frama C

Software Analyzers

Developer Manual

Plug-in Development Guide

Release Boron-20100401

Julien Signoles with Loïc Correnson and Virgile Prevosto

CEA LIST, Software Reliability Laboratory

This work has been supported by the 'CAT' ANR project (ANR-05-RNTL-00301).

Contents

Foreword

This is the documentation of the Frama-C implementation[1] which aims to help any developer to integrate a new plug-in inside this platform. It was first a deliverable of the task 2.3 of the ANR RNTL project CAT[2].

The content of this document corresponds to the version Boron-20100401 (April 12, 2010) 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, Pierre-Loïc Garoche, Nikolaï Kosmatov, Benjamin Monate, Yannick Moy, Anne Pacalet, Armand Puccetti and Boris Yakobowski. We also thank Johannes Kanig for his Mlpost support[3], the tool used for making the figures of this document.

[1] http://frama-c.cea.fr
[2] http://www.rntl.org/projet/resume2005/cat.htm
[3] http://mlpost.lri.fr

Chapter 1

Introduction

This guide aims at helping any developer to 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 4.18.1 for additional details about this documentation).

This guide does not introduce neither the use of Frama-C which is the purpose of the user manual [4], nor the use of plug-ins which should be introduced in separated and dedicated manuals. We assume that the reader of this guide already reads 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 finishes to read the user manual and wishes to develop his/her own analysis with the help of Frama-C, or an intermediate-level plug-in developer who wants to better understand one particular aspect of the framework, or a Frama-C expert who aims to remember details about one specific point of the Frama-C development.

Frama-C is fully developed within the Objective Caml programming language (*aka* OCaml) [11]. Motivations for this choice are given in a Frama-C experience report [6]. However this guide *does not* provide any introduction to this programming language: the World Wide Web already contains plenty resources for OCaml developers (for instance, see `http://caml.inria.fr/resources/doc/index.en.html` for getting many pointers).

1.1 About this document

In order to ease the reading, beginning of sections may state the category of readers it is 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 like ***1*** reference the defining sections for the corresponding entries while other numbers (like 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 ocamldoc 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.

Most important paragraphs are displayed inside a gray box like this one. A plug-in developer **must** follow them very carefully.

There are numbers pieces of code written in this document. Do not copy/paste them from the PDF to your favorite text editor because the PDF text may contain non-ASCII characters preventing the code from compile.

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 design of 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 to explain step-by-step how to proceed towards this goal. It will get you started but does not tell the whole story. In particular, some very important aspects for the integration in the framework are omitted here and are described in chapter 4.

Section 2.1 explains the basis for writing a standard Frama-C plug-in while section 2.2 explains the basis for writing a plug-in integrated with the Frama-C kernel: this is slightly more involved but allows a deeper integration within the Frama-C architecture. You should do this only if you intend to contribute a large and very general purpose plug-in to the community.

2.1 Standard Plug-in

This section will teach you how to write the most basic plug-in and run it from the Frama-C toplevel.

Prerequisite: *To follow this tutorial:*

- Frama-C *needs to be installed in your path;*
- *the* Objective Caml *compilers must be installed in your path. These must be the same compilers as the ones you used to compile* Frama-C[1]*;*
- *GNU* make *must be in your path.*

2.1.1 Plug-in Integration Overview

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

[1] If you have an Objective Caml version <3.11 then only bytecode plug-ins are available. Upgrade to Objective Caml >=3.11 if you need native code plug-ins.

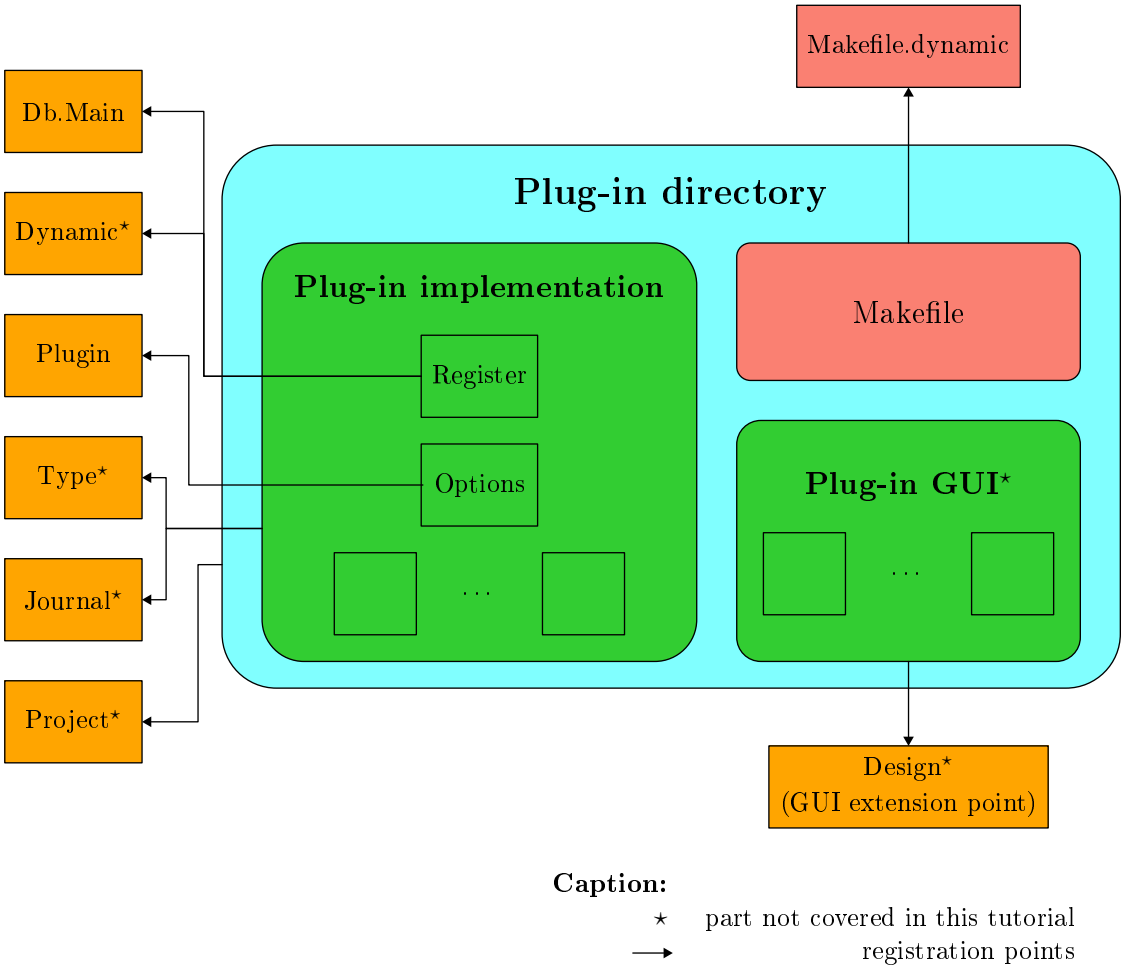

Figure 2.1: Plug-in Integration Overview.

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` as adding a new Frama-C option.

2.1.2 Hello Frama-C World

A very basic plug-in is the 'Hello World' plug-in. This plug-in adds a command line option `-hello` to Frama-C and pretty prints the message 'Hello World!' whenever the option is set. It is possible to program such an option just with the module `Arg` provided by the Objective Caml standard library and without the addition of a Frama-C plug-in, but we use this example to introduce the bases of plug-in development. This plug-in is our running example in this chapter.

The 'Hello World' plug-in consists of only two files: `Makefile` and `hello_world.ml`[2].

1. Create the two files `Makefile` and `hello_world.ml` containing the lines given in the frames at the end of this section.

   The name of each compilation unit (here `hello_world`) must be different of the plug-in name set by the `Makefile` (here `Hello`) in order to compile a plug-in.

2. Run `make` to compile it.

3. Run `make install` to install the plug-in. You need to have write access to the `$(FRAMAC_LIBDIR)/plugins` directory.

4. Test your plug-in with `frama-c.byte -hello`. The sentence 'Hello Frama-C World!' is printed.

File **Makefile**

```
# 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
```

File **hello_world.ml**

```
(** The traditional 'Hello world!' plugin.
    It contains one boolean state [Enabled] which can be set by the
    command line option "-hello".
    When this option is set it just pretty prints a message on the standard
    output. *)

(** Register the new plug-in "Hello World" and provide access to some plug-in
    dedicated features. *)
module Self =
  Plugin.Register
    (struct
       let name = "Hello world"
       let shortname = "hello"
       let descr = "The famous 'Hello world' plugin"
     end)

(** Register the new Frama-C option "-hello". *)
module Enabled =
  Self.False
    (struct
       let option_name = "-hello"
       let descr = "pretty print \"Hello world!\""
     end)

let print () = Self.result "Hello world!"

(** The code below is not mandatory: you can ignore it in a first reading. It
    provides an API for the plug-in, so that the function [run] is callable by
    another plug-in and journalized: first, each plug-in can call [Dynamic.get
```

[2]Both files are distributed within Frama-C and they are available from the directory `src/dummy/hello_world` of the source distribution.

```
    "Hello.run" (Type.func Type.unit Type.unit)] in order to call [run]
    and second, each call to [run] is written in the Frama-C journal. *)
let print =
  Dynamic.register
    ~plugin:"Hello"
    "run"
    ~journalize:true
    (Type.func Type.unit Type.unit)
    print

(** Print 'Hello World!' whenever the option is set. *)
let run () = if Enabled.get () then print ()

(** Register the function [run] as a main entry point. *)
let () = Db.Main.extend run
```

2.2 Kernel-integrated Plug-in

This section is out-of-date. Please report as "feature request" on http://bts.frama-c.com if you really need this section.

Target readers: *It is only for:*

- *beginners who have to implement a plug-in requiring a very deep integration within the* Frama-C *architecture;*
- *new* Frama-C*-kernel developers.*

Prerequisite: *Getting the* Frama-C *source.*

This section will teach you how to write the most basic kernel-integrated plug-in and run it from the Frama-C toplevel. This plug-in will be linked with the Frama-C kernel and with all the other kernel-integrated plug-ins. It is slightly more involved but allows a deeper integration within the Frama-C architecture. The running example of this section is the very same plug-in 'Hello World' than the one of the previous section.

2.2.1 Setup

Frama-C uses a makefile which is generated by the script `configure`. This script checks your system to determine the most appropriate Frama-C configuration, in particular the plug-ins that should be available. This file is itself generated by the autotool `autoconf`. Consequently, you have to execute the following commands:

```
$ autoconf
$ ./configure
```

This generates a proper makefile and lists the available plug-ins. Now you are able to compile sources with `make`.

```
$ make
```

This compilation produces the following binaries (in a standard configuration):

- `bin/toplevel.byte` and `bin/toplevel.opt` (Frama-C toplevel);

- `bin/viewer.byte` and `bin/viewer.opt` (Frama-C GUI);
- `bin/ptests.byte` (Frama-C testing tool).

Suffixes `.byte` and `.opt` respectively correspond to the bytecode and native versions of binaries. If you wish, and before having fun with Frama-C, you can:

- test the compiled platform with `make tests`;
- generate the source documentation with `make doc`;
- generate navigation tags for `emacs` with `make tags`.

2.2.2 Plug-in Integration Overview

Figure 2.2 shows how a kernel-integrated plug-in may integrate in the Frama-C platform. Some elements of this figure are pragmatically explained in the remaining sections of this tutorial.

The implementation of the plug-in is provided inside a specific directory and is connected to the Frama-C platform thanks to some registration points. These registrations are performed either through hooks (by applying a function or a functor) or directly by modifying some specific part of Frama-C files. That is the very major difference with integrating standard plug-ins: standard plug-ins never modify Frama-C files. For instance, you have to extend `Db` with your plug-in-specific operations and to register them inside it if you want people to be able to use your plug-in (see Section 2.2.5). However most of the registration way are the same as standard plug-ins (see Section 2.1) (for instance, extending the Frama-C entry point).

You also have to modify the files `Makefile` and `configure.in` in order to properly link your plug-in with Frama-C (see Section 2.2.4).

Moreover, the developer may provide a plug-in interface (which should usually be empty, see Section 2.2.5) and aventually specific test suites (see Section 2.2.6).

2.2.3 Hello Frama-C World

This section explains how to write the core of a kernel-integrated plug-in `Hello`. This is a plug-in which pretty-prints 'Hello World!' whenever the option `-hello` is set on the Frama-C command line.

First, we add a new subdirectory `hello` in directory `src`.

```
$ mkdir src/hello
```

This new directory is going to contain the source file of our new plug-in[3]. If you want, you can have a quick look at `src` which contains the kernel and existing plug-ins. We only use a few files of this directory in this tutorial.

We can now edit the source file of `hello`, called `src/hello/register.ml`. It should contain exactly the same code than the file `hello_world.ml` given page 15 in Section 2.1.2: there is not a so big difference between kernel-integrated plug-ins and the other ones.

Recommendation 2.1 *In Frama-C, the name of the "main" file of a plug-in p should always be called either `register.ml` or `p_register.ml`.*

At this point, we have a compilable plug-in made of a main function `run`.

[3] As the plug-in `hello` is tiny, it has only one source file.

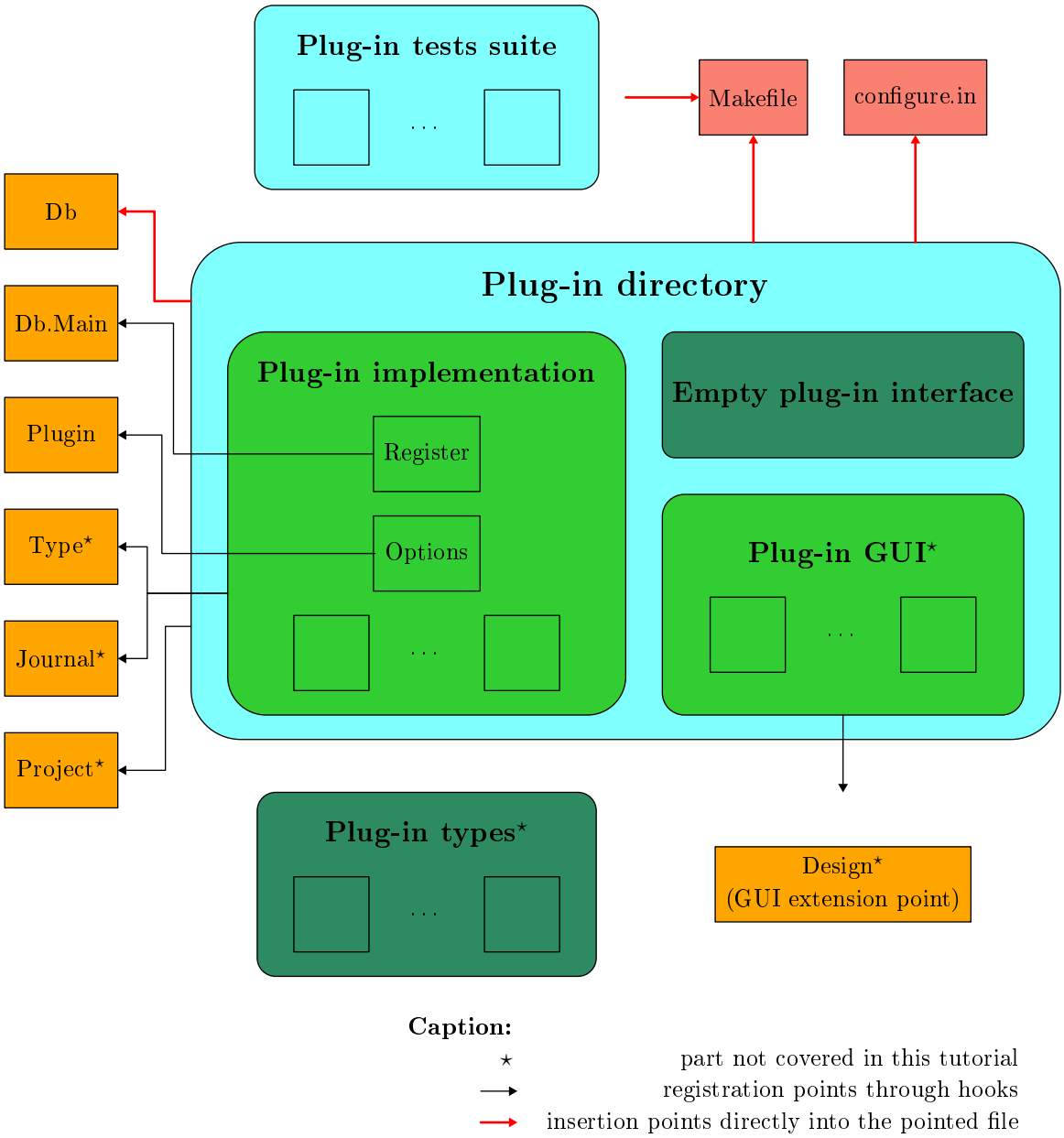

Figure 2.2: Kernel-integrated Plug-in Integration Overview.

2.2.4 Configuration and Compilation

Here we explain how to compile the plug-in `hello`. Section 4.2 and 4.3 provide more details about the configuration and compilation of plug-ins.

Configuration As explained in Section 2.2.1, Frama-C uses both autoconf and make in order to compile. Consequently, we have to modify both files `configure.in` and `Makefile` in order to compile our plug-in within Frama-C. In both files, some predefined scripts help with plug-in integration.

In order to compile the `hello` plug-in, first add the following lines into `configure.in`[4]. They indicate how to configure `hello`, especially whether it has to be compiled or not.

File **configure.in**

```
# Add the following lines after other plug-in configurations,
# in the section 'Plug-in sections'.

# hello
########
check_plugin(hello,src/hello,[support for hello plug-in],yes,no)
```

These lines correspond to the standard scheme for configuring a new plug-in. Function `check_plugin` is defined in `configure.in`. Its first argument is the plug-in name, the second one is the plug-in directory (the directory containing the plug-in source files), the third one is a help message, the fourth one indicates whether the plug-in is available by default (here `yes` says that the plug-in is available by default and a user may use option `-disable-hello` to deactivate the plug-in) and the last one indicates whether the plug-in will be dynamically linked within Frama-C (here `no` says that the plug-in will be statically linked).

Now we are ready to execute

```
$ autoconf
$ ./configure
```

and to check that the new plug-in `hello` is going to compile: you should have the line

```
checking for src/hello... yes
hello... yes
```

in the configuration summary.

Compilation Once `configure.in` is extended, we also have to modify `Makefile` with the following lines.

File **Makefile**

```
# Add the following lines after other plug-ins compilation directives,
# in the section 'Plug-in sections'.

###########
# Hello #
###########
PLUGIN_ENABLE:=@ENABLE_HELLO@
PLUGIN_NAME:=Hello
PLUGIN_DIR:=src/hello
PLUGIN_CMO:= register
PLUGIN_NO_TEST:=yes
include Makefile.plugin
```

[4] In this document, a comment containing ... among lines of code represents an undisplayed piece of code written either previously in the document or by someone else.

These lines use the predefined makefile `Makefile.plugin` which is a generic makefile dedicated to the compilation of one plug-in. There are more than twenty variables than can be used to customize the behavior of `Makefile.plugin`. These variables are all described in Section 5.3.3, but most of them have reasonable default values so that it is not necessary to describe more than the few ones above.

Now we briefly explain the variables that are set for `hello`.

- `PLUGIN_ENABLE` indicates that the plug-in should be compiled. Here we use the variable `@ENABLE_HELLO@` set by `configure.in`.
- `PLUGIN_NAME` is the name of the plug-in.

  The variable `PLUGIN_NAME` must hold a valid OCaml module name (in particular it must be capitalized).
- `PLUGIN_DIR` is the directory containing the source file(s) for the plug-in.
- `PLUGIN_CMO` is the list of the `.cmo` files (without the extension `.cmo` nor the plug-in path) required to compile the plug-in.
- `PLUGIN_NO_TEST` is set to `yes` because there is no specific test directory for the plug-in (see Section 2.2.6 about plug-in testing).

Now we are ready to compile Frama-C with the new plug-in `hello`.

```
$ make
```

At this point, the plug-in works properly: a Frama-C user can run the plug-in safely.

```
$ ./bin/toplevel.byte -hello
Hello World!
```

2.2.5 Connection with the Frama-C World

The plug-in `hello` is now compiled but it is not fully registered within the Frama-C framework. In particular, our plug-in should be added in the plug-in database `Db` in order to be simply used by other plug-ins (see Chapter 3 for details).

Extension of the Plug-in Database For this purpose, we have to extend `Db` with the new plug-in `hello`.

File **src/kernel/db.mli**

```
...
(** Hello World plug-in.
    @see <../hello/index.html> internal documentation. *)
module Hello : sig
  val run: (Format.formatter → unit) ref (** Print "hello world". *)
end
...
```

File **src/kernel/db.ml**

```
...
module Hello = struct let run = mk_fun "Hello_world.run" end
...
```

The interface declares a new module `Hello` containing a single function `run`. Indeed `run` is *a reference* to a function. This reference is not initialized in the implementation of `Db`: we use `mk_fun` (declared in the opened module `Extlib`) in order to declare the reference without instantiating it. This instantiation has to be done by the plug-in itself. Otherwise, a call to `!Db.run` raises the exception `Extlib.NotYetImplemented`. In order to fix this, we modify the module `Register` as follows.

File **src/hello/register.ml**

```
... definition of run
let () = Db.Hello.run ← run
```

It is important to note that the reference `Db.Hello.run` is set at the OCaml module initialization step. So the body of each Frama-C function can safely dereference it.

Documentation We have properly documented the interface of `Db` with `ocamldoc` through special comments between `(**` and `*)`. This documentation is generated by `make doc`. In particular, this command also generates an internal documentation for `hello` which is accessible in the directory `doc/code/hello`.

Hiding the Implementation Finally, we hide the implementation of `hello` to other developers in order to enforce the architecture invariant which is that each plug-in should be used through `Db` (see Chapter 3). For this purpose we add an empty interface to the plug-in in the following way.

File **src/hello/Hello.mli**

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

    No function is directly exported: they are registered in {!Db.Hello}. *)
```

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

Indeed, thanks to `Makefile.plugin`, each plug-in is packed into a single module `$(PLUGIN_NAME)` (here `Hello`) and we simply export an empty interface for it.

We also have to explain to `Makefile.plugin` that we use our own interface `hello.mli` for `Hello`. For this purpose, in `Makefile`, we add the following line before including `Makefile.plugin`.

File **Makefile**

```
###########
# Hello #
###########
PLUGIN_ENABLE:=@ENABLE_HELLO@
PLUGIN_NAME:=Hello
PLUGIN_DIR:=src/hello
PLUGIN_CMO:= register
PLUGIN_NO_TEST:=yes
PLUGIN_HAS_MLI:=yes              # Add this single line
include Makefile.plugin
```

2.2.6 Testing

Frama-C provides a tool, called ptests, in order to perform non-regression and unit tests. This tool is detailed in Section 4.5. This section only covers basic use of ptests. First we have to create a test directory for `hello`

```
$ mkdir tests/hello
```

and, in `Makefile`, we have to remove the line `PLUGIN_NO_TEST:=yes`.

File **Makefile**

```
##########
# Hello #
##########
PLUGIN_ENABLE:=@ENABLE_HELLO@
PLUGIN_NAME:=Hello
PLUGIN_DIR:=src/hello
PLUGIN_CMO:= register
# PLUGIN_NO_TEST:=yes           # Remove this single line
PLUGIN_HAS_MLI:=yes
include Makefile.plugin
```

Now we can add the following test `hello.c` in directory `tests/hello`.

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

```
/* run.config
   OPT: -hello
*/
/* A test of the plug-in hello does not require C code anyway. */
```

It is possible to test the new plug-in on this file with the command

```
$ ./bin/toplevel.byte -hello tests/hello/hello.c
```

which should display

```
[preprocessing] running gcc -C -E -I. tests/hello.c
Hello Frama-C World!
```

The specific output of the plug-in `hello` is the last line.

It is also possible to use ptests to run tests automatically.

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

The above command runs the Frama-C toplevel on each C file contained in the directory `tests/hello`. For each of them, it also uses directives following `run.config` given at the top of files. Here, for the test `tests/hello/hello.c`, the directive specifies that the toplevel has to be executed with the option `-hello`. Below is the output of this command.

```
% Dispatch finished, waiting for workers to complete
% System error while comparing. Maybe one of the files is missing...
tests/hello/result/hello.res.log  or tests/hello/oracle/hello.res.oracle
% System error while comparing. Maybe one of the files is missing...
tests/hello/result/hello.err.log  or tests/hello/oracle/hello.err.oracle
% Comparisons finished, waiting for diffs to complete
% Diffs finished. Summary:
Run = 1
Ok  = 0 of 2
```

This result says that testing fails because it is not possible to compare the execution results with previously stored results (oracles). You have to execute:

```
$ ./bin/ptests.byte -update hello
```

Thus each time one executes `ptests.byte`, differences with the saved oracles are displayed. Furthermore, you can easily check whether the changes in plug-in `hello` are compliant with all existing tests. For example, if we execute one more time:

```
$ ./bin/ptests.byte hello
% Diffs finished. Summary:
Run = 2
Ok  = 2 of 2
```

This indicates that everything is alright.

Finally, you can also check if your changes break something else in the Frama-C kernel or in other plug-ins by executing ptests on all default tests with `make tests`.

Note to SVN users If you have write access to the SVN repository, you may commit your changes into the archive. Before that, you have to perform non-regression tests in order to ensure that your modifications do not break the archive.

So you must execute the following commands.

```
$ svn add ...   # Do not forget new oracles
$ svn up
$ make tests
$ emacs[5] Changelog
$ svn commit -m "informative message"
```

If you created any new files, use the `svn add` command to add them into the archive. The `svn up` command updates your local directory with respect to the root repository. The `make tests` command performs the non-regression tests. Finally, *if and only if the regression tests do not expose any problem,* update the file `Changelog` according to your changes and commit them thanks with the `svn commit` command.

2.2.7 Copyright your Work

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

If you want to redistribute plug-in `hello`, you have to choose a license policy for it (compatible with Frama-C). Section 4.19 provides details about how to proceed. Here, suppose we want to put the plug-in `hello` under the Lesser General Public License (LGPL) and CEA copyright, you simply have to edit the section "File headers: license policy" of `Makefile` with the following line:

File **Makefile**

```
CEA_LGPL= src/hello/*.ml* # ... others files
```

Now executing:

```
$ make headers
```

This adds an header on files of plug-in `hello` in order to indicate that they are under the desired license.

[5] Or what other text editor you want to use.

Chapter 3

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 [14] 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

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 platform has to provide services in order 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. Figure 3.1 summarizes it. Mainly this architecture is separated in three different parts:

- Cil (C Intermediate Language) [14] 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 their API take in the Frama-C kernel. See Section 3.4 for details.

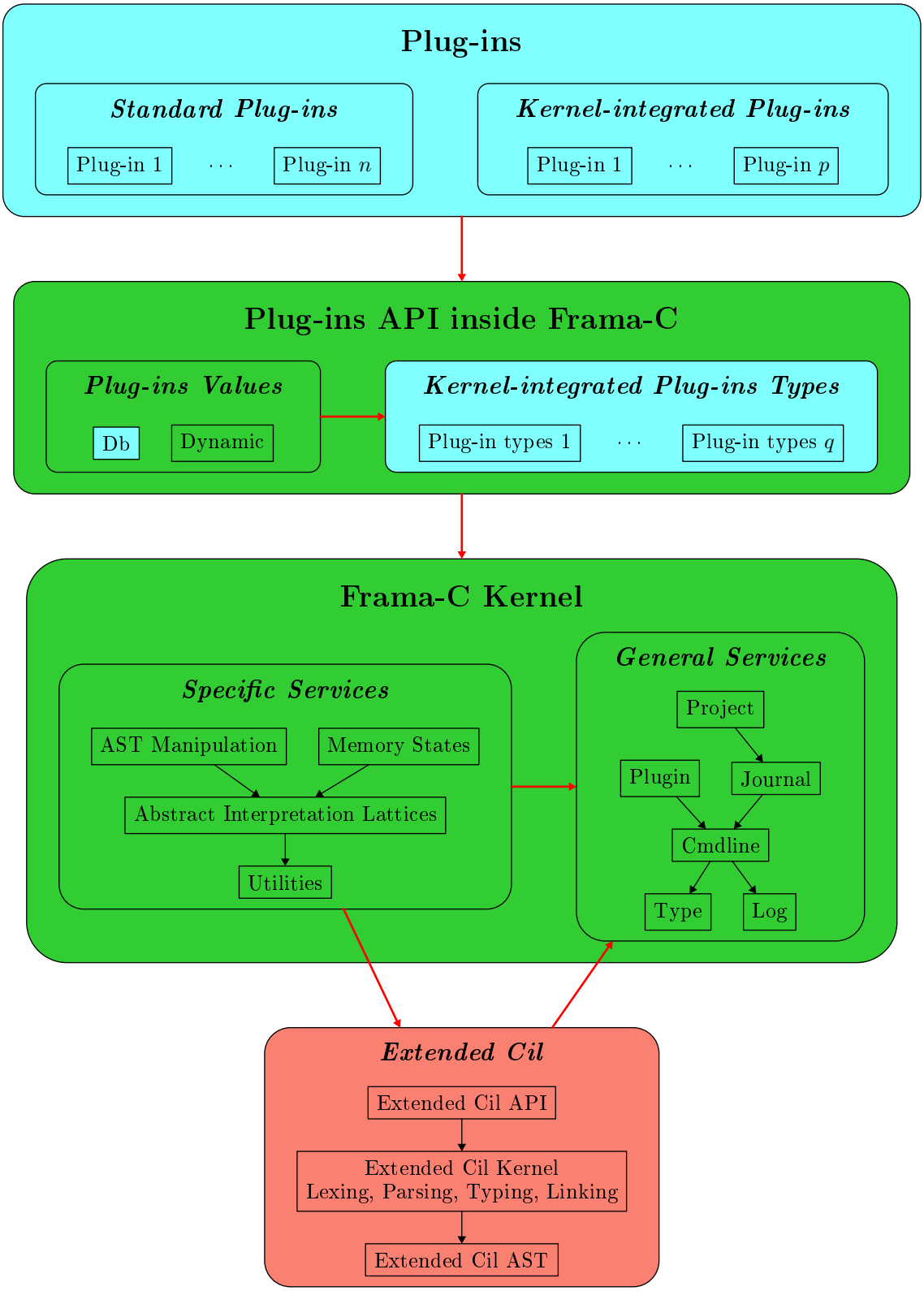

Figure 3.1: Architecture Design.

3.2 Cil: C Intermediate Language

Cil [14] 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 simple but useful miscellaneous datastructures and operations (mainly in module `Cilutil`); and
- 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 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 very 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 modules and does not use modules `Log` nor `Cmdline`.
- Module `Type` is a library handling OCaml types as first-class values. Such values are required by journalization and registration of dynamic values. See section 4.8 for details.
- Module `Journal` handles how Frama-C journalizes its actions. See section 4.9 for details.
- In directory `src/project`, the Frama-C kernel embeds a library, called `Project`, which permits the consistency of results for multi-analysis of multi-ASTs in a dynamic setting. See section 4.11 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 $p_1, \ldots, p_n$ and conversely. Mutual dependences between plug-ins are even possible. If a plug-in is designed to be used by another plug-in, its API has to be registered, either in module `Dynamic` or in module `Db`. This last way is only available for kernel-integrated plug-ins.

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 3.4.

Functionality	Standard plug-in	Kernel-integrated plug-in
dynamic linking	default	possible
static linking	possible	default
API in `Dynamic`	possible	possible
API in `Db`	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 Objective Caml 3.11 or higher as long as a supported architecture. See the Objective Caml manual [11] 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*

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.

Chapter 4

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 4.11.3);
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 of project, see Section 4.11.2).

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 3. Moreover plug-in development requires to use 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 the re 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. Frama-C source code is mostly provided in directories `cil` and `src`. The first one contains the source code of Cil [14] extended

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

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 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 2.2.1 and 2.2.4).

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 script `configure`. Each Frama-C user executes this script which checks his system to determine the most appropriate configuration: at the end of this configuration (if it is 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 to p additional but not highly required features (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 *Plugin 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 macro `check_plugin` sets the following variables: `FORCE_OCCURRENCE`, `REQUIRE_OCCURRENCE`, `USE_OCCURRENCE` and `ENABLE_OCCURRENCE`. `DYNAMIC_OCCURRENCE`.

The first one indicates if the user explicitly requires the availability of `occurrence` *via* setting the option `--enable-occurrence`. The second and third ones are used by others plug-ins in order to handle their dependencies (see Section 4.2.5). The fourth `ENABLE_OCCURRENCE` indicates the plug-in status (available, partially available or not available). At the end of these lines of code, it says if the plug-in should be compiled: if `--enable-occurrence` is set, then `ENABLE_OCCURRENCE` is `yes` (plug-in available); if `--disable-occurrence`, 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 `$default`). 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 plugins needs additional tools or libraries to be fully functional. The configure script takes care of that 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 plugin 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. 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]. `configure_library` takes three arguments. The first one is the (uppercase) name of the library, the second one is a filename which is used by the script to check the availability of the library. In case there are multiple locations possible for the library, this argument can be a list of filenames. In this case, the argument must be properly quoted (*i.e.* enclosed in a `[, ]` pair). Each name is checked in turn. The first one which corresponds to an existing file is selected. If no name in the list corresponds to an existing file, the library is considered to be unavailable. The last argument is a warning message to display if a configuration problem appear (usually because the library does not exist). Using these arguments, the script checks the availability of the library.

[2]For tools, there is a macro `configure_tool` which works in the same way as `configure_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 Objective Caml 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* Lablgtksourceview *(used to have a better rendering of C sources in the GUI) can be found either as part of* Lablgtk2 *or as an independent library. This is checked through the following command:*

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

Moreover, we want to distinguish the two cases, as the independent library denotes a legacy version of Lablgtksourceview*, which has been merged with* Lablgtk2*. This is done by pattern-matching on the variable* `SELECTED_GTKSOURCEVIEW` *as shown below:*

```
case $SELECTED_GTKSOURCEVIEW in
$OCAMLLIB/lablgtksourceview/lablgtksourceview.$LIB_SUFFIX)
  HAS_LEGACY_GTKSOURCEVIEW=yes
  ;;
esac
```

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 *[9]. So, just after its declaration, there are the following lines in* `configure.in`*.*

```
plugin_require_external(gui,lablgtk)
```

This line specify 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 *plugin*1 needs *plugin*2.
- `plugin_use($plugin1$,$plugin2$)` states that *plugin*1 can be used in absence of *plugin*2, but requires *plugin*2 for full functionality.

There can be mutual dependencies between plugins. This is for instance the case for plugins `value` and `from`.

4.2.6 External plugins

External plug-ins can have their own configuration file, and can rely on the macros defined for Frama-C. In addition, as mentioned in section 4.4.3, 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 plugins. 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 plugin together with the main Frama-C), or by calling an installed Frama-C (when installing the plugin separately). The inclusion of `configure.ac` needs to be guarded to prevent multiple inclusions, as the configuration file of the plugin might itself be included by `configure.in` (see section 4.4.3 for more details).

The configuration of the plugin itself can then proceed as described above. References to specific files in the plugin source directory should be guarded with the following macro:

```
PLUGIN_RELATIVE_PATH(file)
```

If the external plugin 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 plugin can have dependencies upon previously installed plugins. However two separately installed plugins can not be mutually dependent on each other. However, they can be compiled together with the main Frama-C sources using the `--enable-external` option of `configure` (see section 4.4.3 for more details).

In addition, it 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.

4.3 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 Ael -warn-error A -dtypes  -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/cxx_types -I src/gui -I lib/plugins
-I lib -I src/lib -I src/project -I src/buckx -I external -I src/project
-I src/buckx -I cil/src -I cil/src/ext -I cil/src/frontc -I cil/src/logic
-I cil/ocamlutil -g  src/kernel/db.ml
```

In order to integrate a new plug-in, you have to extend section "Plug-ins". For this purpose, you have to include `Makefile.plugin` for each new plug-in (hence there are as many lines `include 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 `=`[3].

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

```
#####################
# Value analysis #
#####################
PLUGIN_ENABLE:=@ENABLE_VALUE@
PLUGIN_NAME:=Value
PLUGIN_DIR:=src/value
PLUGIN_CMO:= state_set kf_state eval kinstr register
PLUGIN_GUI_CMO:=value_gui
PLUGIN_HAS_MLI:=yes
PLUGIN_NO_TEST:=yes
PLUGIN_UNDOC:=value_gui.ml
include Makefile.plugin
```

[3] Using `:=` 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 [8]).

As said above, you cannot use the parameters of `Makefile.plugin` anywhere in `Makefile`. You can yet use some plugin-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 modify any other part of `Makefile` or `Makefile.plugin`.

4.4 Plug-in Specific Makefile

Prerequisite: *knowledge of* make.

In this section, we detail how to add a Makefile to a plug-in. Section 4.4.1 introduces how to write a Makefile for its own plug-in thanks to `Makefile.dynamic` while Section 4.4.2 explains how to integrate it in `Makefile`.

4.4.1 Using `Makefile.dynamic`

In this section, we detail how to write a Makefile for its own 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 to write it. This file is installed in the Frama-C share directory. So for writting 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 2.1.2). 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_LIB` 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.*

For running your specific Makefile, you must have properly installed Frama-C before.

Eventually it may be required 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`.

4.4.2 Calling a Plug-in Specific Makefile from the Frama-C Makefile

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

Even if you are writing a kernel-integrated plug-in, it is useful to have your plug-in specific Makefile. For instance, it allows you to easily release your plug-in independently of Frama-C. However, if your version of Frama-C is changing frequently, it is useful to compile together Frama-C and your plug-in without installing Frama-C each time. To reach this goal, you have to mix the integration in `Makefile` described in Section 4.3 and the solution presented in this section: in the section "Plug-ins" of `Makefile` you just have to set few variables before including your plug-in specific Makefile

Example 4.5 *For compiling the plug-in `Ltl_to_acsl`, the following lines are added into `Makefile`.*

```
PLUGIN_ENABLE   :=@ENABLE_LTL_TO_ACSL@
PLUGIN_DIR      :=src/ltl_to_acsl
PLUGIN_DYNAMIC  :=@DYNAMIC_LTL_TO_ACSL@
DISTRIB_FILES += $(PLUGIN_DIR)/Makefile
include $(PLUGIN_DIR)/Makefile
```

4.4.3 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 to be recompiled and tested together with Frama-C's kernel. For that, Frama-C must be aware of the existence of the plug-in. This can be done in two ways:

- All sub-directories of `src/` 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 plugin

In the first case, the plug-in behaves as any other built-ins plugins: `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 plugin is added to the list of external plugins at configure's time. If the plugin has its own configure, it is run as well.

4.5 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*). Test is successful if and only if there is no difference.

Actually the number of results is twice that the number of tests because standard and error outputs are compared separately.

First Section 4.5.1 shows how to use ptests. Next Section 4.5.2 explains how to configure tests through directives. Last Section 4.5.3 describes how to set up various testing goals for the same test base.

4.5.1 Using ptests

The simplest way of using ptests is through `make tests` which is roughly equivalent to

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

This command runs all the tests belonging to a sub-directory of directory `tests`. ptests also accepts specific *test suites* in arguments. A test suite is either a name of a sub-directory in directory `tests` or a filename (with its complete path).

Example 4.6 *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 in order to customize one test sequence. These options are detailed in Section 5.4.

Example 4.7 *If 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.

4.5.2 Configuration

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

The places are:

- inside file `tests/test_config`;
- inside file `tests/`*subdir*`/test_config` (for each sub-directory *subdir* of `tests`); or

- inside each test file, in a special comment of the form

```
/* 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 5.4.

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
*/
```

They say 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
```

4.5.3 Alternative Testing

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

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

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

```
$ ./bin/ptests.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, tests relying on custom dynamically-linked plug-ins must either be always executed in bytecode or provide an alternative configuration named `no_native_dynlink` which performs the test in bytecode, in order for the test to be launched on architectures which do not support native dynlink. If those tests have other special configurations, they must similarly provide for each them a corresponding `<special_name>_no_native_dynlink`.

4.6 Plug-in General Services

Module `Plugin` provides an access to some general services available for all plug-ins. The goal of this module is twofold. First, it helps the developper to use some features of Frama-C. Second, it allows to provide to the end-user a common set of features for 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 descr = "functional dependencies"
   end)
```

Applying this functor mainly provides two different services. First it gives access to functions for printing messages in a Frama-C-compliant way (see Section 4.7). Second it allows to define plug-in specific parameters available as option on the Frama-C command line for the end-user (see Section 4.12).

4.7 Logging Services

Displaying results of plug-in computations to users, warning them at hypothesis taken by static analysis, reporting incorrect inputs, all these tasks are easy to think about, but turn to be difficult to handle in a pretty way. As soon as your plug-in is registered (see Section 4.6 above), though, you automatically benefit from many logging facilities provided by the kernel. More, when logging through these services, the messages of your plug-in kindly interfere with messages from other plug-ins, in a consistent way from the user's point view.

As a general rule, you should *never* writes to standard output and error channels through Objective Caml 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 descr = "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.

4.7.1 From `printf` to `Log`

Below is a simple example of how to make a `printf`-based code towards being `Log`-compliant. The original code, extracted from the Occurrence plugin 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 delegates 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))
```

That's all!

4.7.2 Log Quick Reference

The logging routines for your plugins consists in an implementation of the Log.Messages interface, which is included in the Plugin.S interface returned by the registration of your plugin. 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 like 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 just discover 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 display a failure message otherwise. Finally, it returns the condition value.

  A typical usage is for example:

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

4.7.3 Logging Routine Options

Logging routines have optional parameters to modify their general behavior. Hence their complicated 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.

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 sent to registered callbacks when emitted. See Section 4.7.4 below for more details on how to globally modify such a behavior. During the message construction, you can locally modify the emission process with the following options:

~emitwith:f suppress the echo and send 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.

4.7.4 Advanced Logging Services

Message Emission

During message construction, the message content is echoed in the terminal. This echo may be delayed until message completion when ~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 takes as argument a (polymorphic) formatting string, followed by the formatting parameters, and finally returns unit. It is also possible to catch the generated message, and to pass it to a continuation that finally returns a value different that 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 with the normal way, the message is passed to the continuation f. Hence, f has type $event \rightarrow \alpha$, and the log routine returns α.

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

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

Generic Routines

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

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

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:

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

Channel Management

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

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

Services provided at plugin registration are convenient shortcuts to low-level logging service onto channels. The Log interface allow 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 any direct access to Pervasives.stdout. However, Log interface allow you to create such channels by your own, in addition to the one automatically created for your plug-in.

new_channel *name*
: This create a new channel. There is only one channel *per* name, and the function returns the already created 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` <*options*> `"..."`
This routine is similar to the log one.

with_log_channel `channel f ?kind ?prefix` <*options*> `"..."`
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:

print_on_output `"..."`
The routine immediately lock the output of Log for printing the provided message. All message echoes are delayed until the routine actually returns. Notification to listeners is not delayed, however.

print_delayed `"..."`
This variant lock 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 occur.

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

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

4.8 Types as first class values

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

4.9 Journalization

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

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-ins 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 4.11 *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 Objective Caml.

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 even if it is possible to put a single file in another directory or to put a single type in an existing file (like `src/kernel/db_types.mli`).

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 4.12 *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` [4].

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.*

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`.

Shortly, 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`.

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

Registering a value The signature of `Dynamic.register` is as follows.

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

The first argument is the name of the plug-in registering the value and the second one is a binding name of the registered OCaml value. It must not be used for value registration anywhere else in the Frama-C world. It is required for another plug-in in order to access to this value (see next paragraph). The third argument is the so-called *type value* of the registered value, *i.e.* an OCaml value representing its type (see Section 4.8 for additional details). It is required for safety reasons when accessing to the registered value (see next paragraph). Predefined type values exist in modules `Type` (for usual OCaml types like `int`) and `Kernel_type` (for usual Frama-C types like `Cil_types.varinfo`). The third argument is the value itself.

Example 4.13 *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" (Type.func Type.unit Type.unit) run
```

If the string `"Hello.run"` is already used to register a dynamic value, then the exception `Type.AlreadyExists` is raised during plug-in initialization (see Section 4.13).

The function call `Type.func Type.unit Type.unit` returns the type value of `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`.*

Calling a previously-registered function 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 4.14 *Here is how the previously registered function `run` of `Hello` may be applied.*

```
let () = Dynamic.get ~plugin:"Hello" "run" (Type.func Type.string Type.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`.*

4.11 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 [15]. Section 4.11.1

first introduces the general principle of project. Then Section 4.11.2 explains how to simply use them. Section 4.11.3 introduces the so-called *internal states* for which registration is detailed in Sections 4.11.4, 4.11.5 and 4.11.6. Section 4.11.4 is dedicated to so-called *datatypes*. Section 4.11.5 is dedicated to the internal states themselves. Section 4.11.6 is dedicated to low-level registration. Finally Section 4.11.7 shows how to handle projects and internal states in a clever and proper way.

4.11.1 Overview and Key Notions

In Frama-C, many (mostly global) data are attached to an AST. For example, there are the AST itself, options of the command line (see Section 4.12) and tables containing results of analyses (Frama-C extensively uses memoization [12, 13] in order to avoid re-computating analyses). The set of all these data is called a *project*. It is the only value savable on the disk and restorable by loading.

Several ASTs can exist at the same time in Frama-C and thus several projects as well; there is one AST per project. Besides each data has one value per AST: thus there are as many values for each data as projects/ASTs.

The set of all the projects stands for *the internal state of* Frama-C : it consists of all the ASTs defined in Frama-C and, for each of them, the corresponding values of all the attached data.

A related notion is *internal state* of a data d. That is the different values of d in projects: for each data, the cardinal of this set is equal to the cardinal of the internal state of Frama-C (*i.e.* the number of existing projects).

These notions are summarized in Figure 4.1. One row contains the value of each data for a specific project and one line represents an internal state of a specific data.

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

Figure 4.1: Representation of the Frama-C Internal State.

4.11.2 Using Projects

Actually Frama-C maintains a current project (`Project.current ()`) and a current AST (`Ast.get ()`) which all operations are automatically performed on. But sometimes plug-in developers have to explicitly use them, for example when the AST is modified (usually through the use of a copy visitor, see Section 4.14) or replaced (*e.g.* if a new one is loaded from disk).

An AST must never be modified inside a project. If such an operation is required, you must either create a new project with a new AST, usually by using `File.init_project_from_cil_file` or `File.init_project_from_visitor`; or write the following line of code (see Section 4.11.7):

```
Project.clear
  ~only:
    (Project.Selection.singleton Ast.self Kind.Only_Select_Dependencies)
  ()
```

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 4.15 *Suppose that you saved the current project into file `foo.sav` in a previous* Frama-C *session*[5] *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 _ →
  exit 1
```

displays

```
false
true
false
```

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

An alternative to the use of `Project.set_current` is the use of `Project.on` which 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 4.16 *The following code is equivalent to the one given in Example 4.15.*

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

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

It displays

```
false
true
false
```

4.11.3 Internal State: Principle

If a data should be part of the internal state of Frama-C, you must register it as an internal state (also called a *computation* because it is often related to memoization).

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

Internal 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.
- It is part of the information saved on disk for restoration in a later session.
- It may be part of a *selection* which is, roughly speaking, a set of internal states. Which such a selection, you can control which internal states project operations are applied on (see Section 4.11.7). For example, it is possible to clear all the internal states which depend of the value analysis.
- It is possible to ensure inter-analysis consistency by setting internal state dependencies. For example, if the entry point of the analysed program is changed (using `Globals.set_entry_point`), all the results of analyses depending of it (like the value analysis) are automatically reset. If such a reset was not performed, the results of the value analysis would be not consistent with the current entry point.

  Example 4.17 *Suppose that the value analysis has previously been computed.*

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

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

  ```
  true
  false
  ```

Internal State Registration: Overview

For registering a new internal state, functor `Project.Computation.Register` is provided. Actually it is quite a low-level functor. Higher-level functors are provided to the developer by modules `Computation` and `Cil_computation` that register internal states in a simpler way. They internally apply the low-level functor in a proper way. Module `Computation` provides internal state builders for standard OCaml datastructures like hashtables whereas

`Cil_computation` does the same for standard Cil datastructures (like hashtables indexed by AST statements)[6].

> Registering a new internal state must be performed before parsing command line option. For this purpose, you can register your function through `Cmdline.run_after_extending_stage` (see Section 4.13).

Section 4.11.5 details how to register a new computation.

The registration of a data of type τ requires to register the type τ itself as a *datatype* using functor `Project.Datatype.Register`. A datatype is a type that is aware of projects. Similarly to computations, module `Datatype` (resp. `Cil_datatype`) provides pre-defined datatypes and datatypes-builder for elaborated Cil types[7]. Section 4.11.4 details how to register a new datatype.

Example 4.18 *If you have to register a reference to a boolean initialized to `false` as an internal state, you have to write the following code.*

```
module My_Bool_Ref =
  Computation.Ref
    (struct include Datatype.Bool let default = false end)
    (struct let dependencies = [] let name = "My_Bool_Ref" end)
```

4.11.4 Registering a New Datatype

In order to register a new datatype, you have to apply functor `Project.Datatype.Register` which is a quite low-level functor. In most cases, a direct application of this functor is actually not required because some higher-level and easier-to-use functor does it for you. We explain here the three different possible situations.

Simple registration If the datatype to register is not hash-consed[8] or does not contain hash-consed ones (*i.e.* it is not itself hash-consed or composed of `Cil_types.fundec`, or any Frama-C abstract interpretation type), the easiest way of registering a new datatype d is to apply one of functors `Persistent` or `Imperative` of module `Project.Datatype`, depending on the nature of d (whether it is persistent). The only difference between both functors is that you have to provide a copy function for imperative (*i.e.* mutable) datatypes. This copy function is only used by `Project.copy`.

Example 4.19 *For registering a type `t` containing an immutable field `a`, just do*

```
type a = { a : int }
Project.Datatype.Persistent(struct type t = a let name = "a" end)
```

If the field `a` is mutable, just write

```
type a = { mutable a : int }
Project.Datatype.Imperative
  (struct
     type t = a
```

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

[7] On the contrary to computations, these types are either mutable or persistent because the registration of a type may require the registration of its subtypes (in the sense of syntactically contained in).

[8] Hash-consing is a programming technique saving memory blocks and speeds up operations on datastructures when sharing is maximal [7, 10, 2, 5].

```
    let copy x = { a = x.a }
    let name = "a"
  end)
```

Using predefined datatypes or datatype builders For most useful types, the corresponding datatypes are already provided in modules `Datatype` (*e.g.* `Datatype.Int` for type `int`) and `Cil_datatype` (*e.g.* `Cil_datatype.Stmt` for type `Cil_types.stmt`). Moreover both modules provides a bunch of functors which help to build complex datatypes when `Project.Datatype.Persistent` and `Project.Datatype.Imperative` cannot be used. Interfaces of modules `Datatype` and `Cil_datatype` provided all the available modules.

Example 4.20 *For registering the type of an hashtable associating varinfo to list of kernel functions, it is not possible to apply functor `Project.Datatype.Imperative` because a kernel function is composed of `Cil_types.fundec`. But it is still easy to perform the registration thanks to predefined functors:*

```
Cil_datatype.VarinfoHashtbl(Datatype.List(Kernel_function.Datatype))
```

Direct use of the low-level functor In some cases (*e.g.* registering a new variant type composed of a kernel function), applying functor `Project.Datatype.Register` is required. As input, one has to provide:

- The type itself.
- How to copy it (usually, rebuild the structure by applying the right copy and copy functions on subterms).
- What is the representation of the type in memory and how to rebuild it while unmarshaling. Pending further information, define descr by `let descr = Project.no_descr`.
- A name for the datatype.

Example 4.21 *The type of postdominators is the following variant.*

```
type postdominator = Value of Cilutil.StmtSet.t | Top
```

The corresponding registered datatype used to store results of the postdominator computation is the following (see file `src/postdominators/compute.ml`).

```
Project.Datatype.Register
  (struct
     type t = postdominator
     let map f = function
       | Top → Top
       | Value set → Value (f set)
     let copy = map Cil_datatype.StmtSet.copy
     let descr = Project.no_descr
     let name = "postdominator"
   end)
```

4.11.5 Registering a New Internal State

Here we explain how to register and use an internal state in Frama-C. Registration through the use of low-level functor `Project.Computation.Register` is postponed in Section 4.11.6 because it is more tricky and rarely useful.

In most non-Frama-C applications, a state is a (usually global) mutable value. One can use it in order to store results of the analysis. For example, inside Frama-C, the following piece of code would use value `state` in order to memoize some information attached to statements.

```
open Cilutil
type info = Kernel_function.t * Cil_types.varinfo
let state : info StmtHashtbl.t = StmtHashtbl.create 97
let compute_info = ...
let memoize s =
  try StmtHashtbl.find state s
  with Not_found → StmtHashtbl.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 internal state. A direct consequence is that it is not saved on the disk. For this purpose, one has to transform the above code into the following one.

```
module State =
  Cil_computation.StmtHashtbl
    (Datatype.Couple(Kernel_function.Datatype)(Cil_datatype.Varinfo))
    (struct
       let size = 97
       let name = "state"
       let dependencies = [ Db.Value.self ]
     end)
let compute_info = ...
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 much more complicated (it uses a functor application) but the use of state is simpler. Actually what has changed?

1. To declare a new internal state, apply one of the predefined functors in modules `Computation` or `Cil_computation` (see interfaces of these modules for the list of available modules). Here we use `StmtHashtbl` which provides an hashtable indexed by statements. The type of values associated to statements is a couple of `Kernel_function.t` and `Cil_types.varinfo`. The first argument of the functor is the datatype corresponding to this type (see Section 4.11.4). The second argument provides some additional information: the initial size of the hashtable (an integer similar to the argument of `Hashtbl.create`), a name for the resulting state and its dependencies. This list of dependencies is built upon values `self` which are provided by the application of the low-level functor `Project.Computation.Register`. This value is called the *kind* of the internal state (also called *state kind* and can be used for this purpose. Roughly speaking, it represents the internal state itself.

2. 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 A plug-in p may want to export its state kind (in the previous example, that is value `State.self`). This exportation offers the possibility to other plug-ins to depend on this state. It is a bit tricky because the state kind has to be accessible through `Db`.

There is two ways to achieve such a goal. First, the internal state has to be compiled before `Db`: usually the internal state has to be somewhere in directory p`_types` (see Section 4.10.1). Actually it is quite difficult because the computation of the internal state may be complex and so should not be in p`_types`.

The second way is to put a delayed reference to `self` (*i.e.* the state kind) in `Db` thanks to `Project.Computation.dummy` which provides a dummy kind. This reference is going to be initialized at the plug-in initialization time (see Section 4.13). Now if another plug-in has an internal state which depends on `!Db.My_plugin.self`, it cannot put the dependence when the functor creating the state is applied because the order of plug-in initialization is not specified (see Section 4.13 for more details about initialization steps). So you have to postpone the addition of this dependency; usually by using the function `Cmdline.run_after_extending_stage` (see Section 4.13).

Example 4.22 *Plug-in* `from` *postpones its internal state in the following way.*

File ***src/kernel/db.mli***

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

File ***src/kernel/db.ml***

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

File ***src/from/register.ml***

```
module Functionwise_Dependencies =
  Kernel_function.Make_Table
    (Function_Froms.Datatype)
    (struct
       let name = "functionwise_from"
       let size = 97
       let dependencies = [ Db.Value.self ]
     end)
let () =
  (* performed at module initialization runtime. *)
  Db.From.self ← Functionwise_Dependencies.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.Datatype)
    (struct
       let name = "Pdg.State"
       let dependencies = [] (* postponed *)
```

```
        let size = 97
    end)
let () =
  Cmdline.run_after_extended_stage
    (fun () → Project.Computation.add_dependency Tbl.self !Db.From.self)
```

For standard plug-ins, it is possible to register state kinds in the same way that any other value through `Dynamic.register` (see Section 4.10.2).

4.11.6 Direct Use of Low-level Functor `Project.Computation.Register`

Functor `Project.Computation.Register` is the only functor which really registers an internal 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 no direct use of this functor.

This functor takes three arguments. The first and the third ones respectively correspond to the datatype and to information (name and dependencies) of the internal states: they are similar to the corresponding arguments of the high-level functors (see Section 4.11.5).

The second argument explains how to handle the *local version* of the value of the internal state (under registration). Indeed here is the key point: from the outside, only this local version is used for efficiency purpose. It would work if projects do not exist. Each project knows a *global version*: the set of these global versions is the so-called *internal states*. 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_if_project` (how to clear each value of type `project` in the state if any).

The following invariants must hold:[9]

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

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

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

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

Invariant 4.1 ensures that there is no sharing with any fresh value of a same internal state: so each new project has got its own fresh internal 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 4.23 *To illustrate this, we show how functor `Computation.Ref` (registering a state corresponding to a reference) is implemented.*

```
module Ref(Data:REF_INPUT)(Info:Signature.NAME_DPDS) = struct
  type data = Data.t
  let create () = ref Data.default
  let state = ref (create ())
```

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

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

```
include Project.Computation.Register
  (Datatype.Ref(Data))
  (struct
    type t = data ref (* we register a reference on the given type *)
    let create = create
    let clear tbl = tbl ← Data.default
    let get () = !state
    let set x = state ← x
    let clear_if_project _ _ = 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 ← Data.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 `Computation.Ref` *are provided which hide you such implementations.*

4.11.7 Selections

Most operations working on a single project (*e.g.* `Project.clear` or `Project.on`) have two optional parameters `only` and `except` of type `Project.Selection.t`. These parameters allow to specify which internal states the operation applies on:

- If `only` is specified, the operation is *only* applied on the selected internal states.
- If `except` is specified, the operation is applied on all internal states, *except* the selected ones.
- If both `only` and `except` are specified, the operation *only* applied on the `only` internal states, *except* the `except` ones.

A *selection* is roughly speaking a set of internal states. Moreover it handles states dependencies (that is the specificity of selections).

Example 4.24 *The following statement clears all the results of the value analysis and all its dependencies in the current project.*

```
Project.clear
  ~only:(Project.Selection.singleton Db.Value.self Kind.Select_Dependencies)
  ()
```

The argument `Kind.Select_Dependencies` *says that we also want to clear all the states which depend on the value analysis.*

Use selections carefully: if you apply a function f on a selection s and if f handles a state which does not belong to s, then the Frama-C state becomes lost and inconsistent.

Example 4.25 *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 4.12).*

```
Project.on ~only:(Plugin.get_selection ()) p f ()
```

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

4.12 Command Line Options

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

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

In Frama-C, a parameter is actually a structure implementing signature `Plugin.Parameter` in order to handle projects: each parameter is indeed an internal state (see Section 4.11.5). Actually a bunch of signatures extended `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 parameters.

Implementing such an interface is very easy thanks to internal 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 is a list of most useful 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. `IndexedVal` builds an option for any datatype τ as soon as you provides a partial function from strings to value of type τ.

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 4.26 *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 descr = "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 `-security-lattice` *of the plug-in* `security` *is the module* `Lattice` *(defined in the file* `src/security/options.ml`*). It is implemented as follow.*

```
module Lattice =
  String
    (struct
       let option_name = "-security-lattice"
       let default = "weak"
       let arg_name = ""
       let descr = "specify security lattice"
     end)
```

So it is a string parameter initialized by default to the string `"weak"`*. The field* `arg_name` *is set to the empty string: so a default name for the argument of this option will be choose by* Frama-C *while displaying the help of this parameter. The field* `descr` *is the help message. The Interface for this module is simple:*

```
module Lattice: Plugin.STRING
```

Recommendation 4.3 *Parameters of a same plug-in* `plugin` *should belong to a module called* `Options` *or* `Plugin_options` *inside the plug-in directory.*

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

Example 4.27 *To know whether* Frama-C *uses unicode, just write*

```
Parameters.UseUnicode.get ()
```

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

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

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

Using a parameter of a plug-in p outside p (for instance in another plug-in) requires the use of module `Parameters.Dynamic` because the module defining the parameter is not visible from the outside of its plug-in, so the option is accessible by any other plug-ins (and by the Frama-C kernel as well). Functions of sub-modules of module `Parameters.Dynamic` takes a string in argument which is the option name associated with the parameter.

Example 4.28 *Outside the plug-in* `From`*, just write*

```
Parameters.Dynamic.Bool.get "-calldeps"
```

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

4.13 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 at interactive more deeply with the kernel process. It can be also usefull at 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 (see Section 4.11.5).

In general, a hook does nothing, except when some parameters have been positionned. These parameters comes from the command-line (see Section 4.12) or from the configuration panel of the GUI. However, from the initialization process point of view, both source of parameters are treated as *command line* ones. Thus, registering and executing a hook is tiedly coupled with handling the command line parameters.

The parsing of the command line parameters is performed in several *stages*, each one dedicated to specific operations. For instance, executing journal should be performed after loading dynamic plugins, 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 stages of Frama-C consists are described below, following their execution order.

1. **The Static Link Stage:** the statically linked Frama-C compilation units are initialized, following some *partially* specified order. More precisely:

   (a) the architecture dependencies depicted on Figure 3.1 (cf. p. 25) 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;

   (b) 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;

   (c) 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 module, see Figure 3.1) performs a very early configuration stage, such as checking if journalization has to be activated (cf. Section 4.9), or setting the global verbosity and debugging levels.

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

   (a) first, 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.

3. **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-journal` 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 command line options parsed before;

4. **The Extended State:** 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 comes from postponed internal-state dependencies (see Section 4.11.5).

   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.

5. **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.

6. **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 4.9). 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 that the default one. In such a case:

   They must call the function `Cmdline.is_going_to_load` while initialising.

7. **The Configuring Stage:** this is normal place where plug-ins performs 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. So 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 a unspecified order.

8. **The Main Stage:** this is the step where plug-ins actually run their main entry points registered through `Db.Main.extend`. Thus, you shall consider that this stage is the one where these hooks are executed.

4.14 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 4.14.6, which the reader already familiar with Cil is invited to read carefully.

4.14.1 Entry Points

Cil offers various entry points for the visitor. They are functions called `Cil.visitCil`*AstType* where *astType* is a node type in the Cil's AST. Such a function takes as argument an instance of a `cilVisitor` and an *astType* and gives back an *astType* transformed according to the visitor. The entry points for visiting a whole `Cil_types.file` (`Cil.visitCilFileCopy`, `Cil.visitCilFile` and `visitCilFileSameGlobals`) are slightly different and do not support all kinds of visitors. See the documentation attached to them in `cil.mli` for more details.

4.14.2 Methods

As said above, there is a method for each type in the Cil AST (including for logic annotation). For a given type *astType*, the method is called `v`*astType*[10], 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

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

be done for the children of the resulting AST node, and is presented in the next section. In addition, there are two modes for visiting a `varinfo`: `vvdec` to visit its declaration, and `vvrbl` to visit its uses. More detailed information can be found in `cil.mli`.

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

4.14.3 Action Performed

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

- `SkipChildren` the visitor do 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` only meaningful for the copy visitor. Indicates that the visit should goes on with the children, but only perform a fresh copy of the nodes
- `DoChildrenPost(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.
- `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 `vstmt_aux` and `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.

4.14.4 Visitors and Projects

The visitors takes an additional argument, which is the project in which the transformed AST should be put in. Note that an in-place visitor (see next section) should operate on the current project (otherwise, two projects would share the same AST). If this is not the case, it is up to the developer to ensure that the copy is done by other means, so that there is no sharing.

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.

4.14.5 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 4.2.

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 4.2: Indices of AST nodes.

Last, when using a copy visitor, the actions (see previous section) `SkipChildren` and `ChangeTo` must be used with care, *i.e* one has to ensure that the children are fresh. Otherwise, the new AST will share some nodes with the old one. Even worse, in such a situation the new AST might very well be left in an inconsistent state, with uses of shared node (*e.g.* a `varinfo` for a function `f` in a function call) which do not match the corresponding declaration (*e.g* the `GFun` definition of `f`).
When in doubt, a safe solution is to use `JustCopy` instead of `SkipChildren` and `ChangeDoChildrenPost(x,fun x -> x)` instead of `ChangeTo(x)`.

4.14.6 Differences Between the Cil and Frama-C Visitors

As said in Section 4.14.2, `vstmt` and `vglob` should not be redefined. Use `vstmt_aux` and `vglob_aux` instead. Be aware that the entries corresponding to statements and globals in Frama-C tables are considered more or less as children of the node. In particular, if the method returns `ChangeTo` action (see Section 4.14.3), it is assumed that it has taken care of updating the tables accordingly, which can be a little tricky when copying a `file` from a project to another one. Prefer `ChangeDoChildrenPost`. On the other hand, a `SkipChildren`

action implies that the visit will stop, but the information associated to the old value will be associated to the new one. If the children are to be visited, it is undefined whether the table entries are visited before or after the children in the AST.

4.14.7 Example

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

```
open Cil_types
open Cil

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 = function
        BinOp((Div|Mod),_,e2,_) →
          let t = Cil.typeOf e2 in
          let logic_e2 =
            Logic_const.mk_dummy_term
              (TCastE(t,Logic_utils.expr_to_term ~cast:true e2)) t
          in
          let assertion = Logic_const.prel (Rneq,logic_e2,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 is currently visited
             in the current_stmt method, which returns None when outside
             of a statement, e.g. when visiting a global declaration. Here, it
             necessarily returns Some. *)
          let stmt = Extlib.the (self#current_stmt) in
          (* Since we are copying the file in a new project, we can't 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 stmt [] ~before:true assertion)
            self#get_filling_actions;
          (* Do not forget to recurse on the children of the
             division. *)
          DoChildren
      | _ → DoChildren (* do not do anything on other expressions
                          (except visiting their children)*)
end

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

4.15 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 4.11.2).

So all the logical annotations (including those set by the users) are put in projectified states. So each time a plug-in wants either to access to an annotation or to add a new one or to modify an existing one, it *must* uses these states and not the annotations directly stored in the AST. These states are the following.

- `Globals.Annotations` which contains all the globals annotations (*e.g.* global invariants).
- `Annotations` which contains annotations associated with statements (*e.g.* assertions).
- The field `spec` of kernel functions which contains annotations associated with functions (*e.g.* preconditions).
- `Properties_status` should be used to get or to modify the status of annotations.
- `Db.Properties` contains numbers operations over annotations.

4.16 Locations

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

In Frama-C, different representations of C locations exist. Section 4.16.1 presents them. Moreover, maps indexed by locations are also provided. Section 4.16.2 introduces them.

4.16.1 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 variable and offsets in bytes.
- Type `location` 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_Byte.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 function* `Locations.valid_enumerate_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).

4.16.2 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`).

4.17 GUI Extension

Prerequisite: *knowledge of* Lablgtk2.

Each plug-in can extend the Frama-C graphical user interface (aka *GUI*) in order to support its own functionalities in the Frama-C viewer. For this purpose, a plug-in developer has to register a function of type `Design.main_window_extension_points → unit` thanks to `Design.register_extension`. The input value of type `Design.main_window_extension_points` is an object corresponding to the main window of the Frama-C GUI. It provides accesses to the main widgets of the Frama-C GUI and to several plug-in extension points. The documentation of the class type `Design.main_window_extension_points` is accessible through the source documentation (see Section 4.18).

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 computations taking time have to call time to time function `!Db.progress` in order to keep the gui reactive.

Mainly that's all! The gui implementation uses Lablgtk2 [9]: so you can use any Lablgtk2-compatible code in your gui extension. A complete example of gui extension may be found in 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 behaviour is not specified but it could be quite difficult to understand for an user.

4.18 Documentation

Prerequisite: *knowledge of* ocamldoc.

Here we present some hints on the way to document your plug-in. First Section 4.18.1 introduces a quick general overview about the documentation process. Next Section 4.18.2 focus on the plug-in source documentation. Finally Section 4.18.3 explains how to modify the Frama-C website.

4.18.1 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 some times. So command `make html` 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.

4.18.2 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 itself no function: the only visible plug-in interface should be in `Db`.

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 here required for compilation purpose. 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 `Makefile` (see Section 5.3.3).

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

4.18.3 Website

Target readers: *developers with a CVS access.*

The html sources of the Frama-C website belong to directory `doc/www/src`. Each plug-in available through the Frama-C website (`http://www.frama-c.cea.fr`) may have its own webpage.

For each plug-in p, the source of its webpage should be called p`.prehtml`: this file is preprocessed by the makefile generating the whole website. The format of this page looks like below.

```
<#head>
<h1>Impact plug-in</h1>

... Plug-in description ...
<#foot>
```

This page should be referenced from the page `http://www.frama-c.cea.fr/plugins.html`. For this purpose, you have to edit files `plugins.prehtml` and `index.prehtml`.

In order to generate the html pages from directory `doc/www/src`, just execute

```
$ make html_pages
```

The generated website is available in directory `doc/www/export` and the homepage is `doc/www/export/index.html`.

The html pages belonging to directory `doc/www/src` must not be used in order to display the website because relative links are not the same than those of the real website. Use html pages of directory `doc/www/export` instead.

Recommendation 4.6 *You can use the address `http://validator.w3.org/#validate_by_upload` in order to check the validity of your* html *code. For this purpose, you can use `make all` instead of `make html_pages`.*

If you want to officially put the webpage on the Frama-C website, you have to contact CEA.

4.19 License Policy

Target readers: *developers with a CVS 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 stuffs are provide in `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 4.29 *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`.

Chapter 5

Reference Manual

This chapter is a reference manual for plug-in developers. it provides full details which complete Chapter 4.

5.1 File Tree

This Section introduces 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`). Directory containing Cil implementation is detailed in Section 5.1.1 while directory containing the Frama-C implementation itself is presented in Section 5.1.2.

Figure 5.1 shows directories useful for a plug-in developer. More details are provided below.

Kind	Name	Specification	Reference
	`.`	Frama-C root directory	
Sources	`src`	Frama-C implementation	Section 5.1.2
	`cil`	Cil source files	Section 5.1.1
	`ocamlgraph`	OcamlGraph source files	
	`external`	Source of external free libraries	
Tests	`tests`	Frama-C test suites	Section 4.5
	`ptests`	ptests implementation	
Generated Files	`bin`	Binaries	
	`lib`	Some compiled files	
Documentations	`doc`	Documentation directory	
	`headers`	Headers of source files	Section 4.19
	`licenses`	Licenses used by plug-ins and kernel	Section 4.19
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 and the .
- Directory `tests` contains the Frama-C test suite which is used by tool `ptests` (see Section 4.5).
- Directories `bin` and `lib` contains binary files mainly produced by Frama-C compilation. Frama-C executables belong to directory `bin`, directories `lib/plugins` and `lib/gui` receives the compiled plug-ins and directory `libfc` received the compiled kernel interface. You should never add yourself any file in these directories.
- Documentations (including plug-in specific, source code and ACSL documentations) are provided in directory `doc`. Directories `headers` and `licenses` contains files useful for copyright notification (see Section 4.19).
- Directory `share` contains useful libraries for Frama-C users such as the Frama-C C library (*e.g.* ad-hoc libraries `libc` and `malloc` for Frama-C) and user-oriented Makefiles.

5.1.1 Directory `cil`

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

Name	Specification
`ocamlutil`	OCaml useful utilities
`src`	Main Cil files
`src/ext`	Syntactic analysis provided by Cil
`src/frontc`	C frontend
`src/logic`	ACSL frontend

Figure 5.2: Cil directories.

- `ocamlutil` contains some OCaml utilities useful for a plug-in developer. Most important modules are `Inthash` and `Cilutil`. The first one contains an implementation of hashtables optimized for integer keys while the second one contains some useful functions (*e.g.* `out_some` which extract a value from an option type) and datastructures (*e.g.* module `StmtHashtbl` implements hashtables optimized for statement keys).
- `src` contains the main files of Cil. Most important modules are `Cil_types` and `Cil`. The first one contains type declarations of the Cil AST while the second one contains very 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 parameterised 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 Directory `src`

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 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 one-other for the implementation provided in `Db`.

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

Kind	Name	Specification	Reference
Toolboxes	`kernel`	Kernel toolbox	
	`logic`	Logic toolbox	
	`ai`	Abstract interpretation toolbox	Section 4.16
	`memory_states`	Memory-state toolbox	Section 4.16
Libraries	`project`	Project library	Section 4.11
	`lib`	Miscellaneous libraries	
	`misc`	Additional useful operations	
Entry points	`gui`	Graphical User Interface	Section 4.17

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 4.16). 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.
- Directory `project` is the project library fully described in Section 4.11.
- 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 4.8).
- Directory `gui`[1] contains the `gui` implementation part common to all plug-ins. See Section 4.17 for more details.

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

Kind	Name	Specification	Reference
AST	`Ast`	The Cil AST for Frama-C	
	`Ast_info`	Operations over the Cil AST	
Global tables	`File`	AST creation and access to C files	
	`Globals`	Operations on globals	
	`Kernel_function`	Operations on functions	
	`Annotations`	Operations on annotations	
	`Loop`	Operations on loops	
Database	`Db`	Static plug-in database	Section 4.10.1
	`Db_types`	Type declarations required by `Db`	Section 4.10.1
	`Dynamic`	Interface for dynamic plug-ins	Section 4.10.2
	`Kui`	High-level Frama-C front-end, quite deprecated	
Base Modules	`Config`	Information about Frama-C version	
	`Plugin`	General services for plug-ins	Section 4.6
	`Cmdline`	Command line parsing	Section 4.12
	`Log`	Operations for printing messages	Section 4.7
	`Parameters`	Kernel Parameters	Section 4.12
	`Journal`	Journalization	Section 4.9
	`CilE`	Useful Cil extensions	
	`Alarms`	Alarm management	
	`Kernel_type`	Type value for kernel types	Section 4.10.2
	`Stmts_graph`	Accessibility checks using CFG	
Visitor	`Visitor`	Frama-C visitors (subsume Cil ones)	Section 4.14
Pretty printers	`Ast_printer`	Pretty-printers for AST node	
	`Printer`	Main class for pretty-printing	
Initializer	`Boot`	Last linked module	Section 4.13
	`Gui_init`	Very early initialization of the GUI	Section 4.13
	`Special_hooks`	Registration of some kernel hooks	

Figure 5.4: Main kernel modules.

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 row 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 make	no	
2	Configuration of OCaml	no	
3	Configuration of mandatory tools/libraries	no	
4	Configuration of non-mandatory tools/libraries	no	
5	Platform configuration	no	
6	Wished Frama-C 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, right 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 readibility.

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 disable (at least partially) because they depend on others plug-ins which are not available (at least partially).

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 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_confic`: 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 plu-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 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.

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).

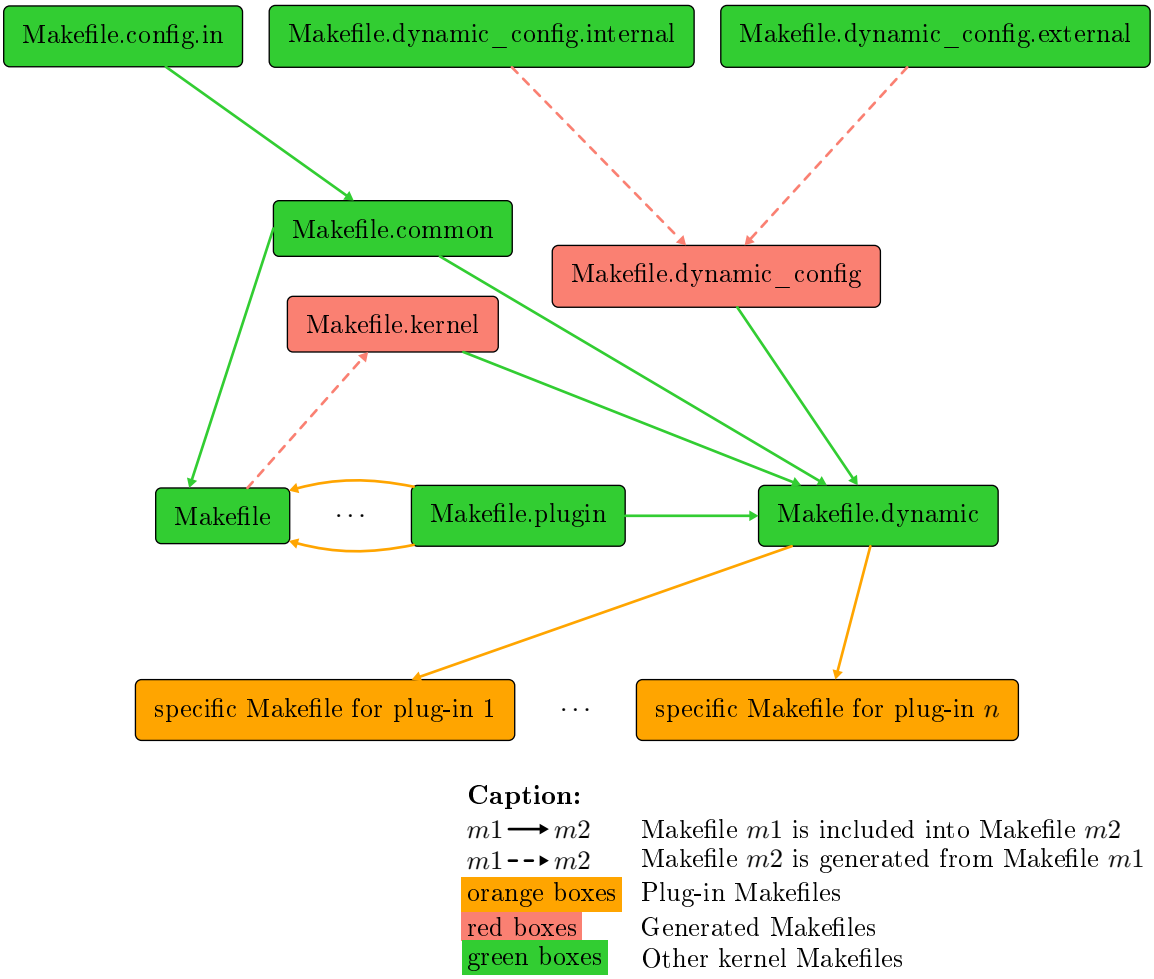

Figure 5.6: Relationship between the Makefiles

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 4.3
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**	***Not written yet.***
29	File headers: license policy	Makefile	**yes**	Section 4.19
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`.

2. **Installation paths** defines where Frama-C has to be installed.
3. **Ocaml stuff** defines the Objective Caml 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 right 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 templates for testing plug-ins.
13. **Generic rules** contains rules in order to automatically produces different kinds of files (*e.g.* `.cm[iox]` from `.ml` or `.mli` for Objective Caml 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 this library.
18. **Ocamlgraph** defines how compile this 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 (see Section 4.5).
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 4.18).
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 4.19).
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 4.5).
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 4.3 and 4.4). 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.

- Variable `PLUGIN_NAME` is the module name of the plug-in.

  So it must be capitalized (as each OCaml module name).

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
	PLUGIN_HAS_MLI	Whether the plug-in gets an interface	
Source files	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 `$(PLUGIN_DIR)`	
	PLUGIN_GUI_CMO	`.cmo` plug-in files not belonging to `$(PLUGIN_DIR)`	
Compilation flags	PLUGIN_BFLAGS	Plug-in specific flags for `ocamlc`	
	PLUGIN_OFLAGS	Plug-in specific flags for `ocamlopt`	
	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 `ocamlc`	
	PLUGIN_LINK_OFLAGS	Plug-in specific flags for linking with `ocamlopt`	
	PLUGIN_LINK_GUI_BFLAGS	Plug-in specific flags for linking a GUI with `ocamlc`	
	PLUGIN_LINK_GUI_OFLAGS	Plug-in specific flags for linking a GUI with `ocamlopt`	

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

- 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 4.18) 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 4.17). 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)`

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

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

- Variable `PLUGIN_NO_TEST` must be set to `yes` if there is no specific test directory for the plug-in.
- Variable `PLUGIN_TESTS_DIRS` is the directories containing plug-in tests. Its default value is `tests/$(notdir $(PLUGIN_DIR))`).
- Variable `PLUGIN_TESTS_LIB` is the `.cmo` plug-in specific files used by plug-in tests. Do not write its file path (which is `$(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_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 is set before including `Makefile.plugin` in order to customize its behavior. Nevertheless 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.*

5.4 Testing

Section 4.5 explains how to test a plug-in. Here Figure 5.11 details the options of `ptests` while Figure 5.12 shows all the directives that can be used in a configuration headers of a test (or a test suite). Some details about them are provided below.

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 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

[2] Variables 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	*plugin*_DIR
Source files	*plugin*_CMO *plugin*_CMI *plugin*_CMX *plugin*_TYPES_CMO *plugin*_TYPES_CMX
Compilation flags	*plugin*_BFLAGS *plugin*_OFLAGS *plugin*_LINK_BFLAGS *plugin*_LINK_OFLAGS *plugin*_LINK_GUI_BFLAGS *plugin*_LINK_GUI_OFLAGS
Dependencies	*plugin*_DEPFLAGS *plugin*_GENERATED
Documentation	*plugin*_DOCFLAGS *plugin*_TYPES_TODOC
Testing	*plugin*_TESTS_DIRS *plugin*_TESTS_LIB

Figure 5.10: Variables defined by `Makefile.plugin`.

`diff`. An exit code of `2` indicates an error during the comparison (for instance because of 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.

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

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

1. default value of the directive (as specified in Fig. 5.12);
2. inside file `tests/test_config`;
3. inside file `tests/`*subdir*`/test_config` (for each sub-directory *subdir* of `tests`); or
4. inside each test file

As presented in Section 4.5.3, alternative directives for test configuration <special_name> can be found in slightly different places:

- default value of the directive (as specified in Fig. 5.12);

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

Figure 5.11: `ptests` options.

- 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.ml`, each existing file in the sequence above is read in order and defines a configuration level (the default configuration level always exists).

- At a given configuration level, the default value for directive `CMD` is the last `CMD` directive of the preceding configuration level. Each directive `CMD` is used only with the next directive `OPT` or `STDOPT`. No test case is generated if there is no further `OPT` directive. Following `OPT` or `STDOPT` directives are applied on the default program until another directive `CMD` is given.
- 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 of the preceding configuration level. The syntax for this directive is the following.

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

Figure 5.12: Directives in configuration headers of test files.

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

options are always given between quotes. An option following a `+` is added to the current set of options while an option following a `-` is removed from it. The directive can be empty (meaning that the corresponding test will use the standard set of options). As with `OPT`, each `STDOPT` corresponds to a test case.

- 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 from preceding levels.

- `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.

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.

- **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**: updateed 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

We list changes of previous releases below.

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 specifities 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 writting 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 ocamldoc tag *@plugin developer guide*
- **Index**: fully new
- **Project**: new sub-section *Internal State: Principle*
- **Reference manual**: largely extended
- **Software architecture**: fully new chapter

Hydrogen-20080501

- **First public release**

Bibliography

[1] Patrick Baudin, Jean-Christophe Filliâtre, Thierry Hubert, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI C Specification Language*, April 2008.

[2] Sylvain Conchon and Jean-Christophe Filliâtre. Type-Safe Modular Hash-Consing. In *ACM SIGPLAN Workshop on ML*, Portland, Oregon, United States, September 2006.

[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, Armand Puccetti, and Julien Signoles. *Frama-C User Manual*, December 2009. http://frama-c.cea.fr/download/user-manual-Beryllium-20090902.pdf.

[5] Pascal Cuoq and Damien Doligez. Hashconsing in an incrementally garbage-collected system, a story of weak pointers and hashconsing in ocaml 3.10.2. In *ACM SIGPLAN Workshop on ML*, Victoria, British Columbia, Canada, September 2008.

[6] 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.

[7] Andrey P. Ershov. On programming of arithmetic operations. *Communication of the ACM*, 1(8):3–6, 1958.

[8] Free Software Foundation. *GNU 'make'*, April 2006. http://www.gnu.org/software/make/manual/make.html#Flavors.

[9] Jacques Garrigue, Benjamin Monate, Olivier Andrieu, and Jun Furuse. LablGTK2. http://wwwfun.kurims.kyoto-u.ac.jp/soft/olabl/lablgtk.html.

[10] Eiichi Goto. Monocopy and Associative Algorithms in Extended Lisp. Technical Report TR-74-03, University of Toyko, 1974.

[11] 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.

[12] Donald Michie. Memo functions: a language feature with "rote-learning" properties. Research Memorandum MIP-R-29, Department of Machine Intelligence & Perception, Edinburgh, 1967.

[13] Donald Michie. Memo functions and machine learning. *Nature*, 218:19–22, 1968.

[14] 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.

[15] Julien Signoles. Foncteurs impératifs et composés: la notion de projet dans Frama-C. In Hermann, editor, *JFLA 09, Actes des vingtièmes Journées Francophones des Langages Applicatifs*, volume 7.2 of *Studia Informatica Universalis*, pages 245–280, 2009. In French.

List of Figures

Index