled before the considered plug-in.

  Using this variable is deprecated: you should consider to use PLUGIN_DEPENDENCIES instead.

- Variable PLUGIN_UNDOC is the source files for which no documentation is provided. Do not write their file path which is automatically set to ${PLUGIN_DIR}.

- Variable PLUGIN_TYPES_TODOC is the additional source files to document with the plug-in. They usually belong to src/plugins/plugin_types where plugin is the plug-in name (see Section 4.9.2).

- Variable PLUGIN_INTRO is the text file to append to the plug-in documentation introduction. Usually this file is doc/code/intro_plugin.txt for a plug-in plugin. It can contain any text understood by ocamldoc.

- Variable PLUGIN_HAS_EXT_DOC is set to yes if the plug-in has its own reference manual. It is supposed to be a pdf file generated by make in directory doc/${PLUGIN_NAME}).

- Variable PLUGIN_NO_TEST must be set to yes if there is no specific test directory for the plug-in.

- Variable PLUGIN_TESTS_DIRS is the directories containing plug-in tests. Its default value is tests/$(notdir $(PLUGIN_DIR))).

- Variable PLUGIN_TESTS_LIB contains the .ml plug-in specific files used by plug-in tests. Note that if you wish to have these files compiled before launching the tests (as opposed to e.g. compile them via an EXECNOW directive in a test file), you should add the corresponding .cmo and .cmxs files as pre-requisite of the ${PLUGIN_DIR}/tests/ptests_config target.
<table>
<thead>
<tr>
<th>Kind</th>
<th>Name</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies</td>
<td>PLUGIN_DEPFLAGS</td>
<td>Plug-in specific flags for ocamldep</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_GENERATED</td>
<td>Plug-in files to be generated before running ocamldep</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_DEPENDS</td>
<td>Other plug-ins to be compiled before the considered one</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_DOCFLAGS</td>
<td>Plug-in specific flags for ocamldoc</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_UNDOC</td>
<td>Source files with no provided documentation</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_TYPES_TO_DOC</td>
<td>Additional source files to document</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_INTRO</td>
<td>Text file to append to the plug-in introduction</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_HAS_EXT_DOC</td>
<td>Whether the plug-in has an external pdf manual</td>
</tr>
<tr>
<td>Testing</td>
<td>PLUGIN_NO_TESTS</td>
<td>Whether there is no plug-in specific test directory</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_TESTS_DIRS</td>
<td>Directories containing tests tests to be included in the default test suite (all directories by default)</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_TESTS_DIRS_DEFAULT</td>
<td>Specific .ml files used by plug-in tests</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_TESTS_LIBS</td>
<td>Specific .ml files used by plug-in tests</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_NO_DEFAULT_TEST</td>
<td>Whether to include tests in default test suite.</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_INTERNAL_TEST</td>
<td>Whether the test suite of the plug-in is located in Frama-C's own tests directory</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_PTESTS_OPTS</td>
<td>Plug-in specific options to ptests</td>
</tr>
<tr>
<td>Distribution</td>
<td>PLUGIN_DISTRIBUTED_BIN</td>
<td>Whether to include the plug-in in binary distribution</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_DISTRIBUTED</td>
<td>Whether to include the plug-in in source distribution</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_DISTRIB_EXTERNAL</td>
<td>Additional files to be included in the distribution</td>
</tr>
</tbody>
</table>

Figure 5.5: Special parameters of Makefile.dynamic.
Variable `PLUGIN_NO_DEFAULT_TEST` indicates whether the test directory of the plug-in should be considered when running Frama-C default test suite. When set to a non-empty value, the plug-in tests are run only through `make $(PLUGIN_NAME)_tests`.

Variable `PLUGIN_INTERNAL_TEST` indicates whether the tests of the plug-in are included in Frama-C's own tests directory. When set to a non-empty value, the tests are searched there. When unset, tests are assumed to be in the tests directory of the plugin main directory itself. Having the tests of a plugin inside Frama-C's own tests suite is deprecated. Plugins should be self-contained.

Variable `PLUGIN_PTESTS_OPTS` allows to give specific options to `ptests` when running the tests. It comes in addition to `PTESTS_OPTS` (see 5.2.2§26). For instance, `PLUGIN_PTESTS_OPTS:=-j 1` will deactivate parallelization of tests in case the plugin does not support concurrent runs.

Variable `PLUGIN_DISTRIB_BIN` indicates whether the plug-in should be included in a binary distribution.

Variable `PLUGIN_DISTRIBUTED` indicates whether the plug-in should be included in a source distribution.

Variable `PLUGIN_DISTRIB_EXTERNAL` is the list of files that should be included within the source distribution for this plug-in. They will be put at their proper place for a release.

As previously said, the above variables are set before including `Makefile.dynamic` in order to customize its behavior. They must not be use anywhere else in the Makefile. In order to deal with this issue, for each plug-in `p`, `Makefile.dynamic` provides some variables which may be used after its inclusion defining `p`. These variables are listed in Figure 5.6. For each variable of the form `p_VAR`, its behavior is exactly equivalent to the value of the parameter `PLUGIN_VAR` for the plug-in `p`.

5.2.4 .Makefile.user

The following variables can be set inside `.Makefile.user`:

- `FRAMAC_PARALLEL`: the contents of this variable are passed to `ptests` when it is called by Frama-C’s Makefile. This variable can be used to override the default value of 4.

  *Example value:* `-j 6`

- `OCAML_ANNOT_OPTION`: this variable of the Makefile can be overridden in `.Makefile.user`. By default, it is set to `-annot -bin-annot`. Users of Merlin do not need `-annot`.

  *Example value:* `-bin-annot`

- `FRAMAC_USER_FLAGS`: the contents of this variable are passed to `ocamlc` and `ocamlopt`, after the standard configuration options coming from the Makefile. It can be used to tweak the warnings emitted by OCaml, and whether they are emitted as errors.

---

1Variables of the form `p_*CMX` have no `PLUGIN_*CMX` counterpart but their meanings should be easy to understand.

2`plugin` is the module name of the considered plug-in (i.e. as set by `$(PLUGIN_NAME)`).
<table>
<thead>
<tr>
<th>Kind</th>
<th>Name</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usual information</td>
<td>plugin_DIR</td>
<td></td>
</tr>
<tr>
<td>Source files</td>
<td>plugin_CMO</td>
<td></td>
</tr>
<tr>
<td></td>
<td>plugin_CMI</td>
<td></td>
</tr>
<tr>
<td></td>
<td>plugin_CMX</td>
<td></td>
</tr>
<tr>
<td></td>
<td>plugin_TYPES_CMO</td>
<td></td>
</tr>
<tr>
<td></td>
<td>plugin_TYPES_CMX</td>
<td></td>
</tr>
<tr>
<td>Targets</td>
<td>plugin_TARGET_CMO</td>
<td>Empty if plugin does not have external dependencies</td>
</tr>
<tr>
<td></td>
<td>plugin_TARGET_CMX</td>
<td>Empty if plugin does not have external dependencies</td>
</tr>
<tr>
<td></td>
<td>plugin_TARGET_CMA</td>
<td>Empty if plugin is not dynamic</td>
</tr>
<tr>
<td></td>
<td>plugin_TARGET_CMXA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>plugin_TARGET_CMXS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>plugin_TARGET_GUI_CMO</td>
<td>Empty if there is no plugin-specific GUI code</td>
</tr>
<tr>
<td></td>
<td>plugin_TARGET_GUI_CMX</td>
<td></td>
</tr>
<tr>
<td></td>
<td>plugin_TARGET_GUI_CMA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>plugin_TARGET_GUI_CMXA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>plugin_TARGET_GUI_CMXS</td>
<td></td>
</tr>
<tr>
<td>Compilation flags</td>
<td>plugin_BFLAGS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>plugin_OFLAGS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>plugin_LINK_BFLAGS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>plugin_LINK_OFLAGS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>plugin_LINK_GUI_BFLAGS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>plugin_LINK_GUI_OFLAGS</td>
<td></td>
</tr>
<tr>
<td>Dependencies</td>
<td>plugin_DEPFLAGS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>plugin_GENERATED</td>
<td></td>
</tr>
<tr>
<td>Documentation</td>
<td>plugin_DOCFLAGS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>plugin_TYPES_TODOCC</td>
<td></td>
</tr>
<tr>
<td>Testing</td>
<td>plugin_TESTS_DIRS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>plugin_TESTS_LIB</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.6: Variables defined by `Makefile.dynamic`.
Example value: \texttt{-warn-error -26-27} (do not consider warnings 26 and 27 as fatal errors)

- \texttt{FRAMAC_USER_MERLIN_FLAGS}: the contents of this variable are passed to a directive \texttt{FLG}, inside the \texttt{.merlin} file generated for \texttt{Frama-C}. (Through the \texttt{merlin} target of the main Makefile.) It can be used to tailor Merlin to your needs. See \texttt{ocamlmerlin -help} for the list of flags.

5.2.5 Makefile\_dynamic

\textit{Not written yet}: please report as “feature request” on \url{http://bts.frama-c.com} if you really need this section.

5.3 Ptests

5.3.1 Pre-defined macros for tests commands

Ptests pre-defines a certain number of macros for each test about to be run. Figure 5.7 gives their definition.

<table>
<thead>
<tr>
<th>Name</th>
<th>Expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{frama-c}</td>
<td>path to \texttt{Frama-C} executable</td>
</tr>
<tr>
<td>\texttt{PTEST_CONFIG}</td>
<td>either the empty string or \texttt{_} followed by the name of the current alternative configuration (see section 4.5.3).</td>
</tr>
<tr>
<td>\texttt{PTEST_DIR}</td>
<td>current test suite directory</td>
</tr>
<tr>
<td>\texttt{PTEST_RESULT}</td>
<td>current result directory</td>
</tr>
<tr>
<td>\texttt{PTEST_FILE}</td>
<td>path to the current test file</td>
</tr>
<tr>
<td>\texttt{PTEST_NAME}</td>
<td>basename of the current test file (without suffix)</td>
</tr>
<tr>
<td>\texttt{PTEST_NUMBER}</td>
<td>rank of current test in the test file. There are in fact two independent numbering schemes: one for \texttt{EXECNOW} commands and one for regular tests (if more than one \texttt{OPT}).</td>
</tr>
</tbody>
</table>

Figure 5.7: Predefined macros for ptests

5.4 Profiling with Landmarks

\textit{Landmarks}\footnote{\url{https://github.com/LexiFi/landmarks}} 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. The \texttt{Frama-C configure} file is setup to enable usage of this library when it is available (the usual way to install it is via the \texttt{landmarks} opam package).

For quick usage of the library:

- ensure that the \texttt{configure} script detected it (there should be a line \texttt{checking for Landmarks... found});
• enable instrumentation when compiling Frama-C’s files, that is, when running `make`, by setting the environment variable `OCAML_LANDMARKS`. For instance, to enable automatic instrumentation of every Frama-C function (note: this increases compilation time of Frama-C), run:

```
OCAML_LANDMARKS=auto make
```

• enable instrumentation during execution of Frama-C, again using `OCAML_LANDMARKS`. Note that the `auto` parameter here is implicit if you enabled it on the previous step. For instance, run:

```
OCAML_LANDMARKS= bin/frama-c [files] [options]
```

Commonly used options include `output=landmarks.log` to output the result to a file instead of `stderr`. Check [https://github.com/LexiFi/landmarks](https://github.com/LexiFi/landmarks) for its documentation.
Appendix A
Changes

This chapter summarizes the major changes in this documentation between each Frama-C release, from newest to oldest.

20.0 Calcium

- **Ptests**: Documentation of the new directive `MODULE`.

19.0 Potassium

- **ACSL Extension**: Document new `status` flag for registration functions
- **Testing**: Document of usage `@@` 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
APPENDIX A. CHANGES

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`.
- **Configure.in**: warn about using Frama-C macros within conditionals

126
- **Logical Annotations**: document ACSL extended clauses mechanism (added section 4.19).
- **Tutorial**: fix hello_world.ml.

**Neon-20140301**

- **Logical Annotations**: document module Property.
- **Command Line Options**: update according to kernel changes that split the module Plugin into several modules.
- **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.
- **Website**: refer to CEA internal documentation.
- **Command Line Options**: explain how to modify the default behavior of an option.
APPENDIX A. CHANGES

- **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 *STD OPT* 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
- **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**


### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Plug-in Integration Overview.</td>
<td>15</td>
</tr>
<tr>
<td>2.2</td>
<td>Control flow graph for file test.c.</td>
<td>29</td>
</tr>
<tr>
<td>2.3</td>
<td>Control flow graph colored with reachability information.</td>
<td>32</td>
</tr>
<tr>
<td>2.4</td>
<td>CFG plug-in architecture</td>
<td>34</td>
</tr>
<tr>
<td>3.1</td>
<td>Frama-C Architecture Design</td>
<td>42</td>
</tr>
<tr>
<td>4.1</td>
<td>ptests options.</td>
<td>56</td>
</tr>
<tr>
<td>4.2</td>
<td>Directives in configuration headers of test files.</td>
<td>57</td>
</tr>
<tr>
<td>4.3</td>
<td>Representation of the Frama-C State.</td>
<td>77</td>
</tr>
<tr>
<td>4.4</td>
<td>Interaction between the project library and your registered global data.</td>
<td>78</td>
</tr>
<tr>
<td>4.5</td>
<td>Indices of AST nodes.</td>
<td>101</td>
</tr>
<tr>
<td>5.1</td>
<td>Sections of configure.in</td>
<td>111</td>
</tr>
<tr>
<td>5.2</td>
<td>Relationship between the Makefiles</td>
<td>113</td>
</tr>
<tr>
<td>5.3</td>
<td>Sections of Makefile.config.in, Makefile.common and Makefile.</td>
<td>115</td>
</tr>
<tr>
<td>5.4</td>
<td>Standard parameters of Makefile.dynamic.</td>
<td>118</td>
</tr>
<tr>
<td>5.5</td>
<td>Special parameters of Makefile.dynamic.</td>
<td>120</td>
</tr>
<tr>
<td>5.6</td>
<td>Variables defined by Makefile.dynamic.</td>
<td>122</td>
</tr>
<tr>
<td>5.7</td>
<td>Predefined macros for ptests</td>
<td>123</td>
</tr>
</tbody>
</table>
Index

.Makefile.plugin.generated, 112
.Makefile.user, 113, 114, 121

Abstract Interpretation, 44, 107
Abstract_interp, 107
Annotation, 99, 103
Annotations, 44, 103
  add_assert, 102
Architecture, 41
  Plug-in, 14
AST, 77, 99, 103
  Copying, 100, 102
  Modification, 45, 83, 99–101
  Sharing, see Sharing
Ast
  add_monotonic_state, 81
  get, 30, 77
  mark_as_changed, 81
  mark_as_grown, 81
  self, 37, 83, 102
Boot, 88
Cabs, 44
  file, 90
Cabs2cil
  convFile, 90
check_plugin, 46
Cil, 99
  cilVisitor, 99, 99
    behavior, 102
    current_kinstr, 102
  fill_global_tables, 100
  get_filling_actions, 100, 102
  vexpr, 102
  vfile, 28
  vglob, 99
  vlogic_ctor_info_decl, 99
  vlogic_ctor_info_use, 99
  vlogic_info_decl, 99
  vlogic_info_use, 99
  vlogic_type_info_decl, 99
  vlogics_type_info_use, 99
  vlogic_var_decl, 99
  vlogic_var_use, 99
  voffs, 99
  vstmt, 99
  vvdec, 99
  vvrbl, 99
  copy_visit, 100, 102
dooryStmt, 75
  get_kernel_function, 102
  get_original_varinfo, 101
  get_stmt, 102
  get_varinfo, 101
  inplace_visit, 100
lzero, 102
  register_behavior_extension, 106
reset_behavior_varinfo, 101
set_varinfo, 101
visiAction, 99
  ChangeDoChildrenPost, 100–102
  ChangeTo, 100–102
  ChangeToPost, 100
  DoChildren, 28, 31, 100, 102
  DoChildrenPost, 28, 100
  JustCopy, 28, 100, 101
  JustCopyPost, 100
  SkipChildren, 28, 100–102
visitCilAstType, 99
visitCilFile, 99
visitCilFileCopy, 99
visitCilFileSameGlobals, 99
visitor_behavior, 100

Cil_datatype, 67
Fundec
  Hashtbl, 37
Stmt
  equal, 67
  Hashtbl, 79
  pretty, 67, 75
  t, 75
  ty, 75, 76
INDEX

Datatype, 67
  Bool, 39
  bool, 67
  char, 67
  func, 74, 76
  func2, 67
  func3, 76
  Function, 71
  identity, 68
  Int, 70
  int, 67
  List, 70, 71
  list, 67
  Make, 68–70
  never_any_project, 68
  Pair, 79
  Polymorphic, 70
  Polymorphic2, 70
  Polymorphic3, 70
  Polymorphic4, 70
  pp_fail, 69
  Ref, 82
  S, 67, 82
  S_with_collections, 67
  Serializable_undefined, 69, 75
  String, 37, 68
    Hashtbl, 70
  Set, 68
  string, 67
  Undefined, 69
  undefined, 69
  unit, 74, 76
Db, 43, 72, 72, 73, 108
  From.self, 80
  Impact.compute_pragmas, 72
Main, 15
  extend, 14, 16, 17, 21, 23, 26, 61, 88, 90, 102
  progress, 107
Properties, 103
Value
  compute, 79, 83
  get_stmt_state, 31
  is_computed, 31, 39, 78, 83
  is_reachable, 31
  self, 37, 79, 80, 84
Design, 15
  main_window_extension_points, 107
  register_source_selector, 32

register_extension, 32, 107
Documentation, 108, 117
  Kernel, 108
  Plug-in, see Plug-in Documentation
  Source, 108
  Tags, 108
Dot, 48
Dynamic, 15, 43, 73, 108
  get, 73, 74, 76
  Parameter, 87
    Bool, 87
    register, 73, 74
Emitter, 85
Emitter
  create, 103
Entry Point, 78
Entry point, 14
Equality
  Physical, 82
  Structural, 82
Extlib
  the, 102
FCHashtbl, 43
File
  add_code_transformation_after_cleanup, 90
  add_code_transformation_before_cleanup, 90
  create_project_from_visitor, 102
  init_from_c_files, 99
  init_from_cmdline, 99
  init_project_from_cil_file, 83, 99
  init_project_from_visitor, 83, 99
  must_recompute_cfg, 90
  new_file_type, 90
  new_machdep, 91
FRAMAC_INTERNAL, 52
FRAMAC_LIBDIR, 19, 20, 22, 33, 51
FRAMAC_SHARE, 19, 20, 22, 33, 51
FRAMAC_SRC_DIRS, 73, 114
From, 80, 81
From_parameters
  ForceCallDeps, 86
Frontc
  add_syntactic_transformation, 90
Globals, 44
Functions
  get, 32
  set_entry_point, 78
GnomeCanvas, 48
INDEX

**Gtk_helper**
- graph_window, 32

**GUI**, 15, 107

**Hashtable**, 79, 80

**Header**, 117

**Hello**, 45

**Highlighting**, 108

**Hook**, 14

**index.html**, 108, 109

**Initialization**, 74, 87, 88

**install**, 52

**install-doc-code**, 109

**Journal**, 15

**Journalisation**, 68

**Journalization**, 36, 74, 88, 89

**Kernel**, 41, 81, 116
- **Internals**, 44
- **Services**, 43

**Kernel**, 85
- **CodeOutput**, 66
- **SafeArrays**, 87
- **Unicode**, 86

**Kernel_function**, 44, 79
- **dummy**, 75
- **get_definition**, 32
- **Make_Table**, 80, 81
- **pretty**, 75
- **t**, 75, 79
- **ty**, 75, 76

**Kind**, 80

**Lablgtk**, 48, 107

**Lablgtksourceview2**, 48

**Landmarks**, 123

**Lattice**, 107

**Lattice_type**, 107

**Library**, 46
- **Configuration**, 47, 111, 112
- **Dependency**, 48

**License**, 117

**Linking**, 88

**Lmap**, 107

**Lmap_bitwise**, 107

**Loading**, 78, 83, 89

**Location**, 106

**Locations**, 106
- **enumerate_valid_bits**, 107

**Location**, 106

**location**, 106

**Location_Bits**, 107

**Location_Bytes**, 106

**Zone**, 107

**Log**
- **add_listener**, 64
- **log_channel**, 66

**Messages**, 61, 62
- **abort**, 62
- **debug**, 62
- **error**, 62
- **failure**, 63
- **fatal**, 63
- **feedback**, 62
- **log**, 65
- **register_warn_category**, 64
- **result**, 62
- **set_warn_status**, 64
- **verify**, 63
- **warn_category**, 64
- **warning**, 62
- **with_log**, 65
- **new_channel**, 66
- **print_delayed**, 66
- **print_on_output**, 66
- **set_echo**, 64
- **set_output**, 66
- **with_log_channel**, 66

**Logging**, see **Messages**

**Logic_const**, 103
- **new_acsl_extension**, 104
- **prel**, 102

**Logic_typing**
- **register_behavior_extension**, 104
- **register_code_annot_extension**, 104
- **register_code_annot_next_both_extension**, 104
- **register_code_annot_next_loop_extension**, 104
- **register_code_annot_next_stmt_extension**, 104
- **register_global_extension**, 104
- **typing_context**, 105

**Logic_utils**, 103
- **expr_to_term**, 102

**Machine model**, 91

**Makefile**, 50, 107–109, 112, 113, 113, 114

**Makefile.clean**, 113
INDEX

Makefile.common, 112, 114
Makefile.config, 112
Makefile.config.in, 112, 113, 114
Makefile.dynamic, 15, 19, 20, 22, 33, 50, 51, 71, 72, 112, 113, 117
Makefile.dynamic_config, 112
Makefile.dynamic_config.external, 112
Makefile.dynamic_config.internal, 112
Makefile.generating, 112, 113, 114
Makefile.generic, 112, 114
Makefile.plugin.generated, 113
Makefile.plugin.template, 112
Marshaling, 68
Memo, 79
Memoization, 77, 79, 80
Merlin, 121, 123
Messages, 60
Module Initialization, see Initialization
Occurrence, 46, 107
Oracle, 52, 53, 56
Parameter, 77
Parameter_customize, 87
    set_negative_option_name, 87
Parameter_sig
    Bool, 85
        Empty_string, 85
        False, 85, 86
        Int, 85
        Kernel_function_set, 86, 86
        String, 85
        String_set, 85
        True, 85
        Zero, 85
        Int, 85, 86
        Kernel_function_set, 86
        S, 85
Parameter_state
    get_selection, 85
Parameters, 85
Pdg, 81
Platform, 112
Plug-in, 13, 43
    Access, 73
    API, 36, 73
    Architecture, 14
    Command Line Options, 18, 30
    Compilation, 116
Configure, 37
Dependency, 46, 46, 49, 112
Directory, 107, 117
Distribution, 121
GUI, 15, 31, 48, 88, 107, 119
Initialization, see Initialization
Kernel-integrated, 43, 111, 114
    Access, 72
    Registration, 72
Makefile, 19, 33
Messages, 17
Name, 117
Occurrence, see Occurrence
Pdg, see Pdg
Registration, 16, 73
Script, 15
Sparecode, see Sparecode
Status, 46
Test, 119, 121
Testing, 22
Types, 72, 119
Wished, 112
plugin_types, 73
Plugin, 14, 15, 60
    Register, 16, 17, 21, 25, 30, 43, 60, 61, 75, 85
PLUGIN_BFLAGS, 119
plugin_BFLAGS, 122
PLUGIN_CMI, 119
plugin_CMI, 122
PLUGIN_CMD, 19, 20, 22, 33, 51, 71, 72, 119
plugin_CMD, 122
PLUGIN_DEPENDENCIES, 72, 117
PLUGIN_DEPENDS, 119
PLUGIN_DEPFLAGS, 119
plugin_DEPFLAGS, 122
PLUGIN_DIR, 117
plugin_DIR, 122
PLUGIN_DISTRIB_BIN, 121
PLUGIN_DISTRIB_EXTERNAL, 121
PLUGIN_DISTRIBUTED, 121
PLUGIN_DOCFLAGS, 119
plugin_DOCFLAGS, 122
PLUGIN_ENABLE, 117
PLUGIN_EXTRA_BYTE, 119
PLUGIN_EXTRA_DIRS, 119
PLUGIN_EXTRA_OPT, 119
PLUGIN_GENERATED, 119
INDEX

plugin_GENERATED, 122
PLUGIN_GUI_CMO, 33, 107, 119
plugin_GUI_OFLAGS, 122
PLUGIN_HAS_EXT_DOC, 119
PLUGIN_HAS_META, 117
PLUGIN_INTERNAL_TEST, 121
PLUGIN_INTRO, 109, 119
plugin_LINK_BFLAGS, 122
PLUGIN_LINK_GUI_BFLAGS, 119
plugin_LINK_GUI_BFLAGS, 122
PLUGIN_LINK_GUI_OFLAGS, 119
plugin_LINK_OFLAGS, 122
PLUGIN_NAME, 19, 20, 22, 33, 51, 71, 72, 108, 117
PLUGIN_NO_DEFAULT_TEST, 121
PLUGIN_NO_TEST, 119
PLUGIN_OFLAGS, 119
plugin_OFLAGS, 122
PLUGIN_PTESTS_OPTS, 121
PLUGINQUIRES, 117
plugin_TARGET_CMA, 122
plugin_TARGET_CMX, 122
plugin_TARGET_CMXA, 122
plugin_TARGET_CMXS, 122
plugin_TARGET_GUI CMA, 122
plugin_TARGET_GUI CMO, 122
plugin_TARGET_GUI CMX, 122
plugin_TARGET_GUI CMXA, 122
plugin_TARGET_GUI CMXS, 122
plugin_TARGET_CMO, 122
PLUGIN_TESTS_DIRS, 22, 119
plugin_TESTS_DIRS, 122
PLUGIN_TESTS_LIB, 119
plugin_TESTS_LIB, 122
PLUGIN_TYPES_CMO, 73, 114, 119
plugin_TYPES_CMO, 122
plugin_TYPES_CMX, 122
plugin_TYPES_TODOCC, 119
plugin_TYPES_TODOCC, 122
PLUGIN_TYPES_TODOCD, 119
PLUGIN_TYPES_TODOCD, 122
PLUGIN_UNDOC, 119
PLUGIN_VERSION, 117
Pretty_source
  PVDecl, 32
Pretty_utils, 43
PRINT_CP, 114
Printer_api
  S.pp_stmt, 27
  S.pp_varinfo, 28
Project, 37, 45, 68, 77, 99, 100
  Current, 77, 77, 82, 83, 85, 100
  Initial, 99
  Use, 83
Project, 15, 83
  clear, 39, 39, 83, 84
  current, 77, 83
  I0Error, 83
  load, 83
  on, 84, 84, 85
  save, 83
  set_current, 83, 83, 84
  t, 39
Project_skeleton
  t, 83
Property, 103
Property_status, 103
PTests, 22, 52, 117
PTESTS_OPTS, 116
Saving, 45, 78, 79, 83
Selection, 78, 84
self, 80
Session, 83
Sharing, 100, 101
  Widget, 108
Side-Effect, 82, 88
Sparecode, 53
State, 77, 84, 85, 99
  Cleaning, 82, 84
  Dependency, 78, 80, 82, 84, 85
    Postponed, 80, 89
  Functionalities, 77
  Global Version, 82
  Kind, see Kind
  Local Version, 82, 82
  Name, 80, 82
  Registration, 77–79
  Selection, see Selection
  Sharing, 82
State, 80
dummy, 80
State_builder, 79, 80
Hashtbl, 37
Ref, 39, 82
Register, 78, 80, 81, 82
State_dependency_graph
  S.add_codependencies, 81
INDEX

State_selection, 84
  only_dependencies, 83
  t, 39
  with_dependencies, 39, 84

Structural_descr
  p_int, 68, 70
  pack, 70
  structure
    Sum, 68, 70
  t
    Structure, 68, 70

Tags, 117
Test, 22, 52, 116
  Configuration, 22, 54
  Directive, 22, 54
    CMD, 57, 58
    COMMENT, 57
    DONTRUN, 57
    EXEC, 57, 58
    EXECNOW, 57, 58
    FILEREG, 57, 59
    FILTER, 57
    GCC, 57
    LOG, 57
    MACRO, 57, 59
    MODULE, 57, 59
    OPT, 54, 57
    STDOPT, 57, 58
  Header, 22, 54, 55
  Suite, 53, 53

Test
  Directive
    OPT, 22
  test_config, 54, 57, 59
  tests, 53, 57

Tool, 46
  Configuration, 47, 111, 112
  Dependency, 48

Type
  Dynamic, 67
  Value, 67, 74, 75

Type, 15
  Abstract, 75, 76
  AlreadyExists, 74
  name, 70
  par, 68, 70
  precedence
    Basic, 68
    Call, 68

Typed_parameter
  t, 85

VERBOSEMAKE, 50, 114, 116
Visitor, 26, 99
  Behavior, 100, 101
  Cil, 99
    Entry Point, 99
  Copy, 83, 100, 100, 101
  In-Place, 100, 100

Visitor
  frama_c_inplace, 27
  frama_c_visitor
    current_kf, 102
    vglobal_aux, 28, 99
    vstmt_aux, 28, 31, 99
  generic_frama_c_visitor, 99, 102
  visitFramacFileSameGlobals, 30
  visitFramacFunction, 38