Version Lithium-20081212
December 16, 2008

Plug-in Development Guide

Julien Signoles with Virgile Prevosto
CEA LIST, Software Reliability Lab.

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

Contents

Foreword

This is a preliminary documentation of the Frama-C implementation (available at http://www.frama-c.cea.fr) which aims to help any developer to integrate a new plug-in inside this platform. It is a deliverable of the task 2.3 of the ANR RNTL project CAT (http://www.rntl.org/projet/resume2005/cat.htm).

The content of this document corresponds to the version Lithium-20081212 (December 16, 2008) of Frama-C. However the development of Frama-C is still ongoing: several features described here may still evolve in the future.

Acknowledgements

We gratefully thank all the people who contributed to this document: Loïc Correnson for his complete reading with excellent suggestions in order to improve the document, Yannick Moy for his careful reading and great improvements of the document, especially the tutorial, and also Patrick Baudin, Pascal Cuoq, Benjamin Monate, Anne Pacalet and Richard Bonichon.

Chapter 1

Introduction

This guide aims at helping any developer to program within the Frama-C platfom, 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 4.12.1 for additional details about this documentation).

The reader of this guide may be either a Frama-C beginner who 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.

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.

Outline This guide is organised in four parts. The first one, 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 to both the Frama-C command line and other plug-in developments. The second part, Chapter 3, presents the design of the Frama-C software architecture. The third part, Chapter 4, details how to use all the services provided by Frama-C in order to develop a fully integrated plug-in. The fourth part, 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 this tutorial, this developer should be able to extend Frama-C with a simple analysis available to both the Frama-C command line and other plug-ins. This chapter was written to explain step-by-step how to proceed towards this goal. This tutorial explains how to integrate a plug-in inside the Frama-C platform. 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 everything you need to know to write a dynamic plug-in: writing a dynamic plug-in is the new preferred way to interface with Frama-C. Section 2.2 explains in detail how to write a static plug-in: 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 general purpose plug-in to the community.

2.1 Dynamic Plug-in

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

Dynamic plug-ins are a new important feature of Frama-C Lithium-20081212: the way Frama-C manages them may change in future release. However, if you follow the guideline given in this manual, it will be easy to switch from this implementation of dynamic plug-ins to a future one.

2.1.1 Setup

To follow this tutorial you have to fulfill the following requirements:

- 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];

[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.

- GNU make must be in your path.

2.1.2 Plug-in Integration Overview

Figure 2.2 shows how a dynamic plug-in can integrate with the Frama-C platform. This tutorial focuses on some parts only of this figure.

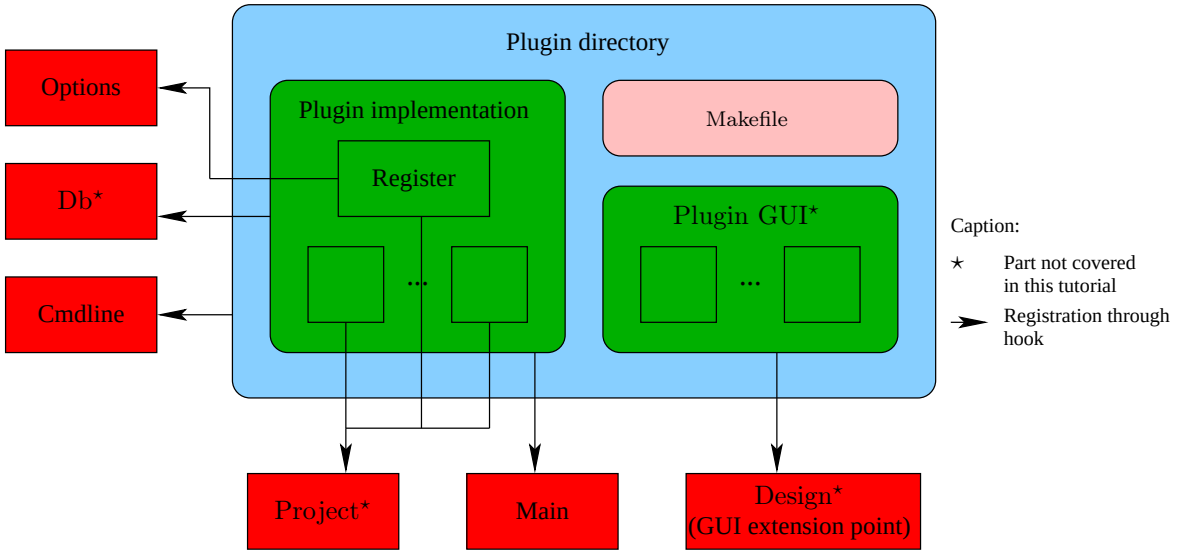

Figure 2.1: Dynamic plug-in Integration Overview.

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 defined in module `Main` if you want to run plug-in specific code when Frama-C is executed;
- add a new Frama-C option registered in module `Cmdline`;
- extend the Frama-C command line through module `Options` in order to add a new plug-in specific option.

2.1.3 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 Frama-C World!' whenever the option is set.

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

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

File *Makefile*

```
# Generic tiny Makefile for dynamic plug−ins
FRAMAC_SHARE=$(shell frama−c −print−path)
PLUGIN_NAME=hello
PLUGIN_SRC=hello_world.ml
include $(FRAMAC_SHARE)/Makefile.template
```

File *hello_world.ml*

```
(** The traditional Hello world! plug−in.
    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 error. *)

(** Register a new Frama-C option. *)
module Enabled =
  Cmdline.Dynamic.Register.False(struct let name = "hello enabled" end)

(** Print 'Hello Frama−C World!' whenever the option is set. *)
let startup fmt =
  if Enabled.get () then Format.fprintf fmt "Hello Frama−C World!"

(** Extend the Frama-C command line. *)
let () =
  Options.add_plugin
    ~name:"hello"                 (* plug−in name *)
    ~descr:"Hello World plugin"   (* plug−in description *)
    [ "−hello",                    (* plug−in option *)
      Arg.Unit Enabled.on,
      ": print \"Hello Frama−C World!\"" ]

(** Extend the Frama-C entry point (the "main" of Frama-C). *)
let () = Db.Main.extend startup
```

2.2 Static Plug-in

Target readers: *not for developers of dynamic plug-ins. It is only for:*

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

This section will teach you how to write the most basic static 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 static plug-ins. It is slightly more involved but allows a deeper integration within the Frama-C architecture.

2.2.1 Setup

If you have a CVS access to the Frama-C repository, you can download the sources for Frama-C with the CVS command below[2] where *login* is your CVS login and *cvsserver* is the name of the Frama-CCVS server name[3].

```
$ cvs -d :ext:login@servername/ppc/ppc co
```

Once you have the sources, you are ready for compilation. 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 -j
```

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] Character '$' (dollar) represents a shell prompt in all commands.
[3] You have to contact CEA in order to obtain the exact server name and a login.

2.2.2 Plug-in Integration Overview

Figure 2.2 shows how a plug-in should be integrated in the Frama-C platform. Most of the elements in this figure are pragmatically explained in the remaining sections of this tutorial.

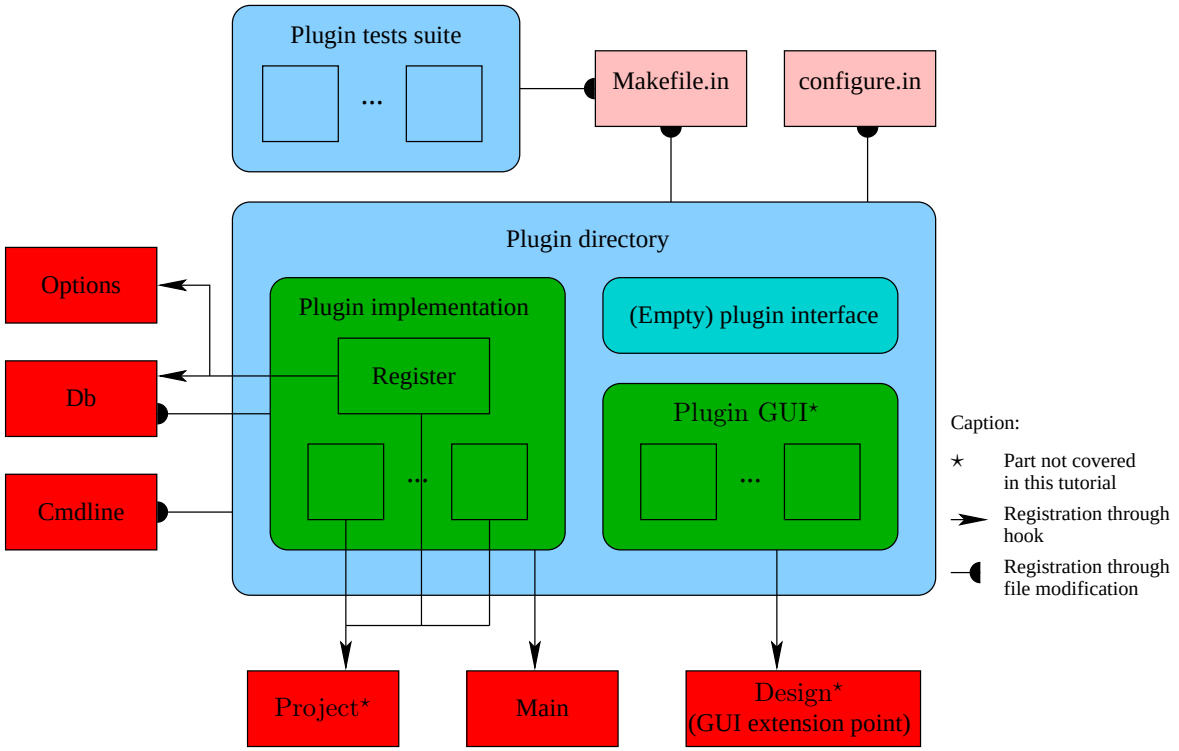

Figure 2.2: Static plug-in Integration Overview.

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 (function/functor applications or setting of references) or directly by modifying some specific part of Frama-C modules. For example, you have:

- to extend `Db` with your plug-in-specific operations and to register them inside it if you want that someone uses your plug-in (see Section 2.2.5);
- to extend the Frama-C entry point defined in module `Main` if you want to run plug-in specific code when Frama-C is executed (see Section 2.2.6);
- to add a sub-module inside module `Cmdline` and to apply a function defined in module `Options` if you want to add a new plug-in specific option to the Frama-C command line (see Section 2.2.6).

You also have to modify the files `Makefile.in` 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 specific test suites (see Section 2.2.7).

2.2.3 Hello Frama-C World

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

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[4]. 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`.

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`*.*

File *src/hello/register.ml*

```
let run fmt = Format.fprintf fmt "Hello Frama−C World!"
```

This file defines a function `run` which prints 'Hello Frama-C World!' on the given formatter.

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

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` 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`[5]. They indicate how to configure `hello`, especially whether it has to be compiled or not.

File *configure.in*

```
... /...
```

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

[5] 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.

```
··· /···
··· Add the following lines after other plug−in configurations.
# hello
#######
check_plugin(hello,src/hello,[support for hello plug−in],yes)
```

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 and the fourth one indicates whether the plug-in is available by default or not (here `yes` says that the plug-in is available by default and an user may use option `-disable-hello` to disactivate the plug-in).

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.in` with the following lines.

File *Makefile.in*

```
··· Add the following lines after other plug−ins compilation directives.
#########
# Hello #
#########
PLUGIN_ENABLE:=@ENABLE_HELLO@
PLUGIN_NAME:=Hello
PLUGIN_DIR:=src/hello
PLUGIN_CMO:= register
PLUGIN_NO_TEST:=yes
include Makefile.plugin
```

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 5.3.2, but most of them have reasonable default values so that it is not necessary to describe more than the few 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 capitalised).
- `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 2.2.7 about plug-in testing).

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

```
$ make -j
```

2.2.5 Connection with the Frama-C World

The plug-in `hello` is now compiled but it is not registered within the Frama-C framework. In particular, our plug-in should be added in the plug-in database `Db` in order to be 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 initialised 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 initialisation 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 capitalisation 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.in`, we add the following line before including `Makefile.plugin`.

File *Makefile.in*

```
... Setting others variables for hello
PLUGIN_HAS_MLI:=yes
... include Makefile.plugin
```

2.2.6 Extending the Command Line

In order to complete our plug-in, we have to register an option and to extend the command-line. Then, we have to make sure the function `!Db.Hello.run` is executed when this option is set. Section 4.8 provides more details about extensions of the command line.

First we add a value in the module `Cmdline` which indicates if the user has set the option `-hello` on the command line (*i.e.* whether we have to print the input files *via* the execution of `!Db.Hello.run` or not).

File *src/kernel/cmdline.mli*

```
...
(** {3 Hello} *)

module Hello: sig
  module Print: BOOL (** Whether to run hello or not. *)
end
...
```

File *src/kernel/cmdline.ml*

```
...
module Hello = struct
  module Print = False(struct let name = "Cmdline.Hello.Print" end)
end
...
```

`Cmdline` contains all the options of Frama-C and its static plug-ins. The above lines of code add a module `Hello` with all the options for `hello`. In fact, the plug-in `Hello` has only one option, called `print`. In Frama-C, each such option is a module. The signature of the module indicates the type of the option: in this case, it is a boolean option (whether to print the input files of Frama-C or not). In order to implement this option, we use a functor, called `False` and defined in top of `Cmdline`, which initialises it to `false` (*i.e.* the option is unset by default).

Once we have introduced this value, we can add the option `-hello` to the toplevel command line by extending the plug-in `hello`.

File *src/hello/register.ml*

```
...
let () =
  Options.add_plugin
    ~name:"hello"                  (* plug-in name *)
    ~descr:"Hello World plugin"    (* plug-in description *)
    [ "-hello",                    (* plug-in option *)
      Arg.Unit Cmdline.Hello.Print.on,
      ": print \"Hello Frama-C World!\"" ]
```

We call `Options.add_plugin`. This function integrates the new option `-hello` and modifies the

value contained in `Cmdline.Hello.Print` as well. This function also adds information about the plug-in `hello` when the predefined option `-help` is set by the user.

Finally we extend the "main" of Frama-C (*i.e.* its entry point) in order to execute the new plug-in whenever its option is set. For this purpose, we augment file `src/hello/register.ml` in the following way.

File *src/hello/register.ml*

```
let startup fmt = if Cmdline.Hello.Print.get () then !Db.Hello.run fmt;
let () = Db.Main.extend startup
```

At this point, the plug-in works properly: all the programming work is done and a Frama-C user can run the plug-in safely.

```
$ frama-c -hello foo.c bar.c baz.c
Parsing
Cleaning unused parts
Symbolic link
Starting semantical analysis
Hello Frama-C World!
```

2.2.7 Testing

Frama-C provides a tool, called ptests, in order to perform non-regression and unit tests. This tool is detailed in Section 4.4. This section only covers basic use of ptests. First we have to create a test directory for `hello`

```
$ mkdir tests/hello
```

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

File *Makefile.in*

```
# ··· Place of variables of plug−in hello
# PLUGIN_NO_TEST:=yes    # unset this variable
```

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
Parsing
Cleaning unused parts
Symbolic link
Starting semantical analysis
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`. It is also possible to add plug-in `hello` to the default test suite by editing the value of the variable `default_suites` in the file `ptests/config.ml`.

Note to CVS users If you have write access to the CVS repository, you can commit your changes into the archive. Before that, you have to perform non-regression tests in order to ensure that the modification does not break the archive.

So you must execute the following commands.

```
$ cvs add ...   # Do not forget new oracles
$ cvs up
$ make tests
$ cvs commit -m "informative message"
```

If you created any new files, use the `cvs add` command to add them into the archive. The `cvs 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,* you can commit your changes thanks with the `cvs commit` command.

2.2.8 Copyright your Work

Target readers: *developers with a CVS access.*

If you want to redistribute plug-in `hello`, you have to choose a license policy for it (compatible with Frama-C). Section 4.13 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.in` with the following line:

File *Makefile.in*

```
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.

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 [11] seen by Frama-C;
- Section 3.3 shows the organisation of the Frama-C kernel;
- and Section 4.5 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) [11] extended with an implementation of the specification language ACSL (ANSI/ISO C Specification Language) [1]. That is the intermediate language upon which Frama-C is based. See Section 3.2 for details.
- The Frama-C kernel. That 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). See Section 3.3 for details.
- Frama-C plug-ins. That is analyses or source-to-source transformations which use the kernel and possibly others plug-ins through the plug-ins database called `Db` (and the interface for dynamic plug-ins called `Dynamic`). See Section 4.5 for details.

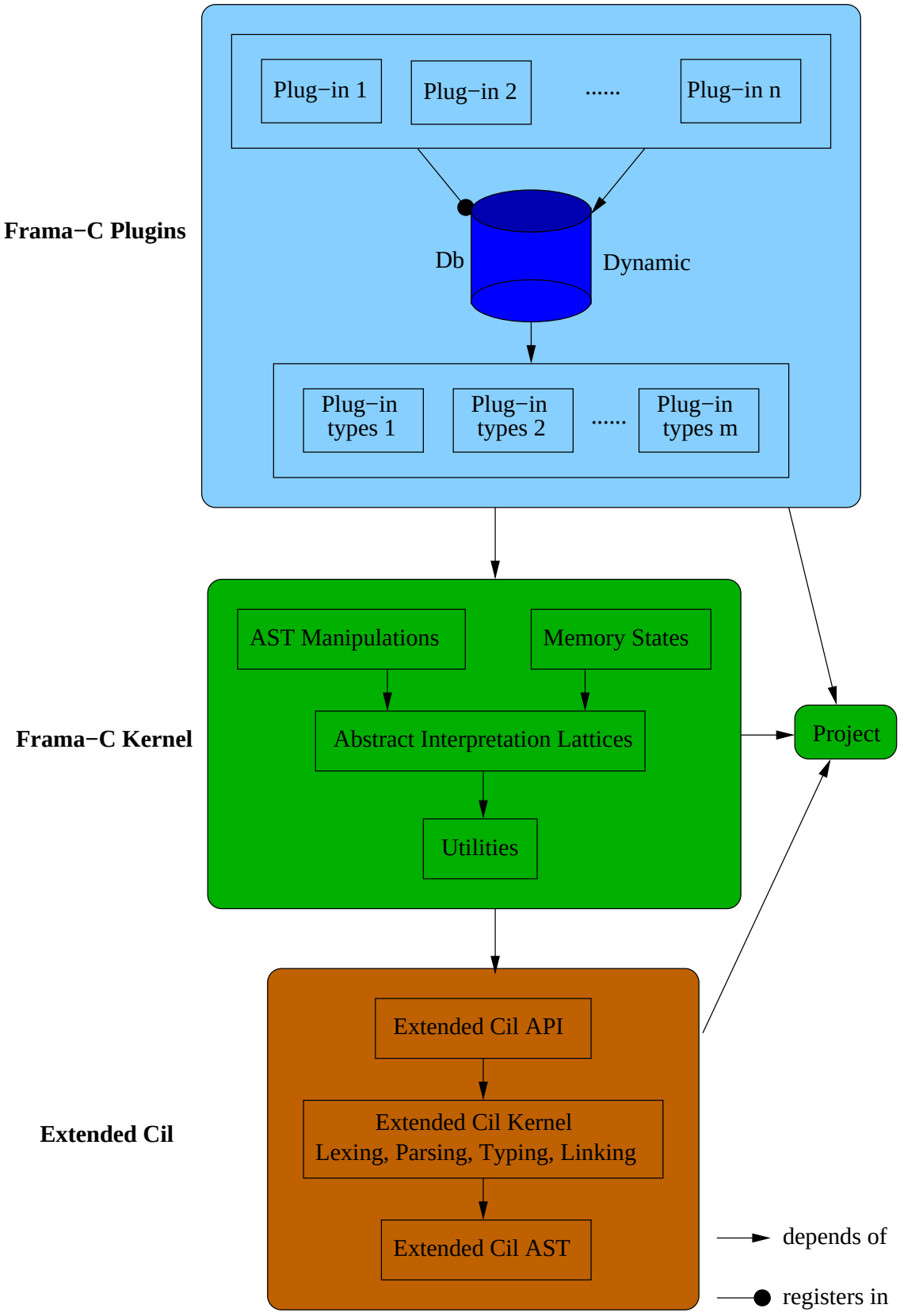

Figure 3.1: Architecture Design.

3.2 Cil: C Intermediate Language

Cil [11] 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 different kinds of modules which are described below.

- In addition to the Cil utilities, Frama-C provides useful operations (mainly in module `Extlib`) and datastructures (e.g. specialised 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, 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. This library is quite independent of Frama-C and may be used anywhere, exactly as an external library.

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 may be used by another plug-in, it has to be registered. There are two different ways to register a plug-in and to access to it, depending on whether the plug-in is static or dynamic. Static plug-ins have to register them by editing module `Db` while dynamic plug-ins have to register them through module `Dynamic`. Similarly, using a function of a static plug-in requires to call a function of module `Db` while using a function of a dynamic plug-in requires to call a function of module `Dynamic`.

Besides, static plug-ins may define their own datatypes (eventually coming with some specific operations) which can be visible from outside (usually because the plug-in API uses them). In order to keep as small as possible the plug-ins database `Db`, these datatypes are put outside of it. For visibility purpose, they are also put outside their plug-in owners. That is the *raison d'être* of plug-ins types. See Section 4.5.1 for additional details.

Even if the *raison d'être* of a dynamic plug-in is to be dynamically loaded by Frama-C, it is still possible to statically linked it with Frama-C and to use it as a static plug-in. For instance, exporting datatypes is possible for a dynamic plug-in statically linked with Frama-C.

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 4.6.3);
3. introducing bugs in your code;
4. introducing bugs in other plug-ins using you 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 4.6.2).

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 autoconf, make and advanced features of OCaml (module system, classes and objects, *etc*). 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 independent from one another. 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 [11] extended with an ACSL [1] implementation. The second one is the core implementation of Frama-C. This last directory contains directories of the Frama-C kernel and directories of the provided Frama-C plug-in.

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

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

4.2 Configure.in

Target readers: *not for dynamic 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 4.2.3) and with other plug-ins (Section 4.2.4). Finally Section 4.2.5 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 specialised 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, there are three actions to perform:

1. add a new subsection for the new plug-in to Section *Plugin wished*;

2. add a new substitution in Section *Substitutions to perform*;
3. add a new entry in Section *Summary*.

All these actions are 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.

Let us illustrate how these actions are done by an example: say we want want to define a plug-in called `occurrence`.

First, Section *Wished Plug-in* introduces a new sub-section for this plug-in in the following way.

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

The first argument is the plug-in name, the second one is the name of directory containing the source files of the plug-in, the third one is a help message for the `-enable-occurence` option of configure and the last one indicates if the plug-in is enabled by default.

The macro `check_plugin` sets the following variables: `FORCE_OCCURRENCE`, `REQUIRE_OCCURRENCE`, `USE_OCCURRENCE` and `ENABLE_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 4.2.4). Finally `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`).

Section *Substitutions to perform* adds respectively a new substitution in `Makefile.in` thanks to the line:

```
AC_SUBST(ENABLE_OCCURRENCE)
```

Similarly Section *Summary* adds a new entry in the summary thanks to the line:

```
AC_MSG_NOTICE([occurrence            : $ENABLE_OCCURRENCE$INFO_OCCURRENCE])
```

The value `@ENABLE_OCCURRENCE@` is then usable in `Makefile.in` in order to know whether the plug-in has to be compiled or not (see Section 4.3) and a notification indicating this value as well as an optional informative message (contained in `$INFO_OCCURRENCE`) is displayed to the user.

4.2.3 Addition of Library/Tool Dependencies

Three different variables are set for each external library and tool used in Frama-C which are

- HAS_*library*
- REQUIRE_*library*
- USE_*library*

where *library* is the name of the considered library or tool (see Section 4.2.5 for explanations about their initialisations and their uses).

HAS_*library* indicates whether the library is available on this platform (its value is `yes`) or not (its value is `no`). This last value is accessible in `Makefile.in` through the variable `@HAS_`*library*`@` (see Section 4.3). Actually we are not concerned by this value in this section.

REQUIRE_*library* (resp. USE_*library*) is a list of plug-in names (separated by spaces). It contains the plug-ins for which *library* is a mandatory (resp. an optional) dependency. So you have to extend these lists in order to add some library/tool dependencies for a new plug-in *p*.

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 4.1 *Plug-in* ***gui*** *requires* Lablgtk2 *[7]. So, just after its declaration, there are the following lines in* ***configure.in****.*

```
if test "$ENABLE_GUI" == "yes"; then
  REQUIRE_LABLGTK=${REQUIRE_LABLGTK}" gui"
fi
```

These lines specify that Lablgtk2 *must be available on the system if the user wants to compile* ***gui****.*

4.2.4 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 4.2.3). For this purpose, `configure.in` uses variables REQUIRE_*plugin* and USE_*plugin* (in the same way that variables REQUIRE_*library* and USE_*library*: they are lists of plug-in names for which *plugin* is respectively a mandatory dependency or an optional dependency).

From the viewpoint of a plug-in developer , the difference between libraries and tools is that the best place to indicate such dependencies is not just after the addition of the plug-in: needed variables REQUIRE_*plugin* and USE_*plugin* could be undeclared at this point (in particular in the case of mutually dependent plug-ins). So dependency indications are postponed at the top of Section *Plug-in dependencies* of `configure.in`.

Example 4.2 *Plug-In* ***value*** *requires plug-in* ***from*** *and may use plug-in* ***gui*** *(for* ValViewer *[2]). So lists* ***REQUIRE_FROM*** *and* ***USE_GUI*** *contain* ***value****. Moreover, as many plug-ins require* ***value****, list* ***REQUIRE_VALUE*** *is quite big. In particular, it contains plug-in* ***from****: both plug-ins* ***value*** *and* ***from*** *are indeed mutually dependent.*

4.2.5 Configuration of New Libraries or Tools

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.

Section 4.2.3 explains how to depend on some external library *library*. Nevertheless if *library* is not used by Frama-C anywhere else, you have to configure it.

First, you have to declare the three variables set by each library: `HAS_`*library*, `USE_`*library* and `REQUIRE_`*library*. This is performed in Section *Configuration of Plug-in Libraries* of file `configure.in`. You should not assign values to these variables (just declare them).

Next, you have to export `HAS_`*library* in `Makefile.in` through `AC_SUBST(HAS_`*library*`)` in Section *Makefile Creation* of `configure.in`.

Last but not least, you have to check that the library is available on the user system. A predefined macro called `configure_library` helps the plug-in developer in this task[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 and put in the variable `SELECTED_$library$`. 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 and, according to it, disables (resp. partially disables) the plug-ins requiring (resp. optionally using) it[3].

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 4.3 *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:*

[2] For tools, there is a macro `configure_tool` which works in the same way as `configure_library`.
[3] As plug-in dependencies are checked after this check, plug-ins are not recursively disabled here.

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

4.3 Makefile.in

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

Prerequisite: *knowledge of* make.

In this section, we detail the use of `Makefile.in` dedicated to Frama-C compilation. This file is split in several sections which are described in Section 5.3.1. 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 5.3.2.

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

Example 4.4 *For compiling the plug-in `Value`, the following lines are added into `Makefile.in`.*

[4] Using `:=` only sets the variable value from the affectation point (as usual in most programming langages) 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 [6]).

```
##################
# 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
```

As said above, you cannot use the parameters of `Makefile.plugin` anywhere in `Makefile.in`. You can yet use some plugin-in specific variables once `Makefile.plugin` has been included. These variables are detailed in Section 5.3.2.

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.in` or `Makefile.plugin`.

4.4 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 4.4.1 shows how to use ptests. Next Section 4.4.2 explains how to configure tests through directives. Last Section 4.4.3 describes how to set up various testing goals for the same test base.

4.4.1 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 4.5 *If you want to test plug-in* `sparecode` *and specific test* `tests/pdg/variadic.c`*, just run*

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

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

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

ptests accepts different options which are used in order to customize one test sequence. These options are detailed in Section 5.4.

Example 4.6 *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.

4.4.2 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 4.7 *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
```

4.4.3 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
```

4.5 Plug-in Registration and Access

In this section, we present how to register plug-ins and how to access to them. Actually there are two different ways to register plug-ins depending on whether they are static or dynamic (cf Section).

Section 4.5.1 indicates how to register and access to a *static* plug-in while Section 4.5.2 details how to register and access to a *dynamic* plug-in.

4.5.1 Static Registration and Access

Target readers: *static plug-ins developers.*

A database, called `Db` (in directory `src/kernel`), groups together all static plug-ins. It also provides their API which permit easy plug-in collaborations. Each static 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 4.8 *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 static 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.

The solution is to put these datatype declarations in 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.in` by extending variable `UNPACKED_DIRS` (see Section 5.3.2).

Example 4.9 *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.in*

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

A bad side effect of this design choice is that export types are not hidden. If you want to hide them, you have to encapsulate them in modules providing required getters and setters. So you have now plug-in code outside plug-in implementation which should be linked before `Db`[5]. Files containing this code has to be known by the makefile: set make variable `PLUGIN_TYPES_CMO` for this purpose (see Section 5.3.2).

4.5.2 Dynamic Registration and Access

Target readers: *dynamic plug-ins developers.*

Registration of static plug-ins requires to modify module `Db` which belongs to the Frama-C kernel. Dynamic plug-ins are fully independent of Frama-C and so cannot modified its kernel. Consequently, another way is provided for registering a plug-in. It uses module `Dynamic`.

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

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

In this way, you can only register *values*, not *types*. Exporting your own types like explained in Section 4.5.1 is not possible with dynamic plug-ins: you must only use types provided by the Frama-C kernel. For the implementation of complex protocols involving sharing of custom data between plug-ins, the use of ACSL annotations [1] is the prefered way to achieve your goal.

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

```
val register: string → 'a Type.t → 'a → unit
```

The first argument has to be the binding name of the registered OCaml value. It should not be used for value registration anywhere else in the Frama-C world. It is required for applying the value in the case of a function (see paragraph below). The second argument is the so-called *type value* of the registered value, *i.e.* an OCaml value representing its type. It is required for safety reasons when applying the registered value in the case of a function (see paragraph below). 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 4.10 *Here is how function `load` of the dynamic plug-in `Journal_loader` is registered. This function is of type `string →unit`*

File *src/journal_loader/register.ml*

```
let load = ···
let () =
  Dynamic.register
    "Journal_loader.load"
    (Type.func Type.string Type.unit)
    load
```

If the string `"Journal_loader.load"` is already used to register a dynamic value, then the exception `FunTbl.AlreadyExists` is raised at plugin initialisation time (see Section 4.7).

The function call `Type.func Type.string Type.unit` returns the type value of `string →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 value `load`) has not the type `string →unit`.

Calling a previously-registered function The signature of function `Dynamic.apply` is as follows.

```
val apply: string → 'a Type.t → 'a
```

The first argument has to be the binding name of the OCaml value registered with `Dynamic.register`. The second argument is the type value of this registered value. It is required for safety reasons. The third argument is the value itself.

Example 4.11 *Here is how the previously registered function `load` of `Journal_loader` may be apply to the string `"frama_c_journal.ml"`.*

File *src/journal_loader/register.ml*

```
let () =
  Dynamic.apply
    "Journal_loader.load"
    (Type.func Type.string Type.unit)
    "frama_c_journal.ml"
```

The given string and the given type value have to be the same than the ones used when registering the function. Otherwise, the exceptions `FunTbl.Not_Registered` and `FunTbl.Incompatible_Type` are respectively raised. Furthermore, because of the type of `Dynamic.apply` and the types of its arguments, the OCaml type checker complains either if the third argument (here `"frama_c_journal.ml"`) is not of type `string` or if the returned value (here `()`) is not of type `unit`.

4.6 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*. Section 4.6.1 first introduces the general principle of project. Then Section 4.6.2 explains how to simply use them. Section 4.6.3 introduces the so-called *internal states* for which registration is detailed in Sections 4.6.4, 4.6.5 and 4.6.6. Section 4.6.4 is dedicated to so-called *datatypes*. Section 4.6.5 is dedicated to the internal states themselves. Section 4.6.6 is dedicated to low-level registration. Finally Section 4.6.7 shows how to handle projects and internal states in a clever and proper way.

4.6.1 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 4.8) and tables containing results of analyses (Frama-C extensively uses memoisation [9, 10] in order to avoid re-computation of 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 resumed 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.

4.6.2 Using Projects

Actually Frama-C maintains a current project (`Project.current ()`) and a current AST (`Cil_state.file ()`) which all operations are automatically performed on. But sometimes a plug-in developer have to explicitly use them, for example when the AST is modified (usually through the use of a copy visitor, see Section 4.10) 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 create a new project with a new AST, usually by using `File.init_project_from_cil_file` or `File.init_project_from_visitor`.

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

Example 4.12 *Suppose that you saved the current project into file* `foo.sav` *in a previous* Frama-C *session*[6] *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
                                  .../...
```

[6] A *session* is one execution of Frama-C (through `toplevel.[byte|opt]` or `viewer.[byte|opt]`).

```
                                    .../...
  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 4.13 *The following code is equivalent to the one given in Example 4.12.*

```
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
```

It displays

```
false
true
false
```

4.6.3 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 memoisation).

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 4.6.7). 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 4.14 *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 `Kernel_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 `Kernel_computation` does the same for standard Frama-C datastructures (like hashtables indexed by AST statements)[7].

Registering a new internal state must be performed before the last initialisation step which is the run of each function registered through argument `toplevel_init` of `Options.add_plugin` (see Section 4.7).

Section 4.6.5 details how to register a new computation.

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

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. `Kernel_datatype`) provides pre-defined datatypes and datatypes-builder for elaborated types[8]. Section 4.6.4 details how to register a new datatype.

Example 4.15 *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)
```

4.6.4 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[9] 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 4.16 *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
     let copy x = { a = x.a }
     let name = "a"
   end)
```

[8] 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).

[9] Hash-consing is a programming technique saving memory blocks and speeds up operations on datastructures when sharing is maximal [5, 8, 3, 4].

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 `Kernel_datatype` (*e.g.* `Kernel_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 `Kernel_datatype` provided all the available modules.

Example 4.17 *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:*

```
Kernel_datatype.VarinfoHashtbl(Datatype.List(Kernel_datatype.KernelFunction))
```

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 and to rehash it (usually just rebuild the structure by applying the right copy and rehash functions on subterms).
- A name for the datatype.

Example 4.18 *The type of postdominators is the following variant.*

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

The corresponding registed 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 Kernel_datatype.StmtSet.copy
     let rehash = map Kernel_datatype.StmtSet.rehash
     let name = "postdominator"
   end)
```

4.6.5 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 4.6.6 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 memoise 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 memoise s =
  try StmtHashtbl.find state s
  with Not_found → StmtHashtbl.add state s (compute_info s)
let run () = ... !Db.Value.compute (); ... memoise 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 =
  Kernel_computation.StmtHashtbl
    (Datatype.Couple(Kernel_datatype.KernelFunction)(Kernel_datatype.Varinfo))
    (struct
      let size = 97
      let name = "state"
      let dependencies = [ Db.Value.self ]
     end)
let compute_info = ···
let memoise = State.memo compute_info
let run () = ... !Db.Value.compute (); ... memoise 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 `Kernel_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 4.6.4). 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 memoise the computation of `compute_info`. This memoisation 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 4.5.1). 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 initialised at the plug-in initialisation time (see Section 4.7). 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 initialisation is not specified (see Section 4.7 for more details about initialisation steps). So you have to postpone the addition of this dependency; usually by using function `Options.register_plugin_init` (see Section 4.8).

Example 4.19 *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 = [ Value.self ]
      end)
let () = Db.From.self := Functionwise_Dependencies.self
                (* performed at module initialisation runtime. *)
```

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 () =
  Options.register_plugin_init
    (fun () → Project.Computation.add_dependency Tbl.self !Db.From.self)
```

For dynamic plug-ins, it is possible to register state kinds in the same way that any other value through `Dynamic.register` (see Section 4.5.2).

4.6.6 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 4.6.5).

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 four functions `create` (creation of a new fresh state), `clear` (cleaning a state), `get` (getting a state) and `set` (setting a state).

The following invariants must hold:[10]

`create ()` returns a fresh value (4.1)

∀`p` of type `t`, `create () = (clear p; set p; get ())` (4.2)

∀`p` of type `t`, `copy p` returns a fresh value (4.3)

∀`p1`, `p2` of type `t`such that `p1 != p2`, `(set p1; get ()) != p2` (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 4.20 *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 ())
```

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
    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
```

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

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.*

4.6.7 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 4.21 *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 4.22 *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 4.8).*

```
Project.on ~only:(Cmdline.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.

4.7 Initialisation Steps

Prerequisite: *knowledge of linking of* OCaml *files and* OCaml *labels.*

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

As side effects are sometimes useful, Frama-C provides some ways to put it at different initialisation times. For this purpose, function `Options.register_plugin_init` allows to register a function executed before parsing the Frama-C command line (see Section 4.6.5) while function `Options.add_plugin` has three optional arguments `plugin_init`, `init` and `toplevel_init` usable in order to control Frama-C initialisation (see Section 4.8). Actually, the whole Frama-C initialisation process is enclosed in module `Boot` (the last linked module) which is the main entry point of Frama-C.

In order to clear what is done when Frama-C is booting, we better specify the Frama-C initialisation order below.

1. Running each Frama-C compilation unit in a mostly unspecified order. The only assumption is that the link order respects the below partial order:
   (a) external libraries
   (b) project files (in `src/project`)
   (c) cil files (in `cil/src` and sub-directories)
   (d) kernel files
   (e) non-gui plug-in files
   (f) gui non plug-in files (in `src/gui`)[11]
   (g) gui plug-in files[11]
   (h) `src/kernel/boot.ml`;
2. Running each function registered through `Options.register_plugin_init` (in an unspecified order). Usually these functions initialise postponed internal-state dependencies (see Section 4.6.5).
3. Running each function registered through argument `plugin_init` of `Options.add_plugin` (in an unspecified order). Usually these functions are used for plug-in initialisations.
4. Parsing the Frama-C command line.
5. Running each function registered through argument `init` of `Options.add_plugin` (in an unspecified order). Usually these functions are used for initialisations depending on command line options.
6. Initialising a bunch of Cil attributes.
7. Running each function registered through argument `toplevel_init` of `Options.add_plugin`. Usually these functions are used in order to launch the right Frama-C entry point (*e.g.* usually defined in `Main` for a non-graphical Frama-C application).

[11] If the graphical user interface is compiled.

4.8 Command Line Options

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

Values associated with command line options are stored in module `Cmdline` while command line options themselves are registered through module `Options`. Section 4.8.1 and 4.8.2 introduces how to store new option values for static and dynamic plug-ins, respectively. Finally, Section 4.8.3 presents how to register new options.

4.8.1 Storing New Static Option Values

In Frama-C, an option value is actually a structure implementing signature `Cmdline.S` in order to handle projects: each option value is indeed an internal state (see Section 4.6.5). This structure should be stored in module `Cmdline`. Actually a bunch of signatures extended `Cmdline.S` are provided in order to deal with the usual option types. For example, there are signatures `Cmdline.INT` and `Cmdline.BOOL` for integer and boolean options. Mostly, these signatures provide getters and setters for options.

Implementing such an interface is very easy thanks to internal functors provided in module `Cmdline`. 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 body of `Cmdline` for the complete list).

1. `False` (resp. `True`) builds a boolean option initialised to `false` (resp. `true`).
2. `Int` (resp. `Zero`) builds an integer option initialised to a specified value (resp. to `0`).
3. `String` (resp. `EmptyString`) builds a string option initialised 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 (at least) a name as argument which corresponds to the name of the internal states for this option (see Section 4.6.5).

Example 4.23 *Value for option* `-slevel` *is the module* `SemanticUnrollingLevel` *of* `Cmdline` *and is implemented as follow.*

```
module SemanticUnrollingLevel =
  Zero(struct let name = "Cmdline.SemanticUnrollingLevel" end)
```

So it is an integer option initialised by default to `0`*. Interface for this module is simply*

```
module SemanticUnrollingLevel: INT
```

Value for option `-general-font` *(viewer only) is the module* `GeneralFontName` *and is implemented as follow.*

```
module GeneralFontName =
  String
    (struct
       let default = "Helvetica 10"
       let name = "Cmdline.GeneralFontName"
     end)
```

So it is a string option initialised by default to `Helvetica 10`. Interface for this module is simply

```
module GeneralFontName: STRING
```

Recommendation 4.3 *Options of a same plug-in `plugin` should belong to a same module `PluginOptions` inside `Cmdline`.*

4.8.2 Storing New Dynamic Option Values

Target readers: *dynamic plug-in developpers.*

As a dynamic plug-in is not allowed to modify the Frama-C kernel, it cannot register its own options in module `Cmdline`.

In order to solve this issue, options of a dynamic plug-in have to be stored in the plug-in itself. For this purpose, module `Cmdline` provides a dedicated sub-module `Cmdline.Dynamic.Register`. This sub-module defines a subset of the bunch of functors than the internal functors of `Cmdline` (see Section 4.8.1 for details about these functors). These modules share the same signature than their internal counterparts. For instance, there are modules `Cmdline.Dynamic.Register.False` and `Cmdline.Dynamic.Register.Zero` which respectively correspond to the internal modules `False` and `Zero` of `Cmdline`. Both modules `Cmdline.Dynamic.Register.False` and `Cmdline.False` share the same signature `Cmdline.BOOL` while both modules `Cmdline.Dynamic.Register.Zero` and `Cmdline.Zero` share the same signature `Cmdline.INT`.

Example 4.24 *Value for the option `-load-journal` is the module `LoadFile` defined inside the dynamic plug-in `Journal_loader` as follows.*

File *src/journal_loader/register.ml*

```
module LoadFile = Cmdline.Dynamic.Register.EmptyString
  (struct let name = "Journal_loader.load" end)
```

Inside the plug-in which defines a dynamic option, you can use this option like any other. But, although the module defining the option is not visible from the outside of its plug-in, the option is accessible by any other plug-ins (and even by the Frama-C kernel as well) through module `Cmdline.Dynamic.Apply`. Functions of sub-modules of module `Cmdline.Dynamic.Apply` takes

a string in argument which is the name associated with the wished option. For instance, that is the string `"Journal_loader.load"` for the value corresponding to the option `-load-journal` (see previous example).

Example 4.25 *Here we shows how module `Options` accesses to the option stored in the module `LoadFile`, dynamically registered in the previous example, in order to load a* Frama-C *journal if required. For clarity, error handling is not shown in this example.*

File *src/toplevel/options.ml*

```
...
let option_name = "Journal_loader.load" in
  (* option_name is exactly the same name used to register the option. *)
(* If the option is set *)
if Cmdline.Dynamic.Apply.String.is_set option_name then
  (* Get the name stored in the option *)
  let filename = Cmdline.Dynamic.Apply.String.get option_name in
  (* Load the journal corresponding to this file name.
     This action is done by a dynamically−registered function.
     See Section 4.5.2. *)
  Dynamic.apply
    "Journal_loader.load"
   (Type.func Type.string Type.unit)
   filename
...
```

4.8.3 Registering New Options

You have to use function `Options.add_plugin` for registering all options of a plug-in. For example, this function automatically displays help messages on the command line in the Frama-C standard form. Moreover it takes optional arguments which allow to customize the plug-in initialisation process (see Section 4.7). See documentation attached to it in file `src/toplevel/options.mli` for more details.

Usually function `Options.add_plugin` is called at module initialisation time: so options are registered when the Frama-C command line is parsed (see Section 4.7).

Example 4.26 *For illustrating the use of this function, we show how two plug-ins use it. First consider plug-in `users` (see file `src/users/users_register.ml`).*

```
let call_for_users = ···
let init () =
  if Cmdline.ForceUsers.get () then
    Db.Value.Call_Value_Callbacks.extend call_for_users

let () =
  Options.add_plugin
```

···/···

```
                                    .../...
    ~name:"users" ~descr:"users of functions" ~init
    [ "-users", Arg.Unit Cmdline.ForceUsers.on,
      ": compute users (through value analysis)"; ]
```

The call to `Options.add_plugin` *adds a single option* `-users` *which sets the value* `Cmdline.ForceUsers` *when it is set. Arguments* `name` *and* `descr` *are used by option* `-help` *of* Frama-C. *Argument* `init` *is performed right after the parsing of the command line (see Section 4.7) and here extends the value analysis in order to execute the users analysis when this is required by the user.*

The second example is plug-in `pdg` *(see file* `src/pdg/register.ml`*).*

```
let () =
  Options.add_plugin ~name:"Program Dependence Graph (experimental)"
    ~descr:""
    ~shortname: "pdg"
    ~debug:[
      "-verbose", Arg.Unit Cmdline.Pdg.Verbosity.incr,
      ": increase verbosity level for the pdg plug-in (can be repeated).";

      "-pdg",
      Arg.Unit Cmdline.Pdg.BuildAll.on,
      ": build the dependence graph of each function for the slicing tool";

      "-fct-pdg",
      Arg.String Cmdline.Pdg.BuildFct.add,
      "f : build the dependence graph for the specified function f";

      "-dot-pdg",
      Arg.String Cmdline.Pdg.DotBasename.set,
      "basename : put the PDG of function f in basename.f.dot";

      "-dot-postdom",
      Arg.String Cmdline.Pdg.DotPostdomBasename.set,
      "basename : put the postdominators of function f in basename.f.dot";
    ]
    [ ]
```

This code adds some debugging options for plug-in `pdg`*. This option are usable right after* `-pdg-debug` *option which is specified thanks to argument* `shortname`*. Actually there is no true option for this plug-in: all options are debugging ones.*

4.9 Locations

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

In Frama-C, different representations of C locations exist. Section 4.9.1 presents them. Moreover, maps indexed by locations are also provided. Section 4.9.2 introduces them.

4.9.1 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).

4.9.2 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`).

4.10 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 4.10.6, which the reader already familiar with Cil is invited to read carefully.

4.10.1 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.

4.10.2 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*[12], and has type *astType*→*astType'* `visitAction`, where *astType'* is either *astType* or *astType* `list` (for instance, one can transform a `global` into several ones). `visitAction` describes what should be done for the children of the resulting AST node, and is presented in the next section. In addition, 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, three methods, `vstmt`, `vfile`, 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.

4.10.3 Action Performed

The return value of visiting methods indicates what should be done next. There are four 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;
- `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 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`

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

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.

4.10.4 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.

4.10.5 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_info`, `predicate_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.

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`).

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

Figure 4.2: Indices of AST nodes.

4.10.6 Differences Between the Cil and Frama-C Visitors

As said in Section 4.10.2, `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 4.10.3), 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.

4.10.7 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_const.expr_to_term 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 =
  let prj = Project.create "syntactic check" in
  File.init_project_from_visitor prj (new Syntactic_check.non_zero_divisor);
  prj
```

4.11 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`. `Design.main_window_extension_points` is a class type properly documented providing access to the main widgets of a Frama-C gui.

Such a code has to be put in separate files into the plug-in directory. Moreover, in `Makefile.in`, variable `PLUGIN_GUI_CMO` has to be set in order to compile the gui plug-in code (see Section 5.3.2).

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 [7]: so you can use any Lablgtk2-compatible code in your gui extension. A complete exemple of gui extension may be found in plug-in `Occurrence` (see file `src/occurrence/register_gui.ml`).

Potential problems 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.

4.12 Documentation

Prerequisite: *knowledge of* ocamldoc.

Here we present some hints on the way to document your plug-in. First Section 4.12.1 introduces a quick general overview about the documentation process. Next Section 4.12.2 focus on the plug-in source documentation. Finaly Section 4.12.3 explains how to modify the Frama-C website.

4.12.1 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.

4.12.2 Plug-in Source Documentation

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

First of all, 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 capitalisation of the filename which is unusual in OCaml but here required for compilation purpose.

Besides, the 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.in` (see Section 5.3.2).

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.

4.12.3 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
```

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.*

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

4.13 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.in`. 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 4.27 *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
	`external`	Source of external free libraries	
Tests	`tests`	Frama-C test suites	Section 4.4
	`ptests`	ptests implementation	
Generated Files	`bin`	Binaries	
	`lib`	Some compiled files	
Documentations	`doc`	Documentation directory	
	`headers`	Headers of source files	Section 4.13
	`licenses`	Licenses used by plug-ins and kernel	Section 4.13
Shared libraries	`share`	Shared files	

Figure 5.1: Frama-C directories.

- The Frama-C root directory contains the configuration files, makefiles and some information files (in uppercase).
- Frama-C sources are split in three directories: `src` (described in Section 5.1.2) contains the core of the implementation while `cil` (described in Section 5.1.1) and `external` respectively

contains the implementation of Cil (extended with ACSL) and external libraries included in the Frama-C distribution.

- Directory `tests` contains the Frama-C test suite which is used by tool `ptests` (see Section 4.4).
- Directories `bin` and `lib` contains binary files mainly produced by Frama-C compilation. In particular Frama-C executables belong to directory `bin` while directory `lib/plugins` receives the compiled plug-ins. 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 4.13).
- 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).

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 4.5.1) 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	
	`ai`	Abstract interpretation toolbox	Section 4.9
	`memory_states`	Memory-state toolbox	Section 4.9
Libraries	`project`	Project library	Section 4.6
	`lib`	Miscellaneous libraries	
	`misc`	Additional useful operations	
Entry points	`toplevel`	Frama-C toplevel	Sections 4.7 and 4.8
	`gui`	Graphical User Interface	Section 4.11

Figure 5.3: Kernel directories.

- Directory `kernel` contains the kernel toolbox over Cil. Main kernel modules are shown in Figure 5.4.
- Directories `ai` and `memory_states` are the abstract interpretation and memory-state toolboxes (see section 4.9). 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 4.6.
- 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 (see Section 4.5.2).
- Directory `toplevel`[1] contains the Frama-C toplevel. In particular, module `Main` defines the main Frama-C entry point (see Section 4.7) and module `Options` manages the Frama-C command line (see Section 4.8).
- Directory `gui`[1] contains the `gui` implementation part common to all plug-ins. See Section 4.11 for more details.

5.2 Configure.in

Figure 5.5 presents the different parts of `configure.in` in the order that they are introduced in the file. The second row of the tabular says whether the given part has to be modified eventually

[1] From the outside, `gui` and `toplevel` may be seen as plug-ins with some exceptions because it has to be linked at the end of the link process.

Kind	Name	Specification	Reference
AST	`Cil_state`	The Cil AST for Frama-C	
	`Ast_info`	Useful operations over the Cil AST	
Specific information	`File`	AST initialisers and accesses to C files	
	`Globals`	Operations on globals	
	`Kernel_function`	Operations on functions	
	`Annotations`	Manage annotations at a propram point	
	`Loop`	Operations on loops	
Database	`Db`	Static plug-in database	Section 4.5.1
	`Db_types`	Type declarations required by `Db`	Section 4.5.1
	`Dynamic`	Interface for dynamic plug-ins	Section 4.5.2
	`Kui`	High-level Frama-C front-end	
Base Modules	`Version`	Information about Frama-C version	
	`Cmdline`	Command line options	Section 4.8
	`CilE`	Useful Cil extensions	
	`Alarms`	Alarm management	
	`Kernel_type`	Type value for kernel types	Section 4.5.2
	`Stmts_graph`	Accessibility checks using CFG	
Visitor	`Visitor`	Frama-C visitors subsuming the Cil ones	Section 4.10
Project	`Kernel_datatype`	High-level datatype builders	Section 4.6.4
	`Kernel_computation`	High-level internal state builders	Section 4.6.5
ACSL printers	`Ast_printer`	Pretty-printer for annotations	
	`Printer`	Class for pretty-printing annotations	
Initializer	`Boot`	Last linked module	Section 4.7

Figure 5.4: Main kernel modules.

by a 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 other mandatory tools/libraries	no	
4	Configuration of other non-mandatory tools/libraries	no	
5	Platform configuration	no	
6	Wished Frama-C plug-in	YES	Sections 4.2.2 and 4.2.3
7	Configuration of plug-in tools/libraries	YES	Section 4.2.5
8	Checking plug-in dependencies	YES	Section 4.2.4
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.in` including information provided by this configuration.

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

5.3 Makefile.in

In this section, we detail the organization of `Makefile.in`. First Section 5.3.1 presents the different sections of this file. Then Section 5.3.2 details variables introduced by `Makefile.plugin`.

5.3.1 Sections

Figure 5.6 presents the different parts of `Makefile.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 plug-in developer. More details are provided below.

Id	Name	Mod.	Reference
1	Global variables from `configure`	no	
2	Shell commands	no	
3	Command names	no	
4	Global plug-in variables	no	
5	Additional global variables	no	
6	Main targets	no	
7	External libraries to compile	no	
8	Internal miscellaneous libraries	no	
9	Kernel	no	
10	Plug-ins	YES	Section 4.3
11	Frontends	no	
12	Generic rules	no	
13	Tests	no	
14	Emacs tags	no	
15	Documentation	no	
16	Distribution	YES	???
17	File headers: license policy	YES	Section 4.13
18	Makefile rebuilting	no	
19	Cleaning	no	
20	Depend	no	
21	ptests	no	

Figure 5.6: Sections of `Makefile.in`.

1. **Global variables from configure** contains variable declarations from variables defined in `configure.in` (see Section 4.2). In particular, set variable `VERBOSEMAKE` to `yes` in order to see the right make commands in the user console. The typical use is

```
$ make VERBOSEMAKE=yes
```

2. **Shell commands** defines shortcuts which should be used in the makefile.

3. **Command names** defines command names displayed on the console in the non-verbose mode.

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

5. **Additional global variables** declares some other variables used throughout the makefile. In particular, it declares `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 5.3.2).

6. **Main targets** defines 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.

7. **External libraries to compile** provides variables and rules for external libraries required by Frama-C. Each library is in a specific sub-section.

8. **Internal miscellaneous libraries** provides variables and rules for Frama-C internal libraries (`Cil` and `Project`), each described in a specific sub-section.

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

10. After Section "Kernel", there are several sections corresponding to **plug-ins** (see Section 5.3.2). This is the part that a plug-in developer has to modify in order to add compilation directives for its plug-in.

11. After plug-in sections, there are sections corresponding to different Frama-C frontends (in particular, Sections **toplevel**, **gui** and **obfuscator**).

12. **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)

13. **Tests** provides rules to execute tests (see Section 4.4).

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

15. **Documentation** provides rules generating Frama-C source documentation (see Section 4.12).

16. **Distribution** provides rules which install the Frama-C distribution.

17. **File headers: license policy** provides variables and rules to manage the Frama-C license policy (see Section 4.13).

18. **Makefile rebuilting** provides rules in order to automatically rebuild `Makefile` and `configure` when required.

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

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

21. **Ptests** provides rules in order to build ptests (see Section 4.4).

5.3.2 Variables of `Makefile.plugin`

Figure 5.7 presents all the variables that can be set before including `Makefile.plugin` (see Section 4.3). Details are provided below.

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

  So it must be capitalised (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
	`PLUGIN_ENABLE`	Whether the plug-in has to be compiled
	`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`
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
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`	Directories containing plug-in 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
	`PLUGIN_DISTRIBUTED`	Whether to include the plug-in in source distribution
	`PLUGIN_DISTRIB_EXTERNAL`	Additional files to be included in the distribution

Figure 5.7: Parameters of `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_HAS_MLI` must be set to `yes` if plug-in *plugin* gets a file `.mli` (which must be capitalised: `Plugin.mli`, see Section 4.12) 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 4.5.1). 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 4.11). As for variable `PLUGIN_CMO`, do not write their file path nor their file extension.
- Variables `PLUGIN_BFLAGS`, `PLUGIN_OFLAGS`, `PLUGIN_DEPFLAGS` and `PLUGIN_DOCFLAGS` are plug-in specific flags for respectively `ocamlc`, `ocamlopt`, `ocamldep` and `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 `$(PLUGIN_DIR)`.
- Variable `PLUGIN_TYPES_TODOC` is the additional source files to document with the plug-in. They usually belong to `src/`*plugin*`_types` where *plugin* is the plug-in name (see Section 4.5.1).
- Variable `PLUGIN_INTRO` is the text file to append to the plug-in documentation introduction. Usually this file is `doc/code/intro_`*plugin*`.txt` for a plug-in *plugin*. It can contain any text understood by `ocamldoc`.
- Variable `PLUGIN_HAS_EXT_DOC` is set to yes if the plug-in has its own reference manual. It is supposed to be a `pdf` file generated by `make` in directory `doc/$(PLUGIN_NAME)`
- Variable `PLUGIN_NO_TEST` must be set to `yes` if there is no specific test directory for the plug-in.
- Variable `PLUGIN_TESTS_DIRS` is the directories containing plug-in tests. Its default value is `tests/$(notdir $(PLUGIN_DIR))`).
- Variable `PLUGIN_TESTS_LIB` is the `.cmo` plug-in specific files used by plug-in tests. Do not write its file path (which is `$(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 $(PLUGIN_NAME)_tests`.
- Variable `PLUGIN_DISTRIB_BIN` indicates whether the plug-in should be included in a binary distribution.
- Variable `PLUGIN_DISTRIBUTED` indicates whether the plug-in should be included in a source distribution.
- Variable `PLUGIN_DISTRIB_EXTERNAL` is the list of files that should be included in the source distribution for this plug-in, outside of the `src/$(PLUGIN_NAME)` directory (and the test and documentation directories if any).

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 5.8. 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].

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`
Dependencies	*plugin*`_DEPFLAGS` *plugin*`_GENERATED`
Documentation	*plugin*`_DOCFLAGS` *plugin*`_TYPES_TODOC`
Testing	*plugin*`_TESTS_DIRS` *plugin*`_TESTS_LIB`

Figure 5.8: Variables defined by `Makefile.plugin`.

5.4 Testing

Section 4.4 explains how to test a plug-in. Here Figure 5.9 details the options of ptests while Figure 5.10 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. Any directive can identify a file using a relative path. The default directory considered for `.` is always the parent directory of directory

[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	Specification	Default
Toplevel	`-add-options`	Additional options to be passed to the toplevel	
	`-byte`	Use bytecode toplevel	no
	`-opt`	Use native toplevel	yes
Behavior	`-run`	Delete results, run tests and examine their results	yes
	`-examine`	Only examine the current results; do not run tests	no
	`-show`	Run tests and show their results; also set `-byte`	no
	`-update`	Take the current results as oracles; do not run tests	no
Misc.	`-diff cmd`	Use `cmd` in order to compare results and oracles	`diff -u`
	`-j n`	Set level of parallelism to `n`	4
	`-v`	Increase verbosity (up to twice)	0
	`-help`	Display helps	no

Figure 5.9: ptests options.

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 to current default	*None*
	`EXECNOW`	Run a command before the test	*None*
	`FILTER`	Command name 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 (present for compatibility)	*None*

Figure 5.10: Directives in configuration headers of test files.

`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).

As said in Section 4.4.2, these directives can be found in different places:

1. default value of the directive (as specified in Fig. 5.10);
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 4.4.3, alternative directives for test configuration <special_name> can be found in slightly different places:

- default value of the directive (as specified in Fig. 5.10);
- 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.

```
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 following tests. `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 changes in this documentation in each Frama-C release. First we list changes of the last release.

- **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

We list changes of previous releases below.

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] CEA LIST, Software Reliability Laboratory. *Documentation of the static analysis tool ValViewer*, May 2008. http://www.frama-c.cea.fr.

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

[4] 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.

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

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

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

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

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

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

[11] 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.

List of Figures

Index