



Software Analyzers

Developer Manual





list

Plug-in Development Guide

Release Fluorine-20130601

Julien Signoles with Loïc Correnson, Matthieu Lemerre and Virgile Prevosto

CEA LIST, Software Reliability Laboratory, Saclay,F-91191

©2009-2013 CEA LIST

This work has been supported by the ANR project CAT (ANR-05-RNTL-00301) and the ANR project U3CAT (08-SEGI-02101).



Contents

Foreword	9
1 Introduction	11
1.1 About this document	11
1.2 Outline	12
2 Tutorial	13
2.1 What a Plug-in Looks Like?	13
2.2 The Hello plug-in	13
2.2.1 A Simple Script	14
2.2.2 Registering a Script as a Plug-in	15
2.2.3 Displaying Messages	15
2.2.4 Writing a Makefile	16
2.2.5 Adding Command Line Options	17
2.2.6 Testing your Plug-in	19
2.2.7 Documenting your Source Code	19
2.3 The CFG plug-in	19
2.3.1 Visiting the AST	19
2.3.2 Interfacing with a kernel-integrated plug-in	22
2.3.3 Extending the Frama-C GUI	23
2.3.4 Plug-In registration and command line options	24
2.3.5 Splitting files and writing a Makefile	25
2.3.6 Getting your Plug-in Usable by Others	28
2.3.7 Writing your Plug-in into the Journal	28
2.3.8 Writing a Configure Script	28
2.3.9 Getting your plug-in Usable in a Multi Projects Setting	28
3 Software Architecture	33
3.1 General Description	33
3.2 Cil: C Intermediate Language	35

CONTENTS

3.3	Kernel	35
3.4	Plug-ins	36
4	Advanced Plug-in Development	39
4.1	File Tree Overview	39
4.2	Frama-C Configure.in	40
4.2.1	Principle	40
4.2.2	Addition of a Simple Plug-in	41
4.2.3	Configuration of New Libraries or Tools	41
4.2.4	Addition of Library/Tool Dependencies	42
4.2.5	Addition of Plug-in Dependencies	43
4.3	Plug-in Specific Configure.in	43
4.4	Frama-C Makefile	44
4.5	Plug-in Specific Makefile	45
4.5.1	Using Makefile.dynamic	45
4.5.2	Compiling Frama-C and external plug-ins at the same time	46
4.6	Testing	46
4.6.1	Using ptests	47
4.6.2	Configuration	48
4.6.3	Alternative Testing	49
4.6.4	Detailed options	50
4.6.5	Detailed directives	50
4.7	Plug-in General Services	52
4.8	Logging Services	53
4.8.1	From printf to Log	54
4.8.2	Log Quick Reference	54
4.8.3	Logging Routine Options	55
4.8.4	Advanced Logging Services	56
4.9	The Type library: Type Values and Datatypes	58
4.9.1	Type Value	59
4.9.2	Datatype	59
4.10	Plug-in Registration and Access	62
4.10.1	Kernel-integrated Registration and Access	62
4.10.2	Dynamic Registration and Access	63
4.11	Journalization	66
4.12	Project Management System	66
4.12.1	Overview and Key Notions	66
4.12.2	State: Principle	67

CONTENTS

4.12.3	Registering a New State	69
4.12.4	Direct Use of Low-level Functor <code>State_builder.Register</code>	71
4.12.5	Using Projects	72
4.12.6	Selections	74
4.13	Command Line Options	74
4.13.1	Definition	75
4.13.2	Tuning	76
4.14	Initialization Steps	77
4.15	Visitors	79
4.15.1	Entry Points	79
4.15.2	Methods	80
4.15.3	Action Performed	80
4.15.4	Visitors and Projects	81
4.15.5	In-place and Copy Visitors	81
4.15.6	Differences Between the Cil and Frama-C Visitors	82
4.15.7	Example	82
4.16	Logical Annotations	83
4.17	Locations	84
4.17.1	Representations	84
4.17.2	Map Indexed by Locations	85
4.18	GUI Extension	85
4.19	Documentation	85
4.19.1	General Overview	86
4.19.2	Source Documentation	86
4.19.3	Website	87
4.20	License Policy	87
5	Reference Manual	89
5.1	File Tree	89
5.1.1	The <code>cil</code> directory	90
5.1.2	The <code>src</code> directory	91
5.2	<code>Configure.in</code>	93
5.3	Makefiles	94
5.3.1	Overview	94
5.3.2	Sections of <code>Makefile</code> , <code>Makefile.config.in</code> and <code>Makefile.common</code>	94
5.3.3	Variables of <code>Makefile.dynamic</code> and <code>Makefile.plugin</code>	99
5.3.4	<code>Makefile.dynamic</code>	102

CONTENTS

A Changes	105
Bibliography	111
List of Figures	113
Index	115

Foreword

This is the documentation of the **Frama-C** implementation¹ which aims to help developers integrate new plug-ins inside this platform. It started as a deliverable of the task 2.3 of the ANR RNTL project CAT².

The content of this document corresponds to the version Fluorine-20130601 (June 11, 2013) 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: Patrick Baudin, Richard Bonichon, Pascal Cuoq, Zaynah Dargaye, Florent Garnier, Pierre-Loïc Garoche, Philippe Herrmann, Boris Hollas, Nikolai Kosmatov, Benjamin Monate, Yannick Moy, Anne Pacalet, Armand Puccetti, Muriel Roger and Boris Yakobowski. We also thank Johannes Kanig for his Mlpost support³, the tool 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 4.19.1 for additional details about this documentation).

This guide introduces neither the use of Frama-C which is the purpose of the user manual [4] and of the reference article [8], nor the use of plug-ins which are documented in separated and dedicated manuals [2, 5, 10, 13, 20]. 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 [14]. Motivations for this choice are given in a Frama-C experience report [9]. However this guide *does not* provide any introduction to this programming language: the World Wide Web already contains plenty resources for OCaml developers (see for instance <http://caml.inria.fr/resources/doc/index.en.html>).

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 `ocaml` 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 does not tell the whole story. You will get it with your own experiment, and by reading the other chapters of this guide on need.

First Section 2.1 shows what a plug-in looks like. Then Section 2.2 explains the basis for writing a standard **Frama-C** plug-in, while Section 2.3 details how to interact with **Frama-C** and others plug-ins to implement analyzers of C programs.

2.1 What a Plug-in Looks Like?

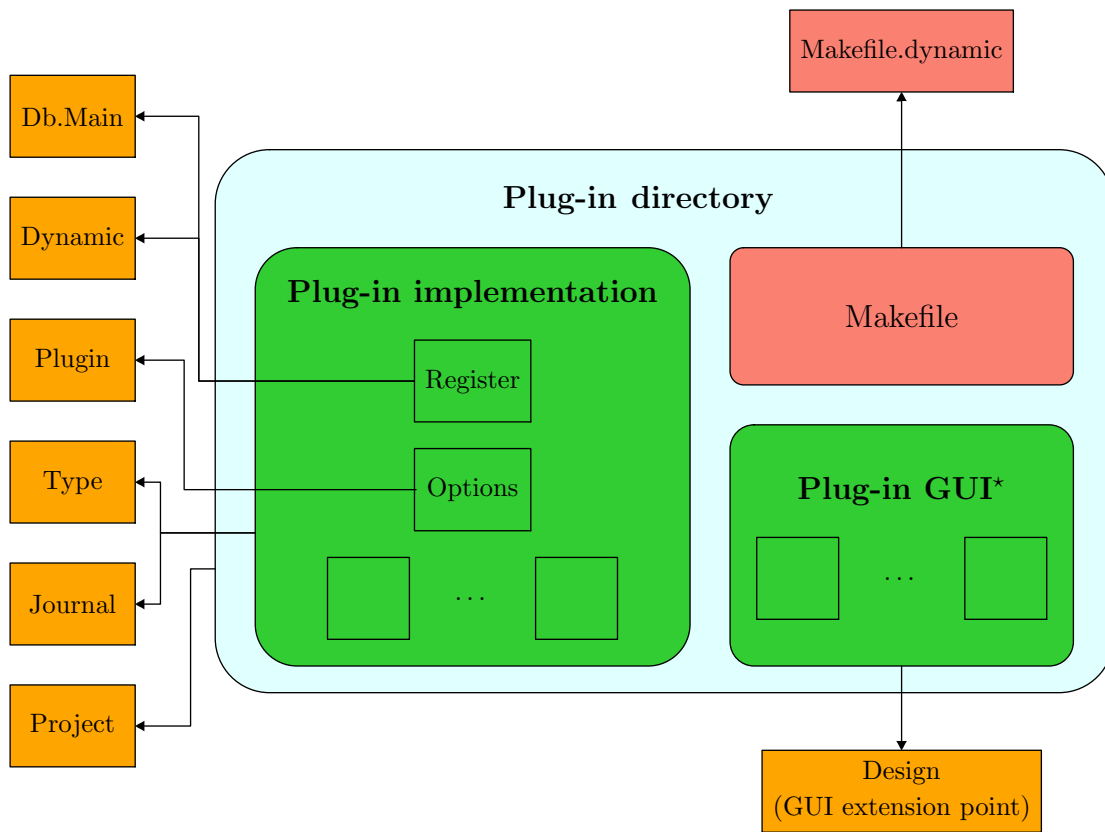
Figure 2.1 shows how a plug-in can integrate with the **Frama-C** platform. This tutorial focuses on specific parts of this figure.

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.

2.2 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 interaction between APIs.



Caption:

→ registration points

Figure 2.1: Plug-in Integration Overview.

2.2.1 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:

File `hello_world.ml`¹

```
let run () =
  let chan = open_out "hello.out" in
  Printf.fprintf chan "Hello, world!\n";
  close_out chan

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

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.

¹To be complete, this code (and some others in this tutorial) should handle exceptions potentially raised by file operations (opening, writing and closing). We omit them for the sake of clarity.

2.2.2 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:

File `hello_world.ml`

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

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

let run () =
  let chan = open_out "hello.out" in
  Printf.fprintf chan "Hello, world!\n";
  close_out chan

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.
- `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 -help`);
- default options for the plug-in work, including the inline help (available with `frama-c -load-script hello_world.ml -hello-help`).

2.2.3 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 instead².

File `hello_world.ml`

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

module Self = Plugin.Register
  (struct
    let name = "hello world"
    let shortname = "hello"
```

²However writing to a new file using standard OCaml primitives is OK.

```

    let help = help_msg
  end)

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

let () = Db.Main.extend run

```

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:

```

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

```

For a comprehensive list of the logging routines and options, see Section 4.8.

2.2.4 Writing a Makefile

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

A simple Makefile

We first write a simple `Makefile` for our `hello_world.ml` plug-in:

File Makefile

```

FRAMAC_SHARE :=$(shell frama-c.byte -print-path)
FRAMAC_LIBDIR :=$(shell frama-c.byte -print-libpath)
PLUGIN_NAME  = Hello
PLUGIN_CMO   = hello_world
include $(FRAMAC_SHARE)/Makefile.dynamic

```

This `Makefile` set 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 4.5 for details).

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

Run `make` to compile it. You can then load and execute the module using `frama-c -load-module ./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.

Splitting your source files

Here is a slightly more complex example where the plug-in has been splitted 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 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 -print-path)
FRAMAC_LIBDIR := $(shell frama-c -print-libpath)
PLUGIN_NAME = Hello
PLUGIN_CMO = hello_options hello_print hello_run
include $(FRAMAC_SHARE)/Makefile.dynamic
```

File `hello_options.ml`

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

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

File `hello_print.ml`

```
let print_hello () = Hello_register.Self.result "Hello, World"
```

File `hello_run.ml`

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

2.2.5 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.

File `hello_world.ml`

```

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

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

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

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

let run () =
  if Enabled.get () then
    let filename = Output_file.get () in
    let output_fun =
      if Output_file.is_default () then
        Self.result "%s"
      else
        let chan = open_out filename in
        Printf.fprintf chan "%s\n" ;
        close_out chan
      in
    output_fun "Hello, world!"

let () = Db.Main.extend run

```

Registering these new options is done by calling the `Self.False` and `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 *via* `Enabled.get ()` and `Output_file.get ()`.

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

```

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

```

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

```

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

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

```

2.2.6 Testing your Plug-in

Not written yet: please report as “feature request” on <http://bts.frama-c.com> if you really need this section.

2.2.7 Documenting your Source Code

Not written yet: please report as “feature request” on <http://bts.frama-c.com> if you really need this section.

2.3 The CFG plug-in

In this section, we create a new CFG 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 the APIs of Frama-C.

2.3.1 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.

The first case is usually best if you have to compute information for most C constructs, while the latter is best 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.

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 label the node representing the current statement: substatements will be represented by other nodes. Thus we will use the following small function:

```
open Cil_types

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

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 of type `Cil_types.stmt`. The `Printer` module contains functions that prints the different Cil types. The documentation of these module is available on the Frama-C website³, 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.

```
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 4.15.4).

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

```
method vfile _ =
  Format.fprintf out "@[<hov 2>digraph cfg {@ ";
  Cil.DoChildrenPost (fun f → Format.fprintf out "}@]."; f)
```

`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.

```
method vglob_aux g =
  match g with
  | GFun(f, _) →
    Format.fprintf out "@[<hov 2>subgraph cluster_%a {@ \
      @[<hv 2>graph@ [label=\"%a\"];@]@ "
      Printer.pp_varinfo f.svar
      Printer.pp_varinfo f.svar;
    Cil.DoChildrenPost (fun g → Format.fprintf out "}@]."; g)
  | _ → Cil.SkipChildren
```

`Cil.SkipChildren` tells the visitor not to visit the children nodes, which makes it more efficient⁴.

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

```
method vstmt_aux s =
  Format.fprintf out "@[<hov 2>s%d@ [[label=\"%a\"];@]@ ";
  s.sid print_stmt s.skind;
  List.iter
    (fun succ → Format.fprintf out "@[s%d → s%d;@]@ " s.sid succ.sid)
    s.succs;
  Format.fprintf out "@]";
  Cil.DoChildren
```

³From <http://frama-c.com/download.html>.

⁴In a copying visitor, `Cil.JustCopy` should have been used instead.

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

Finally we close the `object` definition:

```
| end
```

Hooking into Frama-C

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

```
| let run () =
|   let chan = open_out "cfg.out" 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
```

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

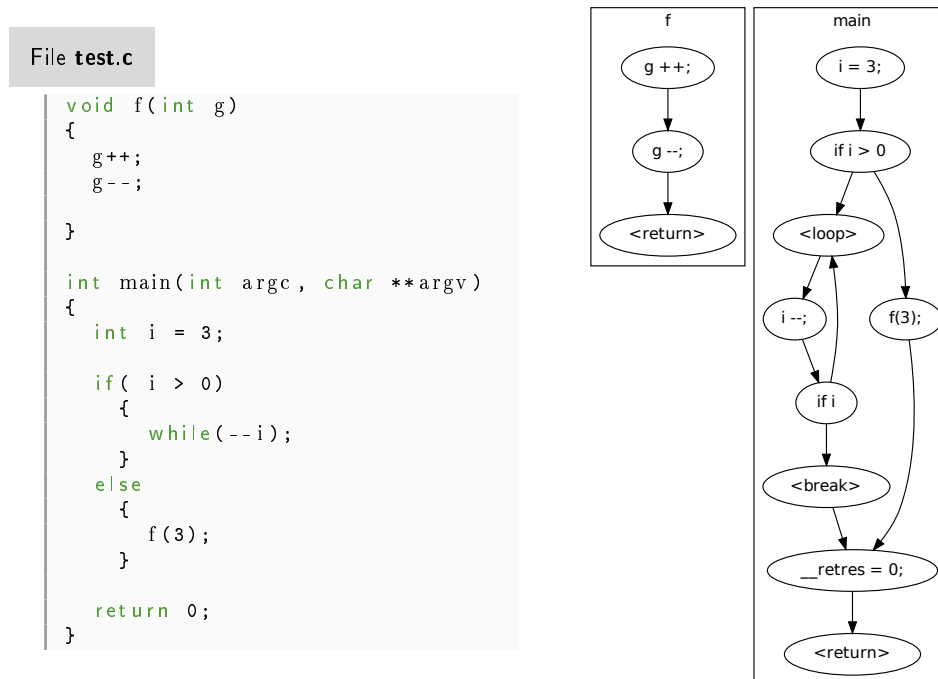


Figure 2.2: Control flow graph for file `test.c`.

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.

2.3.2 Interfacing with a kernel-integrated plug-in

We will first integrate with the `value` plug-in. There are two ways to integrate a plug-in with Frama-C: standard plug-ins, like the `cfg` plug-in, are loaded dynamically. The `value` plug-in is kernel-integrated, which means that it is registered statically through the `Db` module of the Frama-C kernel which is the entry point for all kernel-integrated plug-ins. The `value` plug-in is accessible through the `Db.Value` module.

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:

```
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=\"%a\" %s]@];@ "
    s.sid print_stmt s.kind 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 -val -then -load-script cfg_print.ml && dotty cfg.out
```

The use of `-then` ensures that value analysis is performed before the CFG script is loaded (see Frama-C User Manual [4]).

The resulting graph is shown on Figure 2.3.

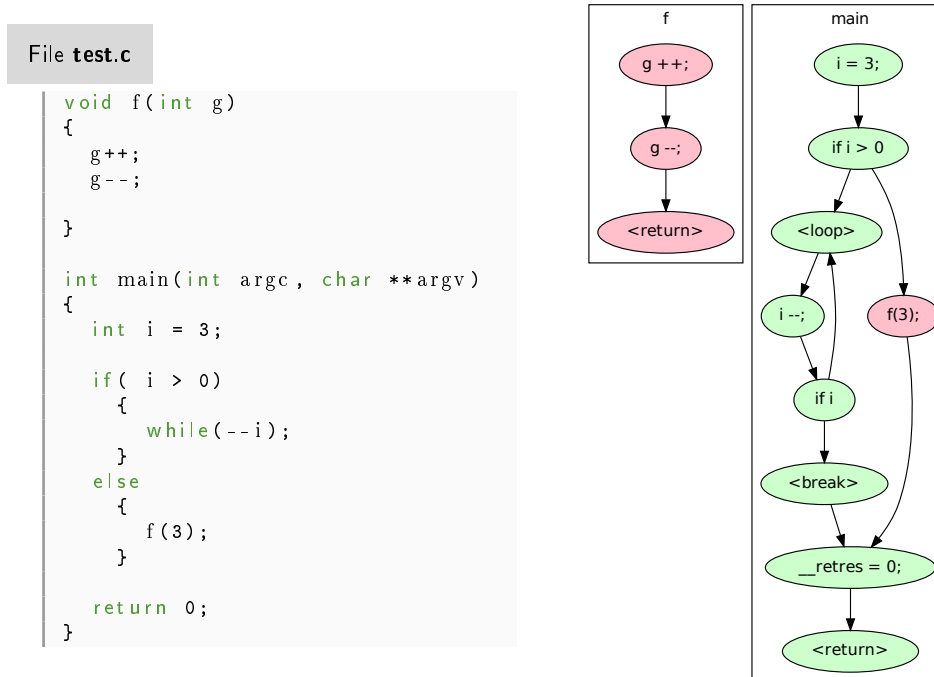


Figure 2.3: Control flow graph colored with reachability information.

2.3.3 Extending the Frama-C GUI

In this section, we will extend the script 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 outputs the CFG of a single function instead.

```

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

```

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:

```

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
      Gtk_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 accesses to the main widgets of the window, and several extension points.

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 `cfg_selector` function.

This function takes a `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 (*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 `callback` function if clicked. The `callback` function calls a Frama-C GUI function that displays a graph from dot printing functions. It uses several important Frama-C APIs: `Globals` and `Kernel_function` which contains several functions for manipulating globals and functions.

A script is not an ideal environment for extending the GUI. In particular, the GUI OCaml modules are available only when the script is loaded with `frama-c-gui`, and not when loaded with `frama-c`. When the user wants to view CFG from the GUI, then outputting the CFG of all functions in `cfg.out` is useless. We will now see how to cope with these problems, by promoting the script into a true plug-in.

2.3.4 Plug-In registration and command line options

We already saw how to register functions in the previous “Hello” tutorial. We now apply these principles to the CFG plug-in.

```

module Self = Plugin.Register(struct
  let name = "control flow graph"
  let shortname = "cfg"
  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 graph 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 have added two options, `-cfg` to compute the CFG conditionnaly, and `-cfg-output` to choose the output file.

An interesting addition would be to add a `-cfg-target` option, which would take a set of files or functions whose CFG would be computed, using the `Self.StringSet` 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 for 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 been already executed. The `apply_once` function in the `State_builder` module helps dealing with this issue (reading the section 2.3.9 of this tutorial and section 4.12 of this manual should help you understand the underlying notion of states).

2.3.5 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 `frama-c-gui`. This requires splitting the module into several files. We choose the following architecture:

- `cfg_options.ml` implements plug-in registration and configuration options;
- `cfg_core.ml` implements the main functions for computing the CFG;
- `cfg_register.ml` implements “global” computation of the CFG using the `-cfg` option, and hooking into the `Frama-C` main loop;
- `cfg_gui.ml` implements GUI registration.

Dependencies between the modules⁵ 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 `run` function is at the top. The GUI is also at the top of the hierarchy: the `Frama-C Makefile` requires that normal plug-in modules do not depend on GUI modules. Note that currently, the dependency from `Cfg_core` and `Cfg_gui` to `Cfg_register` is artificial, but in future evolutions the GUI could depend on configuration options.

File Makefile

```
FRAMAC_SHARE := $(shell frama-c -print-path)
FRAMAC_LIBDIR := $(shell frama-c -print-libpath)
PLUGIN_NAME = Cfg
PLUGIN_CMO = cfg_options cfg_core cfg_register
PLUGIN_GUI_CMO = cfg_gui
include $(FRAMAC_SHARE)/Makefile.dynamic
```

⁵This graphic is generated in file `doc/code/modules.dot` after running `make doc`.

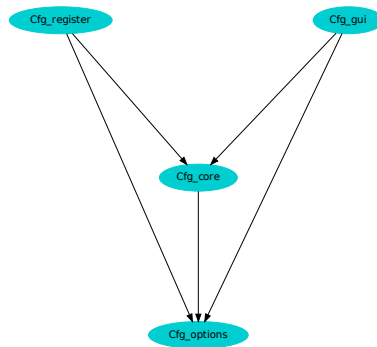


Figure 2.4: CFG plug-in architecture

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`.

Here is the listing for the different modules:

File `cfg_options.ml`

```

module Self = Plugin.Register(struct
  let name = "control flow graph"
  let shortname = "cfg"
  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 graph 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`

```

module Options = Cfg_options
open Cil_types

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

class print_cfg out = object
  inherit Visitor.frama_c_inplace

```

2.3. THE CFG PLUG-IN

```

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

method vglob_aux g =
  match g with
  | GFun(f, _) →
    Format.fprintf out "@[<hov 2>subgraph cluster_%a {0 \
      @[<hv 2>graph0 [label=\"%a\"];0]0 "
      Printer.pp_varinfo f.svar
      Printer.pp_varinfo f.svar;
    Cil.DoChildrenPost(fun g → Format.fprintf out "}0]0 "; g)
  | _ → Cil.SkipChildren

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=\"%a\" %s]0];0 "
  s.sid print_stmt s.skid color;
  List.iter
  (fun succ → Format.fprintf out "@[s%d → s%d;0]0 " s.sid succ.sid)
  s.succs;
  Cil.DoChildren

end

let dump_function fundec fmt =
  Format.fprintf fmt "@[<hov 2>digraph cfg {0 ";
  ignore(Visitor.visitFramacFunction (new print_cfg fmt) fundec);
  Format.fprintf fmt "}0]\n"

```

File `cfg_register.ml`

```

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`

```

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

```

```

    Gtk_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

```

2.3.6 Getting your Plug-in Usable by Others

Not written yet: please report as “feature request” on <http://bts.frama-c.com> if you really need this section.

2.3.7 Writing your Plug-in into the Journal

Not written yet: please report as “feature request” on <http://bts.frama-c.com> if you really need this section.

2.3.8 Writing a Configure Script

Not written yet: please report as “feature request” on <http://bts.frama-c.com> if you really need this section.

2.3.9 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 [16] 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:

```

module Cfg_graph_state = State_builder.Hashtbl
  (Cil_datatype.Fundec.Hashtbl)
  (Datatype.String)
  (struct
    let name = "Data_for_cfg.Cfg_graph_state"
    let dependencies = [ Ast.self; Db.Value.self ]
    let size = 17
    end);;

```

self

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 [19], described in Section 4.9. `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 function has been declared, it is fairly easy to use it.

```

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 resetted 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:

```

module Value_is_computed = State_builder.Ref
  (Datatype.Bool)
  (struct
    let name = "Data_for_cfg.Value_computed"
    let dependencies = []
    let default () = false
  end);;

```

This new state only consists of a reference to a boolean value.

Then we just replace `dump_function` in the code above by the following.

```

let dump_function fundec fmt =
  if not (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* [18] 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 it 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 CFG plug-in is executed only once, as explained in section 2.3.4 of this tutorial.



Software Architecture

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 three different parts:

- Section 3.2 introduces the API of Cil [17] seen by Frama-C;
- Section 3.3 shows the organisation of the Frama-C kernel; and
- Section 3.4 explains the plug-in integration.

3.1 General Description

The Frama-C platform provides 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 is based on a software architecture with a specific design which is presented in this document, and summarized in Figure 3.1. Mainly this architecture is separated in three different parts:

- Cil (C Intermediate Language) [17] extended with an implementation of the specification language ACSL (ANSI/ISO C Specification Language) [1]. This is the intermediate language upon which Frama-C is based. See Section 3.2 for details.
- **The Frama-C kernel.** It is a toolbox on top of Cil dedicated to static analyses. It provides data structures and operations which help the developer to deal with the Cil AST (Abstract Syntax Tree), as well as general services providing an uniform set of features to Frama-C. See Section 3.3 for details.
- **The Frama-C plug-ins.** These are analyses or source-to-source transformations that use the kernel, and possibly others plug-ins through the APIs they register in the Frama-C kernel. See Section 3.4 for details.

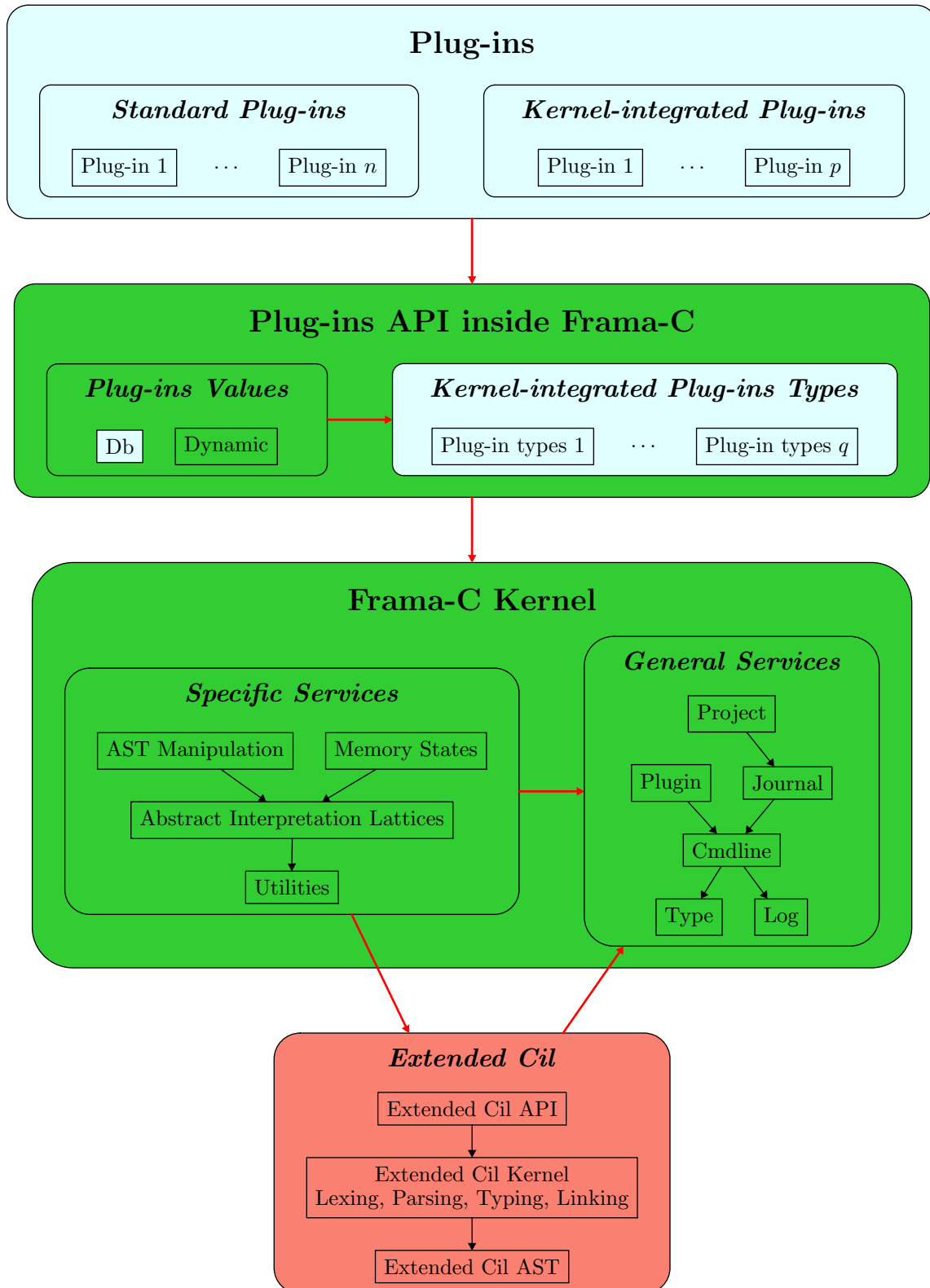


Figure 3.1: Architecture Design.

3.2 Cil: C Intermediate Language

Cil [17] is a high-level representation along with a set of tools that permit easy analysis and source-to-source transformation of C programs.

Frama-C uses Cil as a library which performs the main steps of the compilation of C programs (pre-processing, lexing, parsing, typing and linking) and outputs an abstract syntax tree (AST) ready for analysis. From the Frama-C developer's point of view, Cil is a toolbox usable through its API and providing:

- the AST description (module `Cil_types`);
- useful AST operations (module `Cil`);
- some syntactic analysis like a (syntactic) call graph computation (module `Callgraph`) or generic forward/backward dataflow analysis (module `Dataflow`).

Frama-C indeed extends Cil with ACSL (ANSI/ISO C Specification Language) [1], its specification language. The extended Cil API consequently provides types and operations in order to properly deal with *annotated* C programs.

Cil modules belong to directory (and subdirectories of) `cil/src`.

3.3 Kernel

On top of the extended Cil API, the Frama-C kernel groups together specific services providing in different modules which are described below.

- In addition to the Cil utilities, Frama-C provides useful operations (mainly in module `Extlib`) and datastructures (e.g. specialized version of association tables like `Rangemap`). These modules belong to directories `src/lib` and `src/misc` and they are not specific to the analysis or transformation of C programs.
- Frama-C provides generic lattices useful for abstract interpretation (module `Abstract_interp`) and some pre-instantiated arithmetic lattices (module `Ival`). The abstract interpretation toolbox is available in directory `src/ai`.
- Frama-C also provides different representations of C memory-states (module `Locations`) and data structures using them (e.g. association tables indexing by memory-states in modules `Lmap` and `Lmap_bitwise`). The memory-state toolbox is available in directory `src/memory_state`.
- Moreover, directory `src/kernel` provides a bunch of very helpful operations over the extended Cil AST. For example, module `Globals` provides operations dealing with global variables, functions and annotations while module `Visitor` provides inheritable classes in order to permit easy visiting, copying or in-place modification of the AST.

Besides, Frama-C also provides some general-purpose services, used by all other modules (even the Frama-C version of Cil), which are shortly described below.

- Module `Log` provides an uniform way to display user messages in Frama-C.

- Module `Cmdline` parses the Frama-C command line.
- Module `Plugin` provides a high-level API on top of the two previous modules for the plug-in developer: a developer usually uses this module and does not directly use modules `Log` or `Cmdline`.
- Directory `src/type` contains a library called `Type`. It provides OCaml types as first-class values, as well as several standard operations over types. Such values and operations are required by several other Frama-C services (journalization, registration of dynamic values, projects, *etc*). See section 4.9 for details.
- Module `Journal` handles how Frama-C journalizes its actions. See section 4.11 for details.
- Directory `src/project` contains a library, called `Project`. It provides analysis of several ASTs in the same session. See section 4.12 for details.

3.4 Plug-ins

In Frama-C, plug-ins are analysis or source-to-source transformations. Each of them is an extension point of the Frama-C kernel. Frama-C allows plug-in collaborations: a plug-in p can use a list of plug-ins p_1, \dots, 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, its API has to be registered, either in module `Dynamic` or in module `Db`. This last method is only available to kernel-integrated plug-ins and is now deprecated.

More generally, the set of functionalities available for a standard plug-in and for a kernel-integrated plug-in are mostly the same. The differences between a standard plug-in and a kernel-integrated one are listed Figure 3.2.

Functionality	Standard plug-in	Kernel-integrated plug-in
dynamic linking	default	possible
static linking	possible	default
API in <code>Dynamic</code>	possible	possible
API in <code>Db</code>	no	possible by modifying the kernel
add new abstract types	possible	possible
add new concrete types	no	possible by modifying the kernel

Figure 3.2: Differences between standard plug-ins and kernel-integrated ones.

Both kinds of plug-ins may be either dynamically linked or statically linked within the Frama-C kernel. dynamic linking is the standard way for standard plug-ins while static linking is the standard way for kernel-integrated plug-ins.

Dynamic linking is only available in native mode if you have both OCaml 3.11 or higher and a supported architecture for this feature. See the OCaml manual [14] for additional details.

Both kinds of plug-ins may register their API through module `Dynamic`, but the standard way for kernel-integrated plug-ins is the use of module `Db`. Kernel-integrated plug-ins may also declare any types inside the Frama-C kernel thanks to the so-called *Kernel-integrated Plug-ins*

3.4. PLUG-INS

Types'. Such types are usable by any plug-in, and even by some parts of the **Frama-C** kernel. However any plug-in may still register a new abstract type and use it through the function provided by the plug-in API. See Section [4.10](#) for details.



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 your analysis on disk, see Section 4.12.2);
3. introducing bugs in your code;
4. introducing bugs in other plug-ins using your code;
5. breaking the kernel consistency and so potentially breaking all the `Frama-C` plug-ins (*e.g.* if you modify the AST without changing project, see Section 4.12.5).

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 File Tree Overview

Target readers: *beginners*.

The `Frama-C` main directory is split in several sub-directories. The `Frama-C` source code is mostly provided in directories `cil` and `src`. The first one contains the source code of `Cil` [17]

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

extended with an ACSL [1] implementation. The second one is the core implementation of Frama-C. This last directory contains directories of the Frama-C kernel and directories of the provided Frama-C plug-in.

A pretty complete description of the Frama-C file tree is provided in Section 5.1.

4.2 Frama-C `Configure.in`

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

Prerequisite: *knowledge of autoconf and shell programming.*

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 2.3.8 and 2.3.5).

First Section 4.2.1 introduces the general principle and organisation of `configure.in`. Then Section 4.2.2 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 4.2.4) and with other plug-ins (Section 4.2.5). Finally Section 4.2.3 presents the configuration of external libraries and tools needed by a new plug-in but not used anywhere else in Frama-C.

4.2.1 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 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 5.2. Now we focus on the modifications to perform in order to integrate a new plug-in in Frama-C.

4.2.2 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.

```
# occurrence
#####
check_plugin(occurrence,src/occurrence,
             [support for occurrence analysis],yes,no)
```

The first argument is the plug-in name, the second one is the name of directory containing the source files of the plug-in, the third one is a help message for the `-enable-occurrence` option of `configure`, the fourth one indicates if the plug-in is enabled by default and the last one indicates if the plug-in will be dynamically linked within the Frama-C kernel.

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 4.7 (with spaces replaced by underscore).

The macro `check_plugin` sets the following variables: `FORCE_OCCURRENCE`, `REQUIRE_OCCURRENCE`, `USE_OCCURRENCE`, `ENABLE_OCCURRENCE`, and `DYNAMIC_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 4.2.5). 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`). Finally, `DYNAMIC_OCCURRENCE` indicates whether the plug-in will be dynamically linked within the Frama-C kernel.

4.2.3 Configuration of New Libraries or Tools

Some plug-ins needs 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 the external dependency exists on the system. Second, it verifies for each plug-in that its dependencies are met. Section 4.2.4 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`². 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

²For tools, there is a macro `configure_tool` which works in the same way as `configure_library`.

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 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`. This is checked through the following command:*

```
configure_library(
  [GTKSOURCEVIEW],
  [${OCAMLLIB}/lablgtk2/lablgtksourceview2.${LIB_SUFFIX}],
  [lablgtksourceview not found])
```

4.2.4 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` [12] 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`.*

```
plugin_require_external(gui, lablgtk)
plugin_require_external(gui, gnomecanvas)
plugin_use_external(gui, dot)
```

This line specifies that `Lablgtk2` must be available on the system if the user wants to compile `gui`.

4.2.5 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 4.2.4). 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`.

4.3 Plug-in Specific Configure.in

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 4.5.2, 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

```
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 4.5.2 for more details).

The configuration of the plug-in itself or related libraries and tools can then proceed as described in Sections 4.2.2 and 4.2.3. References to specific files in the plug-in source directory should be guarded with the following macro:

```
| PLUGIN_RELATIVE_PATH(file)
```

If the external plug-in has some dependencies as described in sections 4.2.4 and 4.2.5, the configure script `configure` must check that all dependencies are met. This is done with the following macro:

```
| 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 4.5.2 for more details).

Finally, the configuration must end with the following command:

```
| 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.

4.4 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 5.3.2. By default, executing `make` only displays an overview of commands. For example, here is the output of the compilation of source file `src/kernel/db.cmo`.

```
| $ make src/kernel/db.cmo
| Ocamlc      src/kernel/db.cmo
```

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

```
| $ make VERBOSEMAKE=yes src/kernel/db.cmo
| ocamlc.opt -c -w +a-4-6-7-9 -annot -warn-error +a-32-33-34-35-36-37-38 -g
| -I src/misc -I src/ai -I src/memory_state -I src/toplevel -I src/slicing_types
| -I src/pdg_types -I src/kernel -I src/logic -I src/lib -I src/type
| -I src/project -I src/buckx -I src/gui -I external -I cil/src -I cil/src/ext
| -I cil/src/frontc -I cil/src/logic -I cil/ocamlutil -I lib/plugins -I lib
| src/kernel/db.ml
```

By default, warnings are considered as errors, but some of the new warnings of OCaml 4.00 are not. If you wish to make them errors as well, set variable `WARN_ERROR_ALL` to `yes`³

In order to integrate a new plug-in, you have to extend section “Plug-ins”. For this purpose, you have to include `share/Makefile.plugin` for each new plug-in (hence there are as many lines `include share/Makefile.plugin` as plug-ins). `Makefile.plugin` is a generic makefile dedicated to plug-in compilation. Before its inclusion, a plug-in developer can set some variables in order to customize its behavior. These variables are fully described in Section 5.3.3.

These variables must not be used anywhere else in `Makefile`. Moreover, for setting them, you must use `:=` and not `=`⁴.

In addition, the results of the `configure` script must be exported in `share/Makefile.config.in` (see section 5.3.2). You must in particular add a line of the form

³this has no effect if you use OCaml < 4.00

⁴Using `:=` only sets the variable value from the affectation point (as usual in most programming languages) whereas using `=` would redefine the variable value for each of its occurrences in the makefile (see Section 6.2 “The Two Flavors of Variables” of the GNU Make Manual [11]).

```
ENABLE_plugin=@ENABLE_plugin@
```

so that `make` will know whether the plug-in is supposed to be compiled or not. Other variables may be exported there as well (`DYNAMIC_plugin`, `HAS_library`) if the corresponding information is needed during compilation.

Example 4.3 *For compiling the plug-in `Rte`, the following lines are added into `Makefile`.*

```
#####
# RTE analysis #
#####
PLUGIN_ENABLE := $(ENABLE_RTE_ANNOTATION)
PLUGIN_NAME := RteGen
PLUGIN_DIR := src/rte
PLUGIN_CMO := rte_parameters rte_register
PLUGIN_HAS_MLI := yes
PLUGIN_DISTRIBUTED := yes
PLUGIN_INTERNAL_TEST := yes
include share/Makefile.plugin
```

As said above, you cannot use the parameters of `Makefile.plugin` anywhere in `Makefile`. You can yet use some plug-in specific variables once `Makefile.plugin` has been included. These variables are detailed in Section 5.3.3.

One other variable has to be modified 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). This variable is `UNPACKED_DIRS` and corresponds to the list of non plug-in directories containing source files.

A plug-in developer should not have to modify any other part of any Frama-C Makefile.

4.5 Plug-in Specific Makefile

Prerequisite: *knowledge of `make`.*

In this section, we detail how to add a Makefile to a plug-in.

4.5.1 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 4.4 *A minimal Makefile is shown below. That is the Makefile of the plug-in `Hello World` presented in the tutorial (see Section 2.2). Each variable set in this example has to be set by any plug-in.*

```
# Example of Makefile for dynamic plugins
#####

# Frama-c should be properly installed with "make install"
# before any use of this makefile

FRAMAC_SHARE      :=$(shell frama-c.byte -print-path)
FRAMAC_LIBDIR     :=$(shell frama-c.byte -print-libpath)
PLUGIN_NAME       = Hello
PLUGIN_CMO        = hello_world
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.

To run your specific Makefile, you must have properly installed Frama-C before.

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

Which variable can be set and how they are useful is explained Section 5.3.3. Furthermore, Section 5.3.4 explains the specific features of `Makefile.dynamic`.

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

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

It is also possible to have a completely independent plug-in recompile and test 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/` 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-ins plug-ins: `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.

4.6 Testing

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

`ptests` 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 4.6.1 shows how to use `ptests`. Next Section 4.6.2 introduces how to use predefined directives to configure tests, while Section 4.6.3 explains how to set up various testing goals for the same test base. Last Section 4.6.4 details `ptests`' options, while Section 4.6.5 describes `ptests`' directive.

4.6.1 Using ptests

If you're using a `Makefile` written following the principles given in section 4.5, the simplest way of using `ptests` is through `make tests` which is roughly equivalent to

```
| $ time ./bin/ptests.byte
```

or

```
| $ time ptests.byte
```

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

In addition, `make tests` ensures that the `ptests` options specific to the plug-in under test are correctly set. This can be done manually by issuing the following command:

```
| $ make ptests_local_config.cmo
```

File `ptests_local_config.cmo` is then loaded by `ptests` and contains all the necessary information for handling the test suite of the current plug-in.

`ptests.byte` runs tests belonging to a sub-directory of directory `tests` that is mentioned in `ptests_local_config`. `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*

```
| $ ./bin/ptests.byte sparecode tests/pdg/variadic.c
```

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 4.6.4.

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

```
| $ ./bin/ptests.byte plug-in
| $ ./bin/ptests.byte -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.byte -show tests/plug-in/new_test.c
$ ./bin/ptests.byte -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`

4.6.2 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

```
/* 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 4.6.5.

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` or `EXECNOW` 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` use `make` to create a `.cmxs`

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

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

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

```
$ ./bin/ptests.byte 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
```

4.6.3 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.byte -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 ...
*/
```

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

```
| $ ./bin/ptests.byte -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.

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

Figure 4.1: `ptests` options.

4.6.4 Detailed options

Figure 4.1 details the options of `ptests`.

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.

4.6.5 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 considered for `.` is always the parent directory of directory `tests`. The `DONTRUN` directive

Kind	Name	Specification	default
Command	CMD	Program to run	<code>./bin/toplevel.opt</code>
	OPT	Options given to the program	<code>-val -out -input -deps</code>
	STDOPT	Add and remove options from the default set	<i>None</i>
	EXECNOW	Run a command before the following commands	<i>None</i>
	FILTER	Command used to filter results	<i>None</i>
Test suite	DONTRUN	Do not execute this test	<i>None</i>
	FILEREG	selects the files to test	<code>.*\.(c i\)</code>
Miscellaneous	COMMENT	Comment in the configuration	<i>None</i>
	GCC	Unused (compatibility only)	<i>None</i>

Figure 4.2: Directives in configuration headers of test files.

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 `ptest`s, it is always executed, regardless of the presence of a `DONTRUN` directive.

As said in Section 4.6.2, 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 4.6.3, alternative directives for test configuration `<special_name>` can be found in slightly different places:

- 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 to change 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.

- 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.

```
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).

- The syntax for directive `EXECNOW` is the following.

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

Files after `LOG` are log files generated by command `cmd` and compared from oracles, whereas files after `BIN` are binary files also generated by `cmd` but not compared from oracles. Full access path to these files have to be specified only in `cmd`. All the commands described by directives `EXECNOW` are executed in order and before running any of the other directives. If the execution of one `EXECNOW` directive fails (*i.e.* has a non-zero return code), the remaining actions are not executed. `EXECNOW` directives from a given level are added to the directives of the following levels.

- 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.

4.7 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 developpers 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 4.8). Second it allows to define plug-in specific parameters available as options on the Frama-C command line to the end-user (see Section 4.13).

4.8 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 to be difficult to handle in a readable way. As soon as your plug-in is registered (see Section 4.7 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:*

```

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 high 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.

4.8.1 From printf to Log

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

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

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

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

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

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

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

4.8.2 Log Quick Reference

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

result *<options>* "..."

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

feedback *<options>* "..."

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

debug *<options>* "..."

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

warning *<options>* "..."

For reporting to the user an important information about the validity of the analysis

performed by your plug-in. For instance, if you locally assume non arithmetic overflow on a given statement, *etc.* Typical options include `~current:true` to localize the message on the current source location.

error *<options>* "..."

abort *<options>* "..."

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

The abort routine is a variant that raises an exception and thus aborts the computation.

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.

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)
```

4.8.3 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 output can be associated to a debugging key with the optional argument `~dkey` which takes a string as argument. This can be used in addition to the level of debugging. User can then choose to output debugging messages belonging to a given category with the `-plugin-debug-category <category>` option. A message will be output if either the appropriate level is set or its category is selected. Note that this part of the API is still quite experimental and may evolve at some point.

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

`~source:s` use the source location *s* (see `Log.mli`)

`~current:true` use the current source location managed by `Cil.CurrentLoc`.

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 4.8.4 below for more details on how to globally modify such a behavior. During the message construction, you can locally modify the emission process with the following options:

`~emitwith:f` suppresses the echo and sends the emitted event *only* to the callback function *f*. Listeners are not fired at all.

`~once:true` finally discards the message if the same one was already emitted before with the `~once` option.

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

4.8.4 Advanced Logging Services

Message Emission

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

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

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

Continuations

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

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

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

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


Generic Routines

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

log *?kind ?verbose ?debug <options> "..."*

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

with_log f ?kind <options> "..."

Emits a message like `log`, and finally pass the generated message to the continuation *f*, and returns its result.

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

<code>log "..."</code>	verbosity is at least 1
<code>log ~verbose:n</code>	verbosity is at least <i>n</i>
<code>log ~debug:n</code>	debugging is at least <i>n</i>
<code>log ~verbose:v ~debug:d</code>	<i>either</i> verbosity is at least <i>v</i> <i>or</i> debugging is at least <i>d</i> .

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.

4.9 The Type library: Type Values and Datatypes

Type values and *datatypes* are key notions of Frama-C. They are both provided by the `Type` library. An overview as well as technical details may also be found in a related article in French [19]. A short summary focusing on (un)marshaling is described in another article [7]. First, Section 4.9.1 introduces type values. Then Section 4.9.2 introduces datatypes built on top of type values.

4.9.1 Type Value

A *type value* is an OCaml value which dynamically represents a static monomorphic OCaml type τ . It gets the type τ `Type.t`. There is at most one type value which represents the type τ . Type values are used by Frama-C to ensure safety when dynamic typing is required (for instance to access to a dynamic plug-in API, see Section 4.10.2).

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 Type.t` while the latter has type `string Type.t`.*

Type values are created when building datatypes (see Section 4.9.2). 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 → char → bool` is `Datatype.func2 Datatype.stringring Datatype.char Datatype.bool`. The former has type `int list Type.t` while the latter has type `(string → char → bool) Type.t`.*

4.9.2 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 τ , this extended signature additionally provides modules `Set`, `Map` and `Hashtbl` respectively implementing sets over τ , maps and hashtables indexed by elements of τ .

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

Example 4.13 *The following line of code pretty prints whether two statements are equal.*

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

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

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

Building Datatypes

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

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

```

type ab = A | B of int
module AB =
  Datatype.Make
  (struct
    (* the type corresponding to the datatype *)
    type t = ab
    (* the unique name of the built datatype; usually the name of the
       type *)
    let name = "ab"
    (* represents 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 *)
    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.Undefined
    let equal (x:t) y = x = y
  end)
```

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 variable and the functor `Datatype.Polymorphic2` for types with two type variables. These functors takes 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.*

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

```

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

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

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

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

```

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

4.10 Plug-in Registration and Access

In this section, we present how to register plug-ins and how to access them. Actually there are two different ways to register plug-ins depending on whether they are kernel-integrated or not (cf Section 3.4).

Section 4.10.1 indicates how to register and access a *kernel-integrated* plug-in while Section 4.10.2 details how to register and access a *standard* plug-in.

4.10.1 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 4.10.2 for details).*

A database, called `Db` (in directory `src/kernel`), 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 4.18 *Plug-in Impact registers function `compute_pragmas` in the following way.*

File `src/impact/register.ml`

```

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

```

So 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.

Recommendation 4.2 *The suggested name for this directory is `p_types` for a plug-in `p`.*

If you add such a directory, you also have to modify `Makefile` by extending variable `UNPACKED_DIRS` (see Section 5.3.3).

Example 4.19 *Suppose you are writing a plug-in `plug-in` which exports a specific type `t` corresponding to the result of the plug-in analysis. The standard way to proceed is the following.*

File `src/plugin_types/plugin_types.mli`

```
| type t = ...
```

File `src/kernel/db.mli`

```
module Plugin : sig
  val run_and_get: (unit → Plugin_types.t) ref
  (** Run plugin analysis (if it was never launched before).
      @return result of the analysis. *)
end
```

File `Makefile`

```
UNPACKED_DIRS= ... plugin_types
# Extend this variable with the new directory
```

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 `Db`⁵.

To this effect, the files containing the exterior plug-in code must be added to the `Makefile` variable `PLUGIN_TYPES_CMO` (see Section 5.3.3).

4.10.2 Dynamic Registration and Access

Target readers: *standard plug-ins developers.*

⁵A direct consequence is that you cannot use the whole Frama-C functionalities, such as module `Db`, inside this code.

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

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

Registering a value

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

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

The first argument is the name of the plug-in registering the value and the second one is a binding name of the registered OCaml value. The pair (plug-in name, binding name) must not be used for value registration anywhere else in `Frama-C`. It is required for another plug-in in order to access to this value (see next paragraph). The third argument is the *type value* of the registered value (see Section 4.9.1). 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 4.11). The usual value for this argument is `true`. The fifth argument is the value to register.

Example 4.20 *Here is how the function `run` of the plug-in `hello` of the tutorial is registered. The type of this function is `unit → unit`.*

```
let run () : unit = ...
let () =
  Dynamic.register
    ~plugin:"Hello"
    "run"
    (Datatype.func Datatype.unit Datatype.unit)
    ~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 4.14).

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

Accessing to a registered value

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

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

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

Example 4.21 *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 to access to the type value corresponding to the type of the registered value. Thus it is not possible to access to a value of a plug-in-defined type. For solving this issue, Frama-C provides a way to access to type values of plug-in-defined types in an abstract way through the functor `Type.Abstract`.

Example 4.22 *There is no current example in the Frama-C open-source part, but consider a plug-in which provides a dynamic API for callstacks as follows.*

```

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

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

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

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

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

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

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

```

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

```

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

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

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

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

(* operations over this mutable callstack *)

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

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

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

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

```

4.11 Journalization

Not written yet: please report as “feature request” on <http://bts.frama-c.com> if you really need this section.

4.12 Project Management System

Prerequisite: *knowledge of OCaml module system and labels.*

In Frama-C, a key notion detailed in this section is the one of *project*. An overview as well as technical details may also be found in a related article in French [18]. Section 4.12.1 first introduces the general principle of project. Section 4.12.2 introduces the notion of *states*. State registration is detailed in Sections 4.12.3 and 4.12.4. The former is dedicated to standard (high-level) registration, while the latter is dedicated to low-level registration. Then Section 4.12.5 explains how to use project. Finally Section 4.12.6 details state selections.

4.12.1 Overview and Key Notions

A *project* group together an AST with the set of global values attached to it. Such values are called *states*. Examples of states are parameters (see Section 4.13), results of analyses (Frama-C extensively uses memoization [15, 16] in order to prevent running analysis twice).

In a **Frama-C** session, several project (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 project in any other project. For ensuring this property, each state must be registered in the project library as explained in Sections 4.12.3 and 4.12.4. Such relation between states and projects are summarized in Figure 4.3.

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

Figure 4.3: Representation of the **Frama-C** State.

To ease development, **Frama-C** maintains a current project (`Project.current ()`): all operations are automatically performed on. For instance, calling `Ast.get ()` returns the **Frama-C** AST of the current project. It is also possible to access to values in others projects as explained in Section 4.12.5.

4.12.2 State: Principle

If a data should be part of the state of **Frama-C**, you must register it in the project library (see Sections 4.12.3 and 4.12.4).

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 a 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 provide to each internal state are the following.

- It is automatically updated whenever the current project changes: so your data is 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 this 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.
- It is part of the information saved on disk for restoration in a later session.
- It may be part of a *selection* which is consistent set of states. Which such a selection, you can control on which states project operations are consistently applied (see Section 4.12.6). For example, it is possible to clear all the states which depend on the value analysis' results.
- It is possible to ensure inter-analysis consistency by setting state dependencies. For example, if the entry point of the analysed program is changed (using

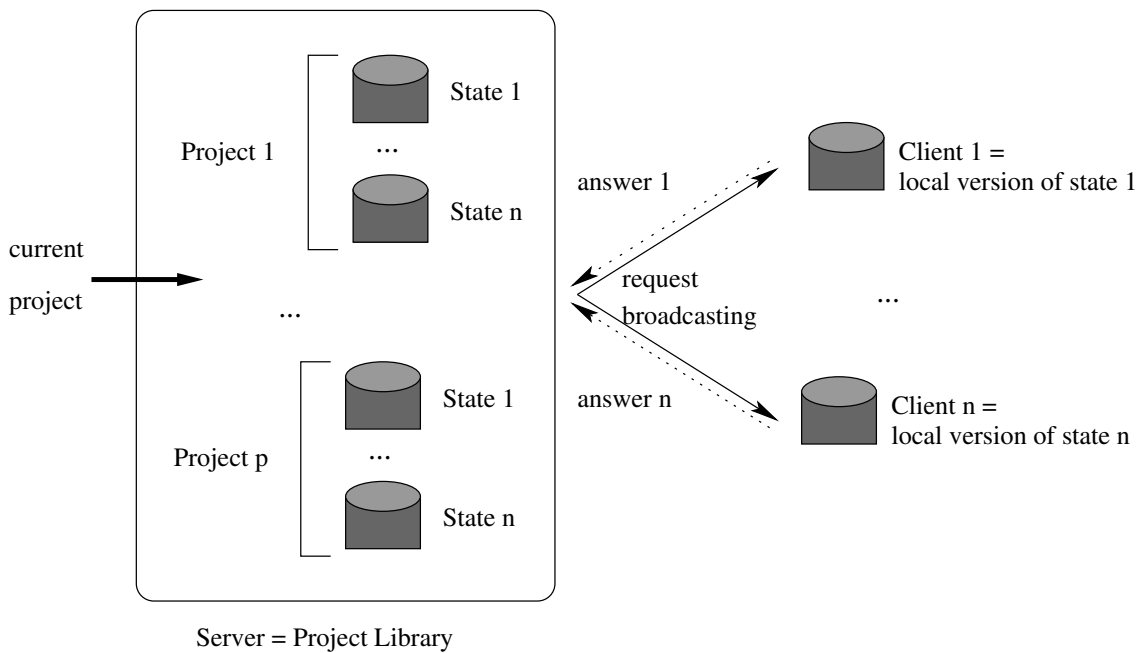


Figure 4.4: Interaction between the project library and your registered global data.

`Globals.set_entry_point`), all the results of analyses depending on it (like value analysis' results) are automatically reset. If such a reset was not performed, the results of the value analysis would not be consistent anymore with the current entry point, leading to incorrect results.

Example 4.23 *Suppose that the value analysis has previously been computed.*

```
Format.printf "%B0." (Db.Value.is_computed ()); (* true *)
Globals.set_entry_points "f" true;
Format.printf "%B0." (Db.Value.is_computed ()); (* false *)
```

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

```
| true
| false
```

State Registration: Overview

For registering a new state, functor `State_builder.Register` is provided. Its use is described in Section 4.12.4 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)⁶. They are described in Section 4.12.3.

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

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).

4.12.3 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 4.12.4 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 in 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 codew.

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

```
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?

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 an 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 4.9.2). 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 4.9.1).

- From outside, a state actually hides its internal representation in order to ensure some invariants: operations on states implementing hashtable does not take an hashtable in argument 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 to a state kind before defining it. That is usually the case when you have two mutually-dependent states: the dependencies of the first one providing 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 4.14 for advanced explanation about the way Frama-C is initialized).

Example 4.24 *Plug-in from puts a reference to its state kind in the following way. This reference is initialized at module initialization time.*

File `src/kernel/db.mli`

```
module From = struct
  ...
  val self: State.t ref
end
```

File `src/kernel/db.ml`

```
module From = struct
  ...
  val self = ref State.dummy (* postponed *)
end
```

File `src/from/functionwise.ml`

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

File `src/pdg/register.ml`

```
module Tbl =
  Kernel_function.Make_Table
  (PdgTypes.Pdg)
```

```

(struct
  |let name = "Pdg.State"
  |let dependencies = [] (* postponed because !Db.From.self may
                          not exist yet *)

  |let size = 97
end)
|let () =
  Cmdline.run_after_extended_stage
  (fun () →
    State_dependency_graph.add_codedependencies
    ~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 state, while `Ast.mark_as_changed` will clear it.

4.12.4 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 is only three direct uses of this functor over thousands 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 4.12.3).

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 purpose (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

$$\text{create } () \text{ returns a fresh value} \quad (4.1)$$

$$\forall p \text{ of type } t, \text{ create } () = (\text{clear } p; \text{ set } p; \text{ get } ()) \quad (4.2)$$

$$\forall p \text{ of type } t, \text{ copy } p \text{ returns a fresh value} \quad (4.3)$$

$$\forall p_1, p_2 \text{ of type } t \text{ such that } p_1 \neq p_2, (\text{set } p_1; \text{ get } ()) \neq p_2 \quad (4.4)$$

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 criteria which ensures that modifying a local version does not affect any other version (different of the global current one) by side-effect.

Example 4.25 *To illustrate this, we show how functor `State_builder.Ref` (registering a state corresponding to a reference) is implemented.*

```

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

```

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

```

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

```

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

```

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

```

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

4.12.5 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 4.15) or replaced (e.g. if a new one is loaded from disk).

⁷As usual in OCaml, `=` stands for *structural* equality while `==` (resp. `!=`) stands for *physical* equality (resp. disequality).

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 4.12.6):

```
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 on the new current project.

Example 4.26 *Suppose that you saved the current project into file `foo.sav` in a previous Frama-C session⁸ 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 () = Format.printf "%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 *)
  Project.set_current old;
  print_computed () (* false *)
with Project.IOError _ →
  Kernel.abort "error while loading"
```

displays

```
| false
| true
| 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 its 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, after, restore the initial context).

Example 4.27 *The following code is equivalent to the one given in Example 4.26.*

```
| let print_computed () = Format.printf "%b@." (Db.Value.is_computed ()) in
| print_computed (); (* false *)
| try
```

⁸A *session* is one execution of Frama-C (through `frama-c` or `frama-c-gui`).

```

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

```

4.12.6 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 4.28 *The following statement clears all the results of the value analysis and all its dependencies in the current project.*

```

let selection = State_selection.with_dependencies Db.Value.self in
Project.clear ~selection ()

```

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

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

Example 4.29 *The following statement applies a function f in the project p (which is not the current one). For efficiency purpose, we restrict the considered states to the command line options (see Section 4.13).*

```

Project.on ~selection:(Plugin.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.

4.13 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.

4.13.1 Definition

In Frama-C, a parameter is represented by a value of type `Parameter.t` and by a module implementing the signature `Plugin.Parameter`. The first representation is a low-level one required by emitters (see Section 4.16). The second one provides a high-level API: each parameter is indeed a state (see Section 4.12.2). Several signatures extending `Plugin.Parameter` are provided in order to deal with the usual parameter types. For example, there are signatures `Plugin.Int` and `Plugin.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 eventually the wished default value. Below are some examples of such functors (see the signature `Plugin.General_services` 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. `EmptyString` `EmptyString`) builds a string option initialized to a specified value (resp. to the empty string `""`).
4. `StringSet` builds an option taking a set of strings 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 4.30 *The parameter corresponding to the option `-occurrence` of the plug-in `occurrence` is the module `Print` (defined in the file `src/occurrence/options.ml`). It is implemented as follow.*

```

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

```

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

```

module Print : Plugin.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/impact/options.ml`). It is implemented as follow.

```

module Pragma =
  StringSet
  (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)

```

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

```
| module Pragma: Plugin.String_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 4.31 *To know whether Frama-C uses unicode, just write*

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

Inside the plug-in `From`, just write

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

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

Using a parameter of a plug-in p in another plug-in p' requires the use of module `Dynamic.Parameter`: since the module defining the parameter is not visible from the outside of its plug-in, you have to use the dynamic API of plug-in p in which p 's parameters are automatically registered (see Section 4.10.2). The module `Dynamic.Parameter` defines sub-modules which provide easy access to parameters according to their OCaml types.

Example 4.32 *Outside the plug-in `From`, just write*

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

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

4.13.2 Tuning

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

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

Example 4.33 *Here is how the option `"-slicing-level"` restricts the range of its argument to the interval $[0; 3]$.*

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

Functions which can be applied before applying the functor are defined at top of module `Plugin`.

Example 4.34 Here is how the opposite of option `"-safe-arrays"` is renamed into `"-unsafe-arrays"` (otherwise, by default, it would be `"-no-safe-arrays"`).

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

4.14 Initialization Steps

Prerequisite: *knowledge of linking of OCaml files.*

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

This section details the different stages of the Frama-C boot process to help advanced plug-in developers interact more deeply with the kernel process. It can be also 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. 33) 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 can not 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 4.11), 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`;
 - (b) the hooks registered through `Cmdline.run_during_extending_stage` are executed. Such hooks include kernel function calls for searching, loading and linking the various plug-ins, 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 4.12.3).

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 informations:
 - (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 4.11). As like 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 states 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.

4.15 Visitors

Prerequisite: *knowledge of OCaml object programming.*

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 4.15.6, which the reader already familiar with Cil is invited to read carefully.

4.15.1 Entry Points

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.

4.15.2 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`⁹, and has type `astType → astType' visitAction`, where `astType'` is either `astType` or `astType list` (for instance, one can transform a `global` into several ones). `visitAction` describes what should be done for the children of the resulting AST node, and is presented in the next section. In addition, some types have two modes of visit: one for the declaration and one for use. This is the case for `varinfo` (`vvdec` and `vvrbl`), `logic_var` (`vlogic_var_decl` and `vlogic_var_use`) `logic_info` (`vlogic_info_decl` and `vlogic_info_use`), `logic_type_info` (`vlogic_type_info_decl` and `vlogic_type_info_use`), and `logic_ctor_info` (`vlogic_ctor_info_decl` and `vlogic_ctor_info_use`). More detailed information can be found in `cil.mli`.

For the Frama-C visitor, two methods, `vstmt` and `vglob` take care of maintaining the coherence between the transformed AST and the internal state of Frama-C. Thus they must not be redefined. One should redefine `vstmt_aux` and `vglob_aux` instead.

4.15.3 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.

⁹This naming convention is not strictly enforced. For instance the method corresponding to `offset` is `voffs`.

- `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.

4.15.4 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).

4.15.5 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.

Type	Index
varinfo	vid
compinfo	ckey
enuminfo	ename
typeinfo	tname
stmt	sid
logic_info	l_var_info.lv_id
logic_var	lv_id
fieldinfo	fname and fcomp.ckey

Figure 4.5: 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)`.

4.15.6 Differences Between the Cil and Frama-C Visitors

As said in Section 4.15.2, `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 4.15.3), 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.

4.15.7 Example

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

```
open Cil_types
open Cil

module M = Plugin.Register

(* Each annotation in Frama-C has an emitter, for traceability.
   We create thus our own, and says that it will only be used to emit code
   annotations, and that these annotations do not depend on Frama-C's command
   line parameters.
*)
let syntax_alarm =
  Emitter.create
    "Syntactic check" [ Emitter.Code_annot ] ~correctness:[] ~tuning:[]
```

```

class non_zero_divisor prj = object (self)
  inherit Visitor.generic_frama_c_visitor (Cil.copy_visit 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 = get_stmt self#behavior stmt in
    let new_kf = get_kernel_function self#behavior kf in
    (* Since we are copying the file in a new project, we cannot insert
       the annotation into the current table, but in the table of the new
       project. To avoid the cost of switching projects back and forth,
       all operations on the new project are queued until the end of the
       visit, as mentioned above. This is done in the following statement. *)
    Queue.add
      (fun () →
        Annotations.add_assert syntax_alarm ~kf:new_kf new_stmt assertion)
      self#get_filling_actions;
    DoChildren
  | _ → DoChildren
end

(* This function creates a new project initialized with the current file plus
   the annotations related to division. *)
let create_syntactic_check_project () =
  ignore
    (File.create_project_from_visitor "syntactic check" (new non_zero_divisor))

let () = Db.Main.extend create_syntactic_check_project

```

4.16 Logical Annotations

Prerequisite: *Nothing special (apart of core OCaml programming).*

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 4.12.5).

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 annotations. Modifying a property statuses requires to define an emitter by `Emitter.create` first. Key concepts and theoretical foundation of this module are described in an associated research paper [6].
- Modules `Logic_const`, `Logic_utils` and `Db.Properties` contain several operations over annotations.

4.17 Locations

Prerequisite: *Nothing special (apart of core OCaml programming).*

In Frama-C, different representations of C locations exist. Section 4.17.1 presents them. Moreover, maps indexed by locations are also provided. Section 4.17.2 introduces them.

4.17.1 Representations

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

- Type `Location_Bytes.t` is used to represent values of C expressions like `2` or `((int) &a) + 13`. With this representation, there is no way to know the size of a value while it is still possible to join two values. Roughly speaking it is represented by a mapping between C variables and offsets in bytes.
- Type `location`, equivalently `Location.t` is used to represent the right part of a C affectation (including bitfields). It is represented by a `Location_Bits.t` (see below) attached to a size. It is possible to join two locations *if and only if they have the same sizes*.
- Type `Location_Bits.t` is similar to `Location_Bytes.t` with offsets in bits instead of bytes. Actually it should only be used inside a location.
- Type `Zone.t` is a set of bits (without any specific order). It is possible to join two zones *even if they have different sizes*.

Recommendation 4.4 *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 `Dataflow`, see Section 5.1.1).

4.17.2 Map Indexed by Locations

Modules `Lmap` and `Lmap_bitwise` provide functors implementing maps indexed by locations and zones (respectively). The argument of these functors have to implement values attached to indices (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 structures implementing values attached to indices are lattices (*i.e.* implement signature `Abstract_interp.Lattice`). For this purpose, functors of the abstract interpretation toolbox can help (see in particular module `Abstract_interp`).

4.18 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` \rightarrow `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 4.19).

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 5.3.3).

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` [12]: 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/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.

4.19 Documentation

Prerequisite: *knowledge of ocamlDoc.*

Here we present some hints on the way to document your plug-in. First Section 4.19.1 introduces a quick general overview about the documentation process. Next Section 4.19.2 focuses on the plug-in source documentation. Finally Section 4.19.3 explains how to modify the Frama-C website.

4.19.1 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.

4.19.2 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.

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

Plug-in API A plug-in should export no function itself: the only visible plug-in interface should be in `Db` or registered through module `Dynamic` as explained in Section 4.10.2.

Recommendation 4.5 *To ensure this invariant, the best way is to provide an empty interface for the plug-in.*

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. If you declare such an interface, you also have to set the variable `PLUGIN_HAS_MLI` in your `Makefile` (see Section 5.3.3).

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 5.3.3).

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.

4.19.3 Website

Target readers: *CEA developers with a SVN access.*

Read private files `README` and `metadoc.NE_PAS_LIVRER` in the SVN directory `doc/www/src`.

4.20 License Policy

Target readers: *developers with a SVN access.*

Prerequisite: *knowledge of make.*

If you want to redistribute a plug-in inside Frama-C, you have to define a proper license policy. For this purpose, some help is provide in the `Makefile`. Mainly we distinguish two cases described below.

- **If the wished license is already used inside Frama-C**, just extend the variable corresponding to the wished license in order to include files of your plug-in. Next run `make headers`.

Example 4.35 *Plug-in slicing is released under LGPL and is proprietary of both CEA and INRIA. So, in the makefile, there is the following line.*

```
CEA_INRIA_LGPL= ... \
                src/slicing_types/*.ml* src/slicing/*.ml*
```

- **If the wished license is unknown inside Frama-C**, you have to:
 1. Add a new variable *v* corresponding to it and assign files of your plug-in;
 2. Extend variable `LICENSES` with this variable;
 3. Add a text file in directory `licenses` containing your licenses
 4. Add a text file in directory `headers` containing the headers to add into files of your plug-in (those assigned by *v*).

The filename must be the same than the variable name *v*. Moreover this file should contain a reference to the file containing the whole license text.

5. Run `make headers`.



Reference Manual

Target readers: *Developers which need to understand some Frama-C internal details.*

This chapter is a reference manual for Frama-C developers. It provides details completing Chapter 4.

5.1 File Tree

This Section introduces the main parts of Frama-C in order to quickly find useful information inside sources. Our goal is *not* to introduce the Frama-C software architecture (that is the purpose of Chapter 3) nor to detail each module (that is the purpose of the source documentation generated by `make doc`). The directory containing the Cil implementation is detailed in Section 5.1.1 while the directory containing the Frama-C implementation itself is presented in Section 5.1.2.

Figure 5.1 shows all directories useful to plug-in developers. More details are provided below.

Kind	Name	Specification	Reference
	.	Frama-C root directory	
Sources	src cil ocamlgraph external	Frama-C implementation Cil source files OcamlGraph source files Source of external free libraries	Section 5.1.2 Section 5.1.1
Tests	tests ptest	Frama-C test suites ptest implementation	Section 4.6
Generated Files	bin lib	Binaries Some compiled files	
Documentations	doc headers licenses man	Documentation directory Headers of source files Licenses used by plug-ins and kernel Man pages	Section 4.20 Section 4.20
Shared libraries	share	Shared files	

Figure 5.1: Frama-C directories.

- The **Frama-C** root directory contains the configuration files, the main **Makefile** and some information files (in uppercase).
- **Frama-C** sources are split in four directories: **src** (described in Section 5.1.2) contains the core of the implementation while **cil** (described in Section 5.1.1), **ocamlgraph** and **external** respectively contains the implementation of Cil (extended with ACSL), a version of the **OcamlGraph** library [3] compatible within **Frama-C**, and external libraries included in the **Frama-C** distribution.
- The directory **tests** contains the **Frama-C** test suite which is used by the **ptests** tool (see Section 4.6).
- Directories **bin** and **lib** contain binary files mainly produced by **Frama-C** compilation. **Frama-C** executables belong to the directory **bin**, the directories **lib/plugins** and **lib/gui** receive the compiled plug-ins, and the directory **lib/fc** receives the compiled kernel interface. You should never add yourself any file in these directories.
- Documentation (including plug-in specific, source code and ACSL documentations) is provided in directory **doc**. Directories **headers** and **licenses** contains files useful for copyright notification (see Section 4.20).
- Directory **share** contains useful libraries for **Frama-C** users such as the **Frama-C** C library (*e.g.* ad-hoc libraries such as **libc** and **malloc** for **Frama-C**), as well as user-oriented **Makefiles**.

5.1.1 The cil directory

The source files of Cil belong to the five directories shown in Figure 5.2. More details are provided below.

Name	Specification
src	Main Cil files
src/ext	Syntactic analysis provided by Cil
src/frontc	C frontend
src/logic	ACSL frontend

Figure 5.2: Cil directories.

- **src** contains the main files of Cil. The most important modules are **Cil_types** and **Cil**. The first one contains type declarations of the Cil AST while the second one contains useful operations over this AST.
- **src/ext** contains syntactic analysis provided by Cil. For example, module **Cfg** provides control flow graph, module **Callgraph** provides a syntactic callgraph and module **Dataflow** provides parameterized forward/backward data flow analysis.
- **src/frontc** is the C frontend which converts C code to the corresponding Cil AST. It should not be used by a **Frama-C** plug-in developer.

- `src/logic` is the ACSL frontend which converts logic code to the corresponding Cil AST. The only useful modules for a Frama-C plug-in developer are `Logic_const` which provides some predefined logic constructs (terms, predicates, ...) and `Logic_typing` which allows to dynamically extend the logic type system.

5.1.2 The `src` directory

The source files of Frama-C are split into different sub-directories inside `src`. Each sub-directory contains either a plug-in implementation or some specific parts of the Frama-C kernel.

Each plug-in implementation can be split into two different sub-directories, one for exported type declarations and related implementations visible from `Db` (see Chapter 3 and Section 4.10.1) and another for the implementation provided in `Db`.

Kernel directories are shown Figure 5.3. More details are provided below.

Kind	Name	Specification	Reference
Toolboxes	<code>kernel</code>	Kernel toolbox	
	<code>logic</code>	Logic toolbox	
	<code>ai</code>	Abstract interpretation toolbox	Section 4.17
	<code>memory_states</code>	Memory-state toolbox	Section 4.17
Libraries	<code>type</code>	Type library	Section 4.9
	<code>project</code>	Project library	Section 4.12
	<code>printer</code>	AST printer library	
	<code>lib</code>	Miscellaneous libraries	
	<code>misc</code>	Additional useful operations	
Entry points	<code>gui</code>	Graphical User Interface	Section 4.18

Figure 5.3: Kernel directories.

- Directory `kernel` contains the kernel toolbox over Cil. Main kernel modules are shown in Figure 5.4.
- Directory `logic` is the logic toolbox. It contains modules helping to handle logical annotations and their status.
- Directories `ai` and `memory_states` are the abstract interpretation and memory-state toolboxes (see section 4.17). In particular, in `ai`, module `Abstract_interp` defines useful generic lattices and module `Ival` defines some pre-instantiated arithmetic lattices while, in `memory_states`, module `Locations` provides several representations of C locations and modules `Lmap` and `Lmap_bitwise` provide maps indexed by such locations.
- Directories `type` and `project` contain the type library and the project library respectively described in details in Sections 4.9 and 4.12.
- Directories `lib` and `misc` contain datastructures and operations used in Frama-C. In particular, module `Extlib` is the Frama-C extension of the OCaml standard library whereas module `Type` is the interface for type values (the OCaml values representing OCaml types) required by dynamic plug-in registrations and uses and journalization (see Section 4.9).

Kind	Name	Specification	Reference
AST	<code>Ast</code> <code>Ast_info</code>	The Cil AST for Frama-C Operations over the Cil AST	
Global tables	<code>File</code> <code>Globals</code> <code>Kernel_function</code> <code>Annotations</code> <code>Emitter</code> <code>Loop</code>	AST creation and access to C files Operations on globals Operations on functions Operations on ACSL annotations Emitter of property statuses Operations on loops	Section 4.16 Section 4.16
Plug-in APIs	<code>Db</code> <code>Dynamic</code>	Static plug-in database Interface for dynamic plug-ins	Section 4.10.1 Section 4.10.2
Base Modules	<code>Config</code> <code>Plugin</code> <code>Parameter</code> <code>Cmdline</code> <code>Log</code> <code>Kernel</code> <code>Journal</code> <code>CilE</code> <code>Alarms</code> <code>Stmts_graph</code> <code>Floating_point</code>	Information about Frama-C version General services for plug-ins Parameter as command line options Command line parsing Printing messages Kernel as a plug-in Journalization Useful Cil extensions Alarm management Accessibility checks using CFG Floating-point operations	Section 4.7 Section 4.13 Section 4.8 Section 4.13 Section 4.11
Visitor	<code>Visitor</code>	Frama-C visitors (subsume Cil ones)	Section 4.15
System	<code>Command</code> <code>Task</code>	System operations Higher-level API than <code>Command</code>	
Initializer	<code>Boot</code> <code>Gui_init</code> <code>Special_hooks</code>	Last linked module Very early initialization of the GUI Registration of some kernel hooks	Section 4.14 Section 4.14

Figure 5.4: Main kernel modules.

- Directory `gui`¹ contains the `gui` implementation part common to all plug-ins. See Section 4.18 for more details.

5.2 Configure.in

Figure 5.5 presents the different parts of `configure.in` in the order that they are introduced in the file. The second column of the tabular says whether the given part has to be modified eventually by a kernel-integrated plug-in developer. More details are provided below.

Id	Name	Mod.	Reference
1	Configuration of <code>make</code>	no	
2	Configuration of <code>OCaml</code>	no	
3	Configuration of mandatory tools/libraries	no	
4	Configuration of non-mandatory tools/libraries	no	
5	Platform configuration	no	
6	Wished <code>Frama-C</code> plug-in	yes	Sections 4.2.2 and 4.2.4
7	Configuration of plug-in tools/libraries	yes	Section 4.2.3
8	Checking plug-in dependencies	yes	Section 4.2.5
9	Makefile creation	yes	Section 4.2.2
10	Summary	yes	Section 4.2.2

Figure 5.5: 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).

¹From the outside, the GUI may be seen as a plug-in with some exceptions.

9. **Makefile Creation** creates `Makefile` from `Makefile` including information provided by this configuration.
10. **Summary** displays summary of each plug-in availability.

5.3 Makefiles

In this section, we detail the organization of the different Makefiles existing in Frama-C. First Section 5.3.1 presents a general overview. Next Section 5.3.2 details the different sections of `Makefile.config.in`, `Makefile.common` and `Makefile`. Next Section 5.3.3 introduces the variables customizing `Makefile.plugin` and `Makefile.dynamic`. Finally Section 5.3.4 shows specific details of `Makefile.dynamic`.

5.3.1 Overview

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

- `Makefile`: the general Makefile of Frama-C;
- `Makefile.config.in`: the Makefile configuring some general variables (especially the ones coming from `configure`);
- `Makefile.common`: the Makefile providing some other general variables and general rules;
- `Makefile.plugin`: the Makefile introducing specific stuff for plug-in compilation;
- `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`.
- `Makefile.kernel` is automatically generated from `Makefile`. It contains several variables useful for linking a plug-in against the Frama-C kernel.

The first one is 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 5.6 shows these relationship. `Makefile` 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 and rules are defined in `Makefile.common` (which includes `Makefile.config.in`). `Makefile.plugin` defines generic rules and variables for compiling plug-ins. It is used both by `Makefile` for kernel-specific plug-ins integrated compiled from the Frama-C Makefile and by `Makefile.dynamic` for plug-ins with their own Makefiles.

5.3.2 Sections of `Makefile`, `Makefile.config.in` and `Makefile.common`

Figure 5.7 presents the different parts of `Makefile.config.in`, `Makefile.common` 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.

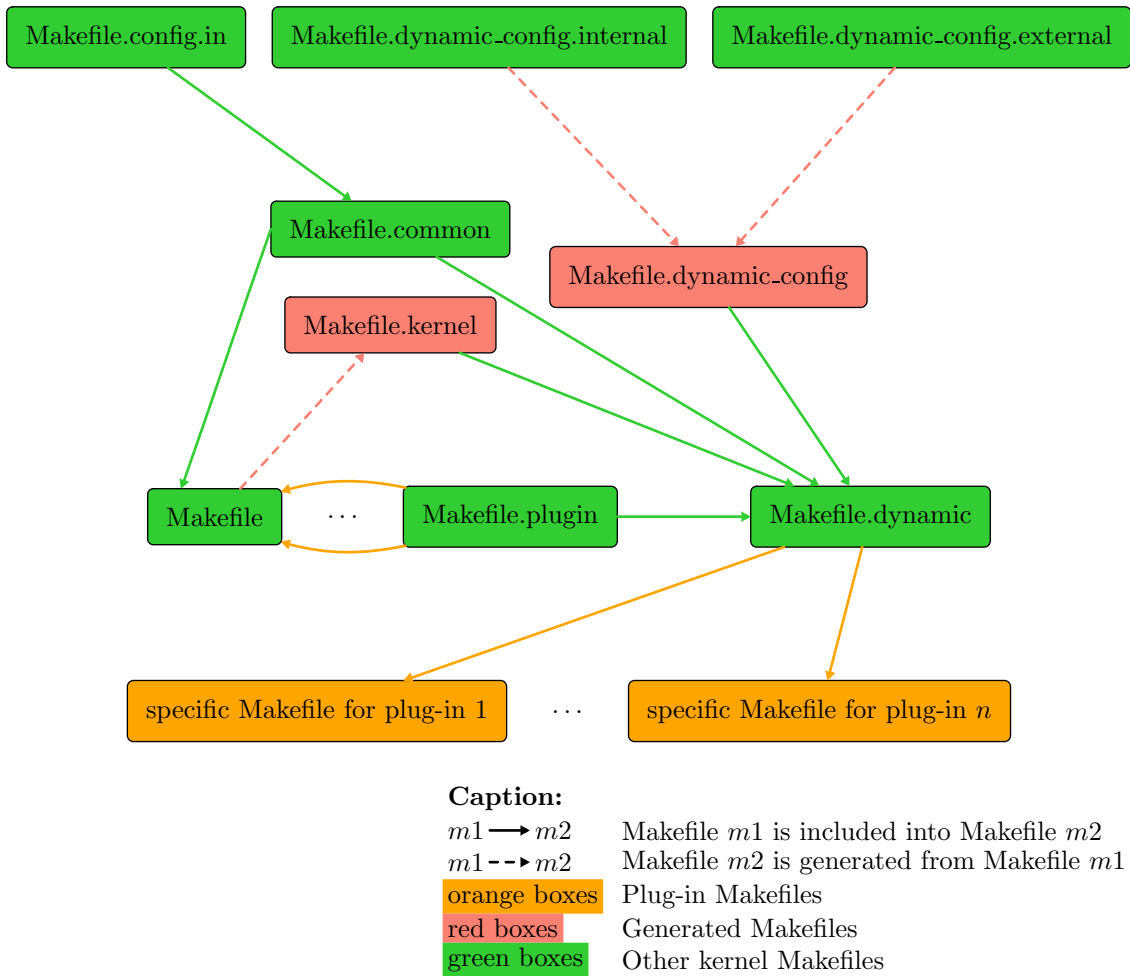


Figure 5.6: Relationship between the Makefiles

Id	Name	File	Mod.	Reference
1	Working directories	Makefile.config.in	no	
2	Installation paths	Makefile.config.in	no	
3	Ocaml stuff	Makefile.config.in	no	
4	Libraries	Makefile.config.in	no	
5	Miscellaneous commands	Makefile.config.in	no	
6	Miscellaneous variables	Makefile.config.in	no	
7	Variables for plug-ins	Makefile.config.in	no	
1 (bis)	Working directories	Makefile.common	no	r
8	Flags	Makefile.common	no	
9	Verbosing	Makefile.common	no	
10	Shell commands	Makefile.common	no	
11	Command pretty printing	Makefile.common	no	
12	Tests	Makefile.common	no	
13	Generic rules	Makefile.common	no	
14	Global plug-in variables	Makefile	no	
15	Additional global variables	Makefile	no	
16	Main targets	Makefile	no	
17	Coverage	Makefile	no	
18	Ocamlgraph	Makefile	no	
19	Frama-C Kernel	Makefile	no	
20	Plug-in sections	Makefile	yes	Section 4.4
21	Generic variables	Makefile	no	
22	Toplevel	Makefile	no	
23	GUI	Makefile	no	
24	Standalone obfuscator	Makefile	no	
25	Tests	Makefile	no	
26	Emacs tags	Makefile	no	
27	Documentation	Makefile	no	
28	Installation	Makefile	yes	<i>Not written yet.</i>
29	File headers: license policy	Makefile	yes	Section 4.20
30	Makefile rebuilding	Makefile	no	
31	Cleaning	Makefile	no	
32	Depend	Makefile	no	
33	ptests	Makefile	no	
34	Source distribution	Makefile	no	

Figure 5.7: Sections of `Makefile.config.in`, `Makefile.common` and `Makefile`.

1. **Working directories** (splitted between `Makefile.config.in` and `Makefile.common` defines the main directories of Frama-C. In particular, it declares the variable `UNPACKED_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 5.3.3).
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


```
| $ 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`.

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

Executing

```
| $ make foo
```

prints

```
| Copying to foo
```

while executing

```
| $ make foo VERBOSEMAKE=yes
```

prints

```
| cp -f bar foo
```

If one of the two commands is missing for the target `foo`, either `make foo` or `make foo VERBOSEMAKE=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[ix]` from `.ml` or `.mli` for OCaml files)

14. **Global plug-in variables** declares some plug-in specific variables used throughout the makefile.
15. **Additional global variables** declares some other variables used throughout the makefile.
16. **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.
17. **Coverage** defines how compile the eponymous library.
18. **Ocamlgraph** defines how compile the eponymous library.
19. **Frama-C Kernel** provides variables and rules for the **Frama-C** kernel. Each part is described in specific sub-sections.
20. After Section “Kernel”, there are several sections corresponding to **plug-ins** (see Section 5.3.3). This is the part that a plug-in developer has to modify in order to add compilation directives for its plug-in.
21. **Generic variables** provides variables containing files to be linked in different contexts.
22. **Toplevel** provides rules for building the files of the form `bin/toplevel.*`.
23. **GUI** provides rules for building the files of the form `bin/viewer.*`
24. **Standalone obfuscator** provides rules for building the **Frama-C** obfuscator.
25. **Tests** provides rules to execute tests. `make tests` takes care of generating the appropriate environment and launching `ptests` (see Section 4.6) for all test suites of the kernel and enabled plugins. It is possible to pass options to `ptests` through the `PTESTS_OPTS` environment variable.
26. **Emacs tags** provides rules which generate `emacs` tags (useful for a quick search of OCaml definitions).
27. **Documentation** provides rules generating **Frama-C** source documentation (see Section 4.19).
28. **Installation** provides rules for installing different parts of **Frama-C**.
29. **File headers: license policy** provides variables and rules to manage the **Frama-C** license policy (see Section 4.20).
30. **Makefile rebuilding** provides rules in order to automatically rebuild `Makefile` and `configure` when required.
31. **Cleaning** provides rules in order to remove files generated by makefile rules.
32. **Depend** provides rules which compute **Frama-C** source dependencies.
33. **Ptests** provides rules in order to build `ptests` (see Section 4.6).
34. **Source distribution** provides rules usable for distributing **Frama-C**.

5.3.3 Variables of `Makefile.dynamic` and `Makefile.plugin`

Figures 5.8 and 5.9 presents all the variables that can be set before including `Makefile.plugin` or `Makefile.dynamic` (see Sections 4.4 and 4.5). The last column is set to *no* if and only if the line is not relevant for a standard plug-in developer. Details are provided below.

Kind	Name	Specification	
Usual information	PLUGIN_NAME	Module name of the plug-in	
	PLUGIN_DIR	Directory containing plug-in source files	no
	PLUGIN_ENABLE	Whether the plug-in has to be compiled	no
	PLUGIN_DYNAMIC	Whether the plug-in is dynamically linked with Frama-C	no
Source files	PLUGIN_HAS_MLI	Whether the plug-in gets an interface	
	PLUGIN_CMO	.cmo plug-in files	
	PLUGIN_CMI	.cmi plug-in files without corresponding .cmo	
	PLUGIN_TYPES_CMO	.cmo plug-in files not belonging to \$(PLUGIN_DIR)	
Compilation flags	PLUGIN_GUI_CMO	.cmo plug-in files not belonging to \$(PLUGIN_DIR)	
	PLUGIN_BFLAGS	Plug-in specific flags for <code>ocamlc</code>	
	PLUGIN_OFLAGS	Plug-in specific flags for <code>ocamlopt</code>	
	PLUGIN_EXTRA_BYTE	Additional bytecode files to link against	
	PLUGIN_EXTRA_OPT	Additional native files to link against	
	PLUGIN_LINK_BFLAGS	Plug-in specific flags for linking with <code>ocamlc</code>	
PLUGIN_LINK_OFLAGS	Plug-in specific flags for linking with <code>ocamlopt</code>		
PLUGIN_LINK_GUI_BFLAGS	Plug-in specific flags for linking a GUI with <code>ocamlc</code>		
PLUGIN_LINK_GUI_OFLAGS	Plug-in specific flags for linking a GUI with <code>ocamlopt</code>		

Figure 5.8: Standard parameters of `Makefile.dynamic` and `Makefile.plugin`.

- 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/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_DYNAMIC` must be set to `yes` if the plug-in has to be dynamically linked with Frama-C.
- Variable `PLUGIN_HAS_MLI` must be set to `yes` if plug-in *plugin* gets a file `.mli` (which must be capitalized: `Plugin.mli`, see Section 4.19) describing its API. Note that this API should be empty in order to enforce the architecture invariant which is that each plug-in is used through `Db` (see Chapter 3).
- 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 (`$(PLUGIN_DIR)/f.cm[io]` for a file *f*).
- Variable `PLUGIN_TYPES_CMO` is the `.cmo` plug-in files which do not belong to `$(PLUGIN_DIR)`. They usually belong to `src/plugin_types` where *plugin* is the plug-in name (see Section 4.10.1). 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 4.18). 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. Note that, in a normal context, it should not be used because a plug-in interface should be empty (see Chapter 3).
- Variable `PLUGIN_UNDOC` is the source files for which no documentation is provided. Do not write their file path which is automatically set to `$(PLUGIN_DIR)`.
- Variable `PLUGIN_TYPES_TODOC` is the additional source files to document with the plug-in. They usually belong to `src/plugin_types` where *plugin* is the plug-in name (see Section 4.10.1).
- Variable `PLUGIN_INTRO` is the text file to append to the plug-in documentation introduction. Usually this file is `doc/code/intro_plugin.txt` for a plug-in *plugin*. It can contain any text understood by `ocamldoc`.
- Variable `PLUGIN_HAS_EXT_DOC` is set to `yes` if the plug-in has its own reference manual. It is supposed to be a `pdf` file generated by `make` in directory `doc/$(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))`.

Kind	Name	Specification	
Dependencies	PLUGIN_DEPFLAGS PLUGIN_GENERATED PLUGIN_DEPENDS	Plug-in specific flags for <code>ocamldep</code> Plug-in files to compiled before running <code>ocamldep</code> Other plug-ins to compiled before the considered one	no
Documentation	PLUGIN_DOCFLAGS PLUGIN_UNDOC PLUGIN_TYPES_TODOC PLUGIN_INTRO PLUGIN_HAS_EXT_DOC	Plug-in specific flags for <code>ocamldoc</code> Source files with no provided documentation Additional source files to document Text file to append to the plug-in introduction Whether the plug-in has an external pdf manual	
Testing	PLUGIN_NO_TESTS PLUGIN_TESTS_DIRS PLUGIN_TESTS_DIRS_DEFAULT PLUGIN_TESTS_LIBS PLUGIN_NO_DEFAULT_TEST PLUGIN_INTERNAL_TEST PLUGIN_PTESTS_OPTS	Whether there is no plug-in specific test directory tests to be included in the default test suite Directories containing tests Specific <code>.cmo</code> files used by plug-in tests Whether to include tests in default test suite. Whether the test suite of the plug-in is located in Frama-C's own tests directory Plug-in specific options to ptests	
Distribution	PLUGIN_DISTRIBUTED_BIN PLUGIN_DISTRIBUTED PLUGIN_DISTRIB_EXTERNAL	Whether to include the plug-in in binary distribution Whether to include the plug-in in source distribution Additional files to be included in the distribution	no no no

Figure 5.9: Special parameters of `Makefile.dynamic` and `Makefile.plugin`.

- Variable `PLUGIN_TESTS_LIB` is the `.cmo` plug-in specific files used by plug-in tests. Do not write its file path (which is `$(PLUGIN_TESTS_DIRS)`) nor its file extension (which is `.cmo`).
- 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 `p-tests` when running the tests. It comes in addition to `PTESTS_OPTS` (see 5.3.2§25). For instance, `PLUGIN_PTESTS_OPTS=-j 1` will deactivate parallelization of tests in case the plugin does not support concurrent runs. It is only used by `Makefile.dynamic`.
- 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.plugin` 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.plugin` provides some variables which may be used after its inclusion defining `p`. These variables are listed in Figure 5.10. 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`².

5.3.4 Makefile.dynamic

Not written yet: please report as "feature request" on <http://bts.frama-c.com> if you really need this section.

²Variables of the form `p_*CMX` have no `PLUGIN_*CMX` counterpart but their meanings should be easy to understand.

³`plugin` is the module name of the considered plug-in (*i.e.* as set by `$(PLUGIN_NAME)`).

Kind	Name ³
Usual information	<i>plugin_DIR</i>
Source files	<i>plugin_CMO</i> <i>plugin_CMI</i> <i>plugin_CMX</i> <i>plugin_TYPES_CMO</i> <i>plugin_TYPES_CMX</i>
Compilation flags	<i>plugin_BFLAGS</i> <i>plugin_OFLAGS</i> <i>plugin_LINK_BFLAGS</i> <i>plugin_LINK_OFLAGS</i> <i>plugin_LINK_GUI_BFLAGS</i> <i>plugin_LINK_GUI_OFLAGS</i>
Dependencies	<i>plugin_DEPFLAGS</i> <i>plugin_GENERATED</i>
Documentation	<i>plugin_DOCFLAGS</i> <i>plugin_TYPES_TODOC</i>
Testing	<i>plugin_TESTS_DIRS</i> <i>plugin_TESTS_LIB</i>

Figure 5.10: Variables defined by Makefile.plugin.



Appendix A

Changes

This chapter summarizes the major changes in this documentation between each Frama-C release. First we list changes of the last release.

- **Tutorial**: fully rewritten.
- **Architecture** and **Reference Manual**: remove references to `Cilutil` module.

Oxygen-20121001

- **Makefile** `WARN_ERROR_ALL` variable.
- **Log**: Debug category (`~dkey` argument).
- **Visitor**: `DoChildrenPost` action.
- **Testing**: document the need for directories to store result and oracles.
- **Project Management System**: Fine tuning of AST dependencies.
- **Testing**: added `PTESTS_OPTS` and `PLUGIN_PTESTS_OPTS` Makefile's variables.
- **Type**: document the `type` library.
- **Logical Annotations**: fully updated.
- **Reference Manual**: update kernel files.
- **Testing**: merge parts in *Advanced Plug-in Development* and in *Reference Manual*.
- **Website**: refer to CEA internal documentation.
- **Command Line Options**: explain how to modify the default behavior of an option.
- **Command Line Options**: fully updated.
- **Project Management System**: fully updated.
- **Plug-in Registration and Access**: `Type` replaced by `Datatype` and document labeled argument `journalize`.

- **Configure.in**: updated.
- **Plug-in General Services**: updated.
- **Software Architecture**: `Type` is now a library, not just a single module.

Nitrogen-20111001

- **Tutorial of the Future**: new chapter for preparing a future tutorial.
- **Types as first class values**: links to articles.
- **Tutorial**: kernel-integrated plug-ins are now deprecated.
- **Visitors**: example is now out-of-date.

Carbon-20110201

Unchanged.

Carbon-20101201-beta1

- **Visitors**: update example to new kernel API.
- **Documentation**: external plugin API documentation.
- **Visitors**: fix bug (replace `DoChildrenPost` by `ChangeDoChildrenPost`), change semantics wrt `vstmt_aux`.

Carbon-20101201-beta1

- **Very Important Preliminary Warning**: adding this very important chapter.
- **Tutorial**: fix bug in the ‘Hello World’ example.
- **Testing**: updated semantics of `CMD` and `STDOPT` directives.
- **Initialization Steps**: updated according to new options `-then` and `-then-on` and to the new ‘Files Setting’ stage.
- **Visitors**: example updated

We list changes of previous releases below.

Boron-20100401

- **Configure.in**: updated
- **Tutorial**: the section about kernel-integrated plug-in is out-of-date
- **Project**: no more `rehash` in datatypes
- **Initialisation Steps**: fixed according to the current implementation
- **Plug-in Registration and Access**: updated according to API changes
- **Documentation**: updated and improved
- **Introduction**: is aware of the Frama-C user manual
- **Logical Annotations**: fully new section
- **Tutorial**: fix an efficiency issue with the Makefile of the `Hello` plug-in

Beryllium-20090902

- **Makefiles**: update according to the new `Makefile.kernel`

Beryllium-20090901

- **Makefiles**: update according to the new makefiles hierarchy
- **Writing messages**: fully documented
- **Initialization Steps**: the different stages are more precisely defined. The implementation has been modified to take into account specificities of dynamically linked plug-ins
- **Project Management System**: mention value `descr` in `Datatype`
- **Makefile.plugin**: add documentation for additional parameters

Beryllium-20090601-beta1

- **Initialization Steps**: update according to the new implementation
- **Command Line Options**: update according to the new implementation
- **Plug-in General Services**: fully new section introducing the new module `Plugin`
- **File Tree**: update according to changes in the kernel
- **Makefiles**: update according to the new file `Makefile.dynamic` and the new file `Makefile.config.in`
- **Architecture**: update according to the recent implementation changes
- **Tutorial**: update according to API changes and the new way of writing plug-ins

- **configure.in**: update according to changes in the way of adding a simple plug-in
- **Plug-in Registration and Access**: update according to the new API of module `Type`

Lithium-20081201

- **Changes**: fully new appendix
- **Command Line Options**: new sub-section *Storing New Dynamic Option Values*
- **Configure.in**: compliant with new implementations of `configure_library` and `configure_tool`
- **Exporting Datatypes**: now embeded in new section *Plug-in Registration and Access*
- **GUI**: update, in particular the full example has been removed
- **Introduction**: improved
- **Plug-in Registration and Access**: fully new section
- **Project**: compliant with the new interface
- **Reference Manual**: integration of dynamic plug-ins
- **Software architecture**: integration of dynamic plug-ins
- **Tutorial**: improve part about dynamic plug-ins
- **Tutorial**: use `Db.Main.extend` to register an entry point of a plug-in.
- **Website**: better highlighting of the directory containing the html pages

Lithium-20081002+beta1

- **GUI**: fully updated
- **Testing**: new sub-section *Alternative testing*
- **Testing**: new directive `STDOPT`
- **Tutorial**: new section *Dynamic plug-ins*
- **Visitor**: `ChangeToPost` in sub-section *Action Performed*

Helium-20080701

- **GUI**: fully updated
- **Makefile**: additional variables of `Makefile.plugin`
- **Project**: new important note about registration of internal states in Sub-section *Internal State: Principle*
- **Testing**: more precise documentation in the reference manual

Hydrogen-20080502

- **Documentation:** new sub-section *Website*
- **Documentation:** new ocaml doc tag *@plugin developer guide*
- **Index:** fully new
- **Project:** new sub-section *Internal State: Principle*
- **Reference manual:** largely extended
- **Software architecture:** fully new chapter

Hydrogen-20080501

- **First public release**



Bibliography

- [1] Patrick Baudin, Jean-Christophe Filliâtre, Thierry Hubert, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI C Specification Language. Version 1.6*, September 2012.
- [2] Patrick Baudin and Anne Pacalet. Slicing plug-in. <http://frama-c.com/slicing.html>.
- [3] Sylvain Conchon, Jean-Christophe Filliâtre, and Julien Signoles. Designing a generic graph library using ML functors. In Marco T. Morazán, editor, *Trends in Functional Programming*, volume 8 of *Trends in Functional Programming*, pages 124–140. Intellect, UK/The University of Chicago Press, USA, 2008. <http://ocamlgraph.lri.fr>.
- [4] Loïc Correnson, Pascal Cuoq, Florent Kirchner, Armand Puccetti, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. *Frama-C User Manual*, September 2012. <http://frama-c.cea.fr/download/user-manual.pdf>.
- [5] Loïc Correnson, Zaynah Dargaye, and Anne Pacalet. *Frama-C's WP plug-in*, April 2013. <http://frama-c.com/download/frama-c-wp-manual.pdf>.
- [6] Loïc Correnson and Julien Signoles. Combining Analysis for C Program Verification. In *Formal Methods for Industrial Critical Systems (FMICS)*, 2012.
- [7] Pascal Cuoq, Damien Doligez, and Julien Signoles. Lightweight Typed Customizable Unmarshaling. *ML Workshop'11*, September 2011.
- [8] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C, A Program Analysis Perspective. In *the 10th International Conference on Software Engineering and Formal Methods (SEFM 2012)*, volume 7504 of *LNCS*, pages 233–247. Springer, 2012.
- [9] Pascal Cuoq and Julien Signoles. Experience Report: OCaml for an industrial-strength static analysis framework. In *Proceedings of International Conference of Functional Programming (ICFP'09)*, pages 281–286, New York, NY, USA, September 2009. ACM Press.
- [10] Pascal Cuoq, Boris Yakobowski, and Virgile Prevosto. *Frama-C's value analysis plug-in*, April 2013. <http://frama-c.cea.fr/download/value-analysis.pdf>.
- [11] Free Software Foundation. *GNU 'make'*, April 2006. <http://www.gnu.org/software/make/manual/make.html#Flavors>.
- [12] Jacques Garrigue, Benjamin Monate, Olivier Andrieu, and Jun Furuse. LablGTK2. <http://lablgtk.forge.ocamlcore.org>.
- [13] Philippe Hermann and Julien Signoles. *Frama-C's RTE plug-in*, April 2013. <http://frama-c.com/download/frama-c-rte-manual.pdf>.

BIBLIOGRAPHY

- [14] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system*. <http://caml.inria.fr/pub/docs/manual-ocaml/index.html>.
- [15] Donald Michie. Memo functions: a language feature with "rote-learning" properties. Research Memorandum MIP-R-29, Department of Machine Intelligence & Perception, Edinburgh, 1967.
- [16] Donald Michie. Memo functions and machine learning. *Nature*, 218:19–22, 1968.
- [17] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag.
- [18] Julien Signoles. Foncteurs impératifs et composés: la notion de projet dans Framac. In Hermann, editor, *JFLA 09, Actes des vingtièmes Journées Francophones des Langages Applicatifs*, volume 7.2 of *Studia Informatica Universalis*, pages 245–280, 2009. In French.
- [19] Julien Signoles. Une bibliothèque de typage dynamique en OCaml. In Hermann, editor, *JFLA 11, Actes des vingt-deuxièmes Journées Francophones des Langages Applicatifs*, Studia Informatica Universalis, pages 209–242, January 2011. In French.
- [20] Nicolas Stouls and Virgile Prevosto. *Framac's Aorai plug-in*, April 2013. <http://frama-c.com/download/frama-c-aorai-manual.pdf>.

List of Figures

2.1	Plug-in Integration Overview.	14
2.2	Control flow graph for file <code>test.c</code>	21
2.3	Control flow graph colored with reachability information.	23
2.4	CFG plug-in architecture	26
3.1	Architecture Design.	34
3.2	Differences between standard plug-ins and kernel-integrated ones.	36
4.1	<code>ptests</code> options.	50
4.2	Directives in configuration headers of test files.	51
4.3	Representation of the Frama-C State.	67
4.4	Interaction between the project library and your registered global data.	68
4.5	Indices of AST nodes.	82
5.1	Frama-C directories.	89
5.2	Cil directories.	90
5.3	Kernel directories.	91
5.4	Main kernel modules.	92
5.5	Sections of <code>configure.in</code>	93
5.6	Relationship between the Makefiles	95
5.7	Sections of <code>Makefile.config.in</code> , <code>Makefile.common</code> and <code>Makefile</code>	96
5.8	Standard parameters of <code>Makefile.dynamic</code> and <code>Makefile.plugin</code>	99
5.9	Special parameters of <code>Makefile.dynamic</code> and <code>Makefile.plugin</code>	101
5.10	Variables defined by <code>Makefile.plugin</code>	103



Index

- Abstract Interpretation, [84](#)
 - Lattice, *see* Lattice
 - Toolbox, [35](#), [85](#), [91](#)
- Abstract Syntax Tree, *see* AST
- Abstract_interp, [35](#), [85](#), [91](#)
 - Lattice, [85](#)
- ACSL, [33](#), [35](#), [40](#), [90](#)
 - Frontend, [91](#)
- ai, [91](#)
- Alarms, [92](#)
- Annotation, [35](#), [80](#), [83](#), [92](#)
- Annotations, [83](#), [92](#)
 - add_assert, [82](#)
- ANSI C Specification language, *see* ACSL
- Architecture, [33](#)
 - Plug-in, [13](#)
- AST, [33](#), [35](#), [67](#), [79](#), [80](#), [83](#), [90](#), [92](#)
 - Copying, [81](#), [82](#)
 - Initializer, [92](#)
 - Modification, [35](#), [39](#), [72](#), [73](#), [80](#), [81](#)
 - Sharing, *see* Sharing
- Ast, [92](#)
 - add_monotonic_state, [71](#)
 - get, [25](#), [67](#)
 - mark_as_changed, [71](#)
 - mark_as_grown, [71](#)
 - self, [29](#), [73](#), [82](#)
- Ast_info, [92](#)
- bin, [90](#)
- Binary, [90](#)
- Boot, [77](#), [92](#)
- C Intermediate Language, *see* Cil
- Call graph computation, [35](#)
- Callgraph, [35](#), [90](#)
- CEA_INRIA_LGPL, [87](#)
- CFG, [92](#)
- Cfg, [90](#)
- check_plugin, [41](#)
- CIL, [90](#)
 - Syntactic Analysis, [90](#)
 - Visitor, [92](#)
- Cil, [33](#), [34](#), [35](#), [39](#), [79](#)
 - API, [34](#), [35](#)
 - AST, *see* AST
 - Visitor, [79](#)
 - Entry Point, [79](#)
- Cil, [35](#), [90](#)
 - cilVisitor, [79](#), [79](#)
 - behavior, [82](#)
 - current_kinstr, [82](#)
 - fill_global_tables, [81](#)
 - get_filling_actions, [81](#), [82](#)
 - vexpr, [82](#)
 - vfile, [20](#)
 - vglob, [80](#)
 - vlogic_ctor_info_decl, [80](#)
 - vlogic_ctor_info_use, [80](#)
 - vlogic_info_decl, [80](#)
 - vlogic_info_use, [80](#)
 - vlogic_type_info_decl, [80](#)
 - vlogic_type_info_use, [80](#)
 - vlogic_var_decl, [80](#)
 - vlogic_var_use, [80](#)
 - voffs, [80](#)
 - vstmt, [80](#)
 - vvdec, [80](#)
 - vvrbl, [80](#)
 - copy_visit, [81](#), [82](#)
 - DoChildren, [21](#), [22](#)
 - DoChildrenPost, [20](#)
 - dummyStmt, [65](#)
 - get_name, [81](#)
 - get_kernel_function, [82](#)
 - get_original_name, [81](#)
 - get_stmt, [82](#)
 - inplace_visit, [81](#)
 - JustCopy, [20](#)
 - lzero, [82](#)
 - reset_behavior_name, [81](#)
 - set_name, [81](#)

- SkipChildren, 20
- visitAction, 80
 - ChangeDoChildrenPost, 81, 82
 - ChangeTo, 80, 82
 - ChangeToPost, 80
 - DoChildren, 80, 82
 - DoChildrenPost, 80
 - JustCopy, 80, 82
 - JustCopyPost, 80
 - SkipChildren, 80, 82
- visitCilAstType, 79
- visitCilFile, 80
- visitCilFileCopy, 80
- visitCilFileSameGlobals, 80
- visitor_behavior, 81
- cil, 39, 90, 90
 - src, 35, 90
 - ext, 90
 - frontc, 90
 - logic, 91
- Cil_datatype, 59
 - Fundec
 - Hashtbl, 29
 - Stmt
 - equal, 59
 - Hashtbl, 69
 - pretty, 59, 65
 - t, 65
 - ty, 65, 66
 - Varinfo, 69
- Cil_state_builder, 68, 69
 - Stmt_hashtbl, 69
- Cil_types, 35, 90
 - binop
 - Div, 82
 - Mod, 82
 - Block, 20
 - Break, 20
 - compinfo, 81, 82
 - Continue, 20
 - enuminfo, 81, 82
 - exp_node
 - BinOp, 82
 - fieldinfo, 81, 82
 - file, 79–82
 - fundec, 23
 - GFun, 20
 - global, 80
 - Goto, 20
 - If, 20
 - Instr, 20
 - logic_ctor_info, 80
 - logic_info, 80–82
 - logic_type_info, 80
 - logic_var, 80–82
 - Loop, 20
 - offset, 80
 - relation
 - Rneq, 82
 - Return, 20
 - stmt, 81, 82
 - Switch, 20
 - TryExcept, 20
 - TryFinally, 20
 - typeinfo, 81, 82
 - UnspecifiedSequence, 20
 - varinfo, 69, 80–82
- CilE, 92
- Cmdline, 36, 77, 92
 - Exit, 78
 - is_going_to_load, 79
 - nop, 78
 - run_after_configuring_stage, 79
 - run_after_early_stage, 78
 - run_after_exiting_stage, 78
 - run_after_extended_stage, 70, 78
 - run_after_loading_stage, 78
 - run_after_setting_files, 79
 - run_during_extending_stage, 78
 - stage
 - Configuring, 79
 - Early, 78
 - Exiting, 78
 - Extended, 78
 - Extending, 78
 - Loading, 78
- Command, 92
- Command Line, 17, 24
 - ocode, 58
 - Option, 53, 74, 74
 - Parsing, 77
- Config, 92
- configure.in, 40, 43, 93
 - check_plugin, 41
 - check_plugin_dependencies, 43
 - configure_library, 41
 - configure_tools, 41
 - DYNAMIC_plugin, 41

- ENABLE_plugin, [41](#)
- FORCE_plugin, [41](#)
- HAS_library, [42](#)
- LIB_SUFFIX, [42](#)
- OBJ_SUFFIX, [42](#)
- plugin_require, [43](#)
- plugin_require_external, [42](#)
- plugin_use, [43](#)
- plugin_use_external, [42](#)
- REQUIRE_plugin, [41](#)
- SELECTED_library, [42](#)
- USE_plugin, [41](#)
- Consistency, [39](#), [67](#), [74](#), [80](#), [81](#)
- Context Switch, [71](#), [73](#)
- Control Flow Graph, *see* CFG
- Copyright, [87](#)
- CP, [97](#)
- Dataflow, [35](#), [84](#), [90](#)
- Dataflow analysis, [35](#), [90](#)
- Datatype, [58](#), [59](#), [69](#), [71](#)
- Datatype, [59](#)
 - Bool, [30](#)
 - bool, [59](#)
 - char, [59](#)
 - func, [64](#), [66](#)
 - func2, [59](#)
 - func3, [66](#)
 - Function, [62](#)
 - identity, [60](#)
 - Int, [61](#)
 - int, [59](#)
 - List, [61](#), [62](#)
 - list, [59](#)
 - Make, [60](#), [61](#)
 - never_any_project, [60](#)
 - Pair, [69](#)
 - Polymorphic, [61](#)
 - Polymorphic2, [61](#)
 - pp_fail, [60](#)
 - Ref, [72](#)
 - S, [59](#), [72](#)
 - S_with_collections, [59](#)
 - Serializable_undefined, [65](#)
 - String, [29](#), [59](#)
 - Hashtbl, [61](#)
 - Set, [59](#)
 - string, [59](#)
 - Undefined, [61](#)
 - undefined, [61](#)
 - unit, [64](#), [66](#)
- Db, [34](#), [36](#), [62](#), [63](#), [64](#), [86](#), [91](#), [92](#)
 - From.self, [70](#)
 - Impact.compute_pragmas, [62](#)
 - Main, [14](#)
 - extend, [13–17](#), [53](#), [77](#), [79](#), [82](#)
 - progress, [85](#)
 - Value, [29](#)
 - compute, [69](#), [73](#)
 - get_stmt_state, [22](#)
 - is_computed, [22](#), [30](#), [68](#), [73](#)
 - is_reachable, [22](#)
 - self, [69](#), [70](#), [74](#)
- Db.Properties, [84](#)
- Design, [14](#)
 - main_window_extension_points, [85](#)
 - register_source_selector, [24](#)
 - register_extension, [24](#), [85](#)
- doc, [90](#)
- Documentation, [85](#), [90](#), [98](#)
 - Kernel, [86](#)
 - Plug-in, *see* Plug-in Documentation
 - Source, [86](#)
 - Tags, [86](#)
- Dot, [42](#)
- Dynamic, [14](#), [36](#), [64](#), [86](#), [92](#)
 - get, [64](#), [66](#)
 - Parameter, [76](#)
 - Bool, [76](#)
 - register, [64](#), [64](#)
- Emitter, [75](#)
- Emitter, [92](#)
 - create, [83](#), [84](#)
- Entry Point, [67](#)
- Entry point, [13](#)
- Equality
 - Physical, [71](#)
 - Structural, [71](#)
- external, [90](#)
- Extlib, [35](#), [91](#)
 - the, [82](#)
- File, [92](#)
 - create_project_from_visitor, [82](#)
 - init_from_c_files, [79](#)
 - init_from_cmdline, [79](#)
 - init_project_from_cil_file, [73](#), [79](#)
 - init_project_from_visitor, [73](#), [79](#)
- Floating_point, [92](#)

- FRAMAC_LIBDIR, 16, 17, 25, 45
- FRAMAC_SHARE, 16, 17, 25, 45
- From, 70
- From_parameters
 - ForceCallDeps, 76
- Function, 35
- Globals, 35, 92
 - Functions
 - get, 24
 - set_entry_point, 68
- GnomeCanvas, 42
- Gtk_helper
 - graph_window, 24
- GUI, 14, 85, 93
- gui, 93
- Gui_init, 92
- Hashtable, 68, 69
- Header, 87, 98
- headers, 87, 90
- Hello, 39
- Highlighting, 85
- Hook, 13
- index.html, 86
- Initialization, 64, 77, 77
- install-doc-code, 87
- Ival, 35, 91
- Journal, 14, 36, 92
- Journalisation, 60
- Journalization, 28, 64, 77, 78
- Kernel, 33, 34, 35, 40, 71, 91, 94, 98
 - Toolbox, 91
- Kernel, 74, 92
 - CodeOutput, 58
 - SafeArrays, 77
 - Unicode, 76
- kernel, 91
- Kernel_function, 69, 92
 - dummy, 65
 - get_definition, 24
 - Make_Table, 70
 - pretty, 65
 - t, 65, 69
 - ty, 65, 66
- Kind, 69
- Lablgtk, 42, 85
- Lablgtksourceview2, 42
- Lattice, 34, 35, 85, 91
- Lexing, 34, 35
- LGPL, 87
- lib, 90, 91
 - fc, 90
 - gui, 90
 - plugins, 90
- Library, 40, 90
 - Configuration, 41, 93
 - Dependency, 42
- licences, 87
- License, 87, 98
- LICENSES, 87
- licenses, 90
- Linking, 34–36, 77, 78
- Lmap, 35, 85, 91
- Lmap_bitwise, 35, 85, 91
- Loading, 67, 72, 78
- Location, 84, 91
- Locations, 35, 84, 91
 - enumerate_valid_bits, 84
 - Location, 84
 - location, 84
 - Location_Bits, 84
 - Location_Bytes, 84
 - Zone, 84
- Log, 35, 92
 - add_listener, 56
 - log_channel, 57
 - Messages, 53, 54
 - abort, 55
 - debug, 54
 - error, 55
 - failure, 55
 - fatal, 55
 - feedback, 54
 - log, 57
 - result, 54
 - verify, 55
 - warning, 54
 - with_log, 57
 - new_channel, 57
 - print_delayed, 58
 - print_on_output, 58
 - set_echo, 56
 - set_output, 58
 - with_log_channel, 57
- Logging, *see* Messages

- logic, **91**
- Logic Type System, **91**
- Logic_const, **84, 91**
 - prel, **82**
- Logic_typing, **91**
- Logic_utils, **84**
 - expr_to_term, **82**
- Loop, **92**
- Makefile, **44, 85–87, 90, 94, 94, 95**
- Makefile.common, **94**
- Makefile.config.in, **44, 94, 95**
- Makefile.dynamic, **14, 16, 17, 25, 45, 45, 94, 95, 99**
- Makefile.dynamic_config, **94**
- Makefile.dynamic_config.external, **94**
- Makefile.dynamic_config.internal, **94**
- Makefile.kernel, **94**
- Makefile.plugin, **44, 45, 94, 95, 99**
- Marshaling, **60**
- memo, **69**
- Memoization, **66, 69, 70**
- Memory State, **34, 35**
- Memory States
 - Toolbox, **91**
- memory_states, **91**
- Messages, **53**
- misc, **91**
- Module Initialization, *see* Initialization
- ocamlgraph, **90**
- Occurrence, **41, 85**
- Oracle, **46, 48, 50**
- Parameter, **66**
- Parameter, **92**
- Parameter.t, **75**
- Parameters, **74**
- Parsing, **34, 35**
- Pdg, **70**
- Platform, **93**
- Plug-in, **13, 33, 36**
 - Access, **63**
 - API, **28, 63**
 - Architecture, **13**
 - Command Line Options, **17, 24**
 - Compilation, **98**
 - Compiled, **90**
 - Configure, **28**
 - Dependency, **40, 40, 43, 93, 100**
 - Directory, **85, 99**
 - Distribution, **102**
 - Documentation, **19, 86, 86, 100**
 - GUI, **14, 23, 42, 77, 85, 100**
 - Implementation, **91**
 - Initialization, *see* Initialization
 - Interface, **86, 100**
 - Kernel-integrated, **36**
 - Access, **62**
 - Registration, **62**
 - License, **87**
 - Makefile, **16, 25**
 - Messages, **15**
 - Name, **99**
 - Occurrence, *see* Occurrence
 - Pdg, *see* Pdg
 - Registration, **15, 63**
 - Script, **14**
 - Slicing, *see* Slicing
 - Sparecode, *see* Sparecode
 - Status, **40**
 - Test, **100, 102**
 - Testing, **19**
 - Types, **34, 37, 63, 91, 100**
 - Wished, **93**
- plugin_types, **63**
- Plugin
 - Kernel-integrated, **93, 94**
- Plugin, **13, 14, 36, 52, 76, 92**
 - Bool, **75**
 - General_services, **75, 75**
 - False, **75, 75**
 - Int, **75**
 - String, **75**
 - StringSet, **75, 75**
 - True, **75**
 - Zero, **75**
 - get_selection, **74**
 - Int, **75, 75**
 - Parameter, **75**
 - Register, **15–17, 24, 52, 53, 65, 75**
 - set_negative_option_name, **77**
 - String_set, **75**
- PLUGIN_BFLAGS, **100**
- plugin_BFLAGS, **103**
- PLUGIN_CMI, **100**
- plugin_CMI, **103**
- PLUGIN_CMO, **16, 17, 25, 45, 100**
- plugin_CMD, **103**
- PLUGIN_DEPENDS, **100**

- PLUGIN_DEPFLAGS, **100**
- plugin_DEPFLAGS*, **103**
- PLUGIN_DIR, **45, 99**
- plugin_DIR*, **103**
- PLUGIN_DISTRIB_BIN, **102**
- PLUGIN_DISTRIB_EXTERNAL, **102**
- PLUGIN_DISTRIBUTED, **45, 102**
- PLUGIN_DOCFLAGS, **100**
- plugin_DOCFLAGS*, **103**
- PLUGIN_DYNAMIC, **100**
- PLUGIN_ENABLE, **45, 99**
- PLUGIN_EXTRA_BYTE, **100**
- PLUGIN_EXTRA_OPT, **100**
- PLUGIN_GENERATED, **100**
- plugin_GENERATED*, **103**
- PLUGIN_GUI_CMO, **25, 85, 100**
- plugin_GUI_OFLAGS*, **103**
- PLUGIN_HAS_EXT_DOC, **100**
- PLUGIN_HAS_MLI, **45, 100**
- PLUGIN_INTERNAL_TEST, **45, 102**
- PLUGIN_INTRO, **86, 100**
- plugin_LINK_BFLAGS*, **103**
- PLUGIN_LINK_GUI_BFLAGS, **100**
- plugin_LINK_GUI_BFLAGS*, **103**
- PLUGIN_LINK_GUI_OFLAGS, **100**
- PLUGIN_LINK_OFLAGS, **100**
- plugin_LINK_OFLAGS*, **103**
- PLUGIN_NAME, **16, 17, 25, 45, 86, 99, 102**
- PLUGIN_NO_DEFAULT_TEST, **102**
- PLUGIN_NO_TEST, **100**
- PLUGIN_OFLAGS, **100**
- plugin_OFLAGS*, **103**
- PLUGIN_PTESTS_OPTS, **102**
- PLUGIN_TESTS_DIRS, **100**
- plugin_TESTS_DIRS*, **103**
- PLUGIN_TESTS_LIB, **102**
- plugin_TESTS_LIB*, **103**
- PLUGIN_TYPES_CMO, **45, 63, 97, 100**
- plugin_TYPES_CMO*, **103**
- plugin_TYPES_CMX*, **103**
- PLUGIN_Types_TODO, **100**
- PLUGIN_UNDOC, **100**
- Preprocessing, **35**
- Pretty_source
 - PVDecl, **24**
- PRINT_CP, **97**
- Printer_api
 - S.pp_exp, **20**
 - S.pp_instr, **20**
 - S.pp_stmt, **19**
 - S.pp_varinfo, **20**
- Project, **28, 39, 60, 66, 79, 81, 91**
 - Current, **67, 67, 71, 72, 74, 81**
 - Initial, **79**
 - Use, **72**
- Project, **14, 34, 36, 73**
 - clear, **30, 30, 73, 74**
 - current, **67, 73**
 - IOError, **73**
 - load, **73**
 - on, **73, 74**
 - save, **73**
 - set_current, **73, 73**
- project, **91**
- Project_skeleton
 - t, **73**
- Property_status, **84**
- Ptests, **46, 98**
- ptest_local_config.cmo, **47**
- PTESTS_OPTS, **98**
- Rangemap, **35**
- Rte, **45**
- Saving, **39, 67, 69, 73**
- Selection, **67, 74**
- self, **69**
- Session, **73**
- share, **90**
- Sharing, **81, 82**
 - Widget, **85**
- Side-Effect, **72, 77**
- Slicing, **87**
- Sparecode, **47**
- Special_hooks, **92**
- src, **39, 90, 91**
 - ai, **35**
 - kernel, **35**
 - lib, **35**
 - memory_state, **35**
 - misc, **35**
 - project, **36**
 - type, **36**
- State, **66, 67, 74, 75, 79, 80**
 - Cleaning, **72, 74**
 - Dependency, **67, 69, 71, 74**
 - Postponed, **70, 78**
 - Functionalities, **67**
 - Global Version, **71**

- Kind, *see* Kind
- Local Version, **71, 72**
- Name, **69, 71**
- Registration, **67–69**
- Selection, *see* Selection
- Sharing, **72**
- State, **70**
 - dummy, **70**
- State_builder, **68, 69**
 - Hashtbl, **29**
 - Ref, **30, 72**
 - Register, **68, 69, 71, 72**
- State_dependency_graph
 - S
 - add_codedependencies, **70**
- State_selection, **74**
 - S
 - only_dependencies, **73**
 - with_dependencies, **74**
 - S.with_dependencies, **30**
 - with_dependencies, **31**
- Stmts_graph, **92**
- Structural_descr
 - p_int, **60, 61**
 - pack, **61**
 - structure
 - Sum, **60, 61**
 - t
 - Structure, **60, 61**
- Tags, **98**
- Task, **92**
- Test, **19, 46, 98**
 - Configuration, **48**
 - Directive, **48**
 - Header, **48, 49**
 - Suite, **47, 48, 90**
- Test
 - Directive
 - CMD, **51**
 - COMMENT, **51**
 - DONTRUN, **51**
 - EXECNOW, **51, 52**
 - FILEREG, **51, 52**
 - FILTER, **51**
 - GCC, **51**
 - OPT, **49, 51**
 - STDOPT, **51, 52**
- test_config, **48, 51, 52**
- tests, **47, 50, 90**
- Tool, **40**
 - Configuration, **41, 93**
 - Dependency, **42**
- Type, **91**
 - Dynamic, **58**
 - Library, **58**
 - Value, **59, 64, 65**
- Type, **14, 36, 91**
 - Abstract, **65, 66**
 - AlreadyExists, **64**
 - name, **61**
 - par, **60, 61**
 - precedence
 - Basic, **60**
 - Call, **60**
 - t, **59, 64**
- type, **91**
- Type value, **91**
- Typing, **34, 35**
- UNPACKED_DIRS, **45, 63, 97**
- Variable
 - Global, **35**
- VERBOSEMAKE, **44, 97**
- Visitor, **19, 79**
 - Behavior, **81, 81**
 - Cil, *see* Cil Visitor
 - Copy, **72, 81, 81, 82**
 - In-Place, **81, 81**
- Visitor, **35, 92**
 - frama_c_inplace, **20**
 - frama_c_visitor
 - current_kf, **82**
 - vglob_aux, **20, 80**
 - vstmt_aux, **21, 22, 80**
 - generic_frama_c_visitor, **79, 82**
 - visitFramacFileSameGlobals, **25**
 - visitFramacFunction, **29**
- WARN_ERROR_ALL, **44**
- Website, **87**