Version Hydrogen-20080502
May 26, 2008

Plugin 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 plugin 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 Hydrogen-20080502 (May 26, 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 and Anne Pacalet.

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 plugin.

It is organised in four parts. The first one, Chapter 2, is a step-by-step tutorial for developing a new plugin 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 plugin developements. The second part, Chapter 3, presents the design of the Frama-C software architecture. The third part, Chapter 4, details how to use the services provided by Frama-C in order to be fully operational with the development of Frama-C plugins. The forth part, chapter 5, is a short reference manual which fully documents some particular points of the Frama-C platform.

Most important parts are displayed inside a gray box like this one. A plugin developer **must** take them carefully.

Chapter 2

Tutorial

This chapter aims at helping a developer to write his first Frama-C plugin. 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 plugin developements. With this goal in mind, this chapter was written to explain step-by-step how to proceed. This tutorial mainly explains how to integrate a plugin inside the Frama-C platform but does not focus on how to properly write a whole analysis. In particular, some very important aspects for such a purpose are fully omited here and are described in chapter 4.

2.1 Setup

If you have a CVS access to the Frama-C repository, it is possible to download the sources for Frama-C with the CVS command below[1] where *login* is your CVS login and *cvsserver* is the name of the Frama-CCVS server name[2].

```
$ 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 plugins that should be available. It is generated itself by the autotool autoconf. So you have to execute the following commands:

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

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

```
$ make -j
```

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

This compilation usually 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` above correspond respectively 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 helping tags for emacs with `make tags`.

2.2 Plugin Integration Overview

Figure 2.1 shows how a plugin should be integrated in the Frama-C platform. Most parts of this Figure are concretely explained in the remainder sections of this tutorial.

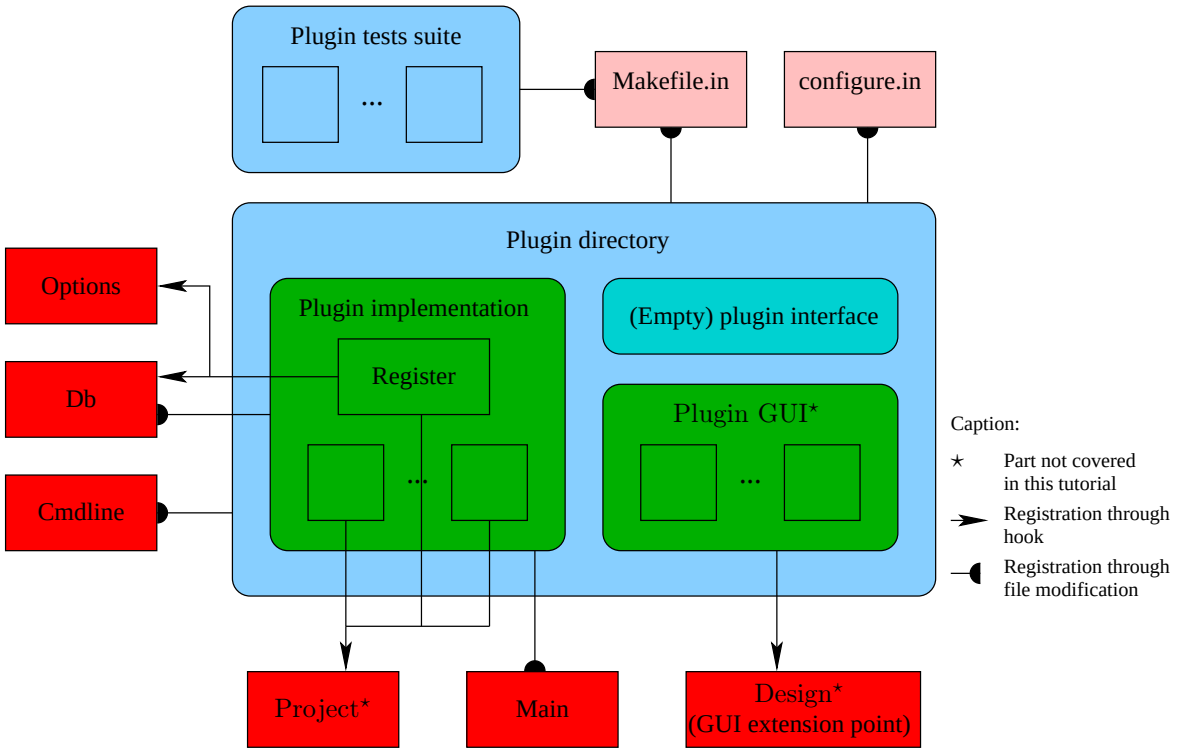

Figure 2.1: Plugin Integration Overview.

The implementation of the plugin 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 plugin-specific operations and to register them inside it if you want that someone uses your plugin (see Section 2.5);
- to extend the Frama-C entry point defined in module `Main` if you want to run plugin-specific code when Frama-C is executed (see Section 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 plugin-specific option to the Frama-C command line (see Section 2.6).

He has also to modify compilation files in order to properly link his plugin with Frama-C (see Section 2.4).

Moreover, the developer may provided a plugin interface (which should usually be empty, see Section 2.5) and specific tests suite (see Section 2.7).

2.3 Hello World

In this Section, we explain how to write the core of a plugin `Hello`. This is a tiny plugin which pretty-prints the names of the input files given to Frama-C if 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 plugin, but we use this example to introduce the bases of plugin development. This plugin is our running example in this chapter.

First of all, we add a new sub-directory `hello` in directory `src`.

```
$ mkdir src/hello
```

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

Now we are able to 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* `plugin` *should always be called either* `register.ml` *or* `plugin_register.ml`*.*

File *src/hello/register.ml*

```
let run () =
  Format.printf "Hello Frama-C World.\nInput files are:\n";
  List.iter (fun s → Format.printf "\t%s\n" s) (Cmdline.Files.get ());
  Format.print_flush ()
```

This file defines a function `run`. It uses the OCaml module `Format` to pretty-print the list of input filenames given by `Cmdline.Files.get ()`.

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

More generally, the Frama-C module `Cmdline` provides information about the Frama-C command line (options and files set by the user, see Section 2.6).

At this point, we have a compilable plugin containing its main function.

2.4 Configuration and Compilation

Here we explain how to compile the plugin `hello`. Section 4.2 and 4.3 provide more details about configuration and compilation of plugins.

Configuration As explained in Section 2.1, Frama-C uses both autoconf and make in order to compile. So we have to modify both files `configure.in` and `Makefile.in` in order to compile our plugin within Frama-C. In both files, some predefined scripts help with plugin integration.

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

File *configure.in*

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

These lines correspond to the standard scheme for configuring a new plugin. Function `check_plugin` is defined in `configure.in`. Its first argument is the plugin name, the second one is the plugin directory (the directory containing the plugin source files), the third one is an help message and the fourth one indicates whether the plugin is available by default or not (here `yes` says that the plugin is available by default and an user may use option `-disable-hello` to disactivate the plugin).

Now we are ready to execute

```
$ ./configure
```

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

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

in the configuration summary.

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

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 plugins 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 plugin compilation. A plugin developer can set some variables before including `Makefile.plugin` in order to control its behaviour. Now we briefly explain the variables set for `hello`.

- `PLUGIN_ENABLE` says whether the plugin should be compiled or not. Here we use the variable `@ENABLE_HELLO@` set by `configure.in`.
- `PLUGIN_NAME` is the name of the plugin.

  It must be a valid OCaml module name (in particular it must be capitalised).
- `PLUGIN_DIR` is the directory containing the source file(s) for the plugin.
- `PLUGIN_CMO` is the list of the `.cmo` files (without the extension `.cmo` or the plugin path) required to compile the plugin.
- `PLUGIN_NO_TEST` is set to yes when there is no specific test directory for the plugin (see Section 2.7 about plugin testing).

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

```
$ make -j
```

2.5 Connection with the Frama-C World

The plugin `hello` is now compiled but it is not strongly connected to the Frama-C framework. In particular, our plugin should be added in the plugin database `Db` in order to be used by other plugins (see Chapter 3 for details).

Extension of the Plugin Database For this purpose, we have to extend `Db` with the new plugin `hello`.

File *src/kernel/db.mli*

```
...
(** Hello World plugin.
    @see <../hello/index.html> internal documentation. *)
module Hello : sig
  val run: (unit → 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 plugin itself. if not, a call to `!Db.run` raises exception `Extlib.NotYetImplemented`. So we modify module `Register` in the following way.

File *src/hello/register.ml*

```
··· definition of run
let () = Db.Hello.run := run
```

It is important to notice 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 generate an internal documentation for `hello` which is accessible in the directory `doc/code/hello`.

Hiding the Implementation Last but not least, we hide the implementation of `hello` to other developers in order to enforce the architecture invariant which is that each plugin should be used through `Db` (see Chapter **??**). For this purpose we only add an empty interface to the plugin in the following way.

File *src/hello/Hello.mli*

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

```
(** Hello World plugin.

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

Indeed, thanks to `Makefile.plugin`, each plugin is packed into a whole 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 for `Hello`. So, 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
```

2.6 Extension to the Command Line

Now, in order to complete our plugin, we have to add an option on the Frama-C command line and to execute the function `!Db.Hello.run` when this option is set by a user. Section 4.8 provides more details about extensions of the command line.

First we add a value in the module `Cmdline` which says whether the user sets the option `-hello` on the command line or not (*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 plugins. The above lines of codes add a module `Hello` which contains all the options for `hello`. In fact we only have one option, called `print`. In Frama-C, each option indeed looks like a module. The signature of the module indicates

the type of the option: 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 introduce this value, we are able to add the option `-hello` to the toplevel command line by extending the plugin `hello`.

File *src/hello/register.ml*

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

We use function `Options.add_plugin` which integrates the new option `-hello` and modifies the value contained in `Cmdline.Hello.Print` as well. This function also adds information about the plugin `hello` when the predefined option `-help` is set by the user.

Finally we modify the "main" of Frama-C (*i.e.* its entry point) in order to execute the new plugin when its option is set.

File *src/toplevel/main.ml*

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

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

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

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. We give here only some hints. First we have to create a test directory for `hello`

```
$ mkdir tests/hello
```

and we can remove in `Makefile.in` the line which sets `PLUGIN_NO_TEST`.

File *Makefile.in*

```
# ··· Place of variables of plugin 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 plugin hello does not require C code anyway. */
```

Of course, it is possible to test the new plugin 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
Analysed files are:
        tests/hello/hello.c
```

The specific output of the plugin `hello` are the last two lines.

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

```
$ ./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 test `tests/hello/hello.c`, the directive says 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 plugin `hello` are compliant or not 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 plugins by executing ptests on all default tests with `make tests`. It is also possible to add plugin `hello` to the default test suite by editing the value of the variable `default_suites` at the top of file `ptests/ptests.ml`.

Note to CVS users If you have got a 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 be sure 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"
```

The first line adds new files into the archive (if any) while the second one updates your local version with the root repository. The third line performs non-regression tests. Finally, *if and only if you have no problem,* you can commit your changes thanks to the last line.

2.8 Copyright your Work

If you want to redistribute plugin `hello`, you have to choose a licence policy for it (compatible with Frama-C). Section 4.13 gives details about how to proceed in a general way. Here, suppose we want to put the plugin `hello` under the Lesser General Public Licence (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 plugin `hello` in order to indicate that they are under the desired licence.

Chapter 3

Software Architecture

In this chapter, we present the software architecture of Frama-C. First, Section 3.1 presents its general overview. Then, three main parts are separately detailed: Section 3.2 introduces the API of Cil [10] seen by Frama-C, Section 3.3 shown the organisation of the Frama-C kernel and Section 3.4 explains the plugin 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 plugins 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 plugins; and
- plugins 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) [10] extended with an implementation of the specification language ACSL (ANSI C Specification Language) [1]. That is the intermediate language which Frama-C is based on. See Section **??** 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 **??** for details.
- Frama-C plugins. That is analyses or source-to-source transformations which use the kernel and possibly others plugins through the plugins database called `Db`. See Section **??** for details.

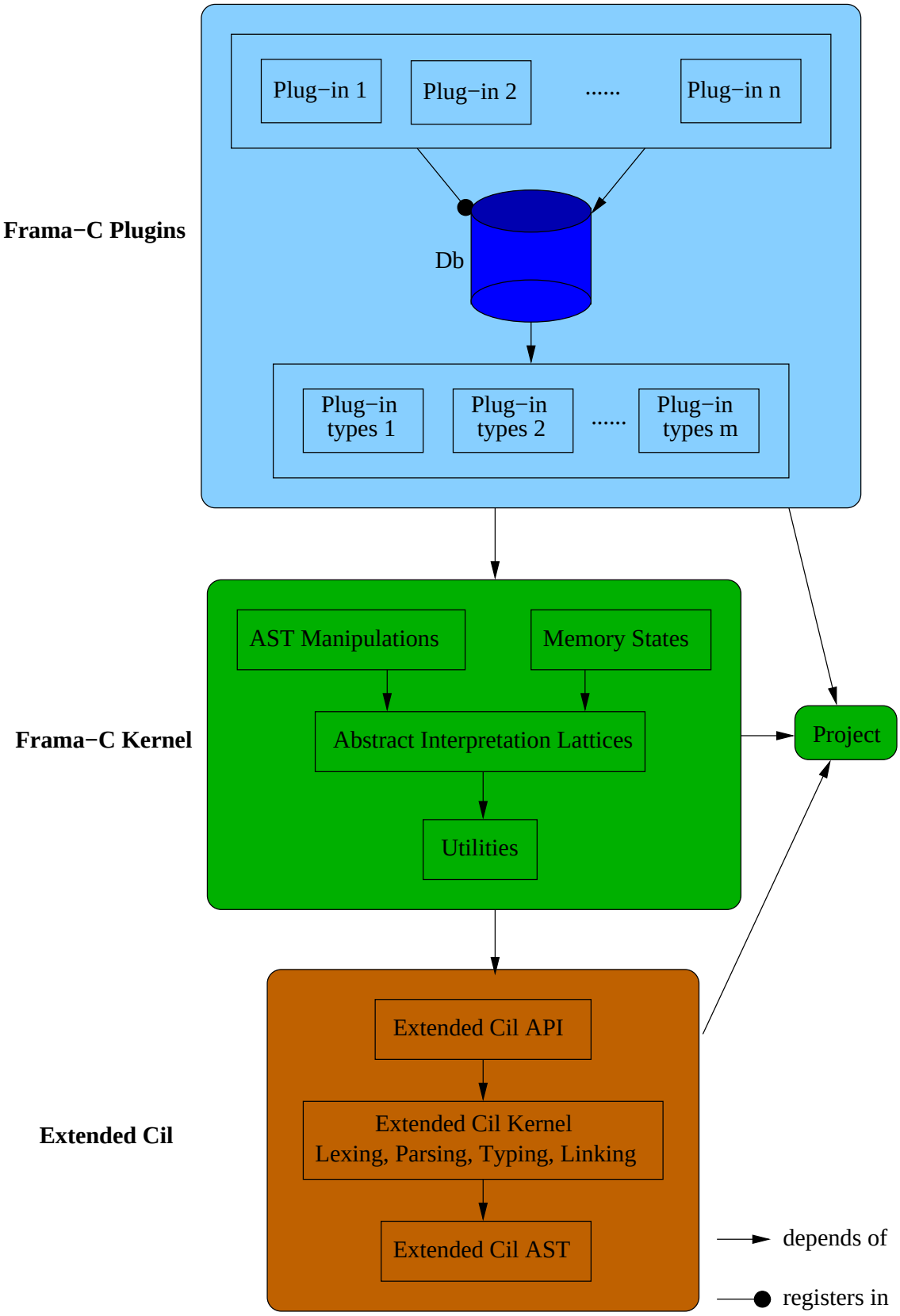

Figure 3.1: Architecture Design.

3.2 Cil: C Intermediate Language

Cil [10] 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 main steps of compilation of C programs (preprocessing, 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 analyses like a (syntactic) call graph computation (module `Callgraph`) or generic forward/backward dataflow analyses (module `Dataflow`).

Frama-C indeed extends Cil with ACSL (ANSI C Specification Language) [1], its specification language. So, the extended Cil API 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 datastructures 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-analyses 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 plugins

In Frama-C, plugins are analyses or source-to-source transformations. Each of them is an extension point of the Frama-C kernel. A database, called `Db` (in directory `src/kernel`), groups together all of them. It also provides their API which permit easy plugins collaborations. Each plugin is only visible through `Db`. For example, if a plugin `A` wants to know results of another plugin `B`, it uses the part of `Db` corresponding to `B`. A consequence of this design is that each plugin has to register in `Db` in order to be usable from others plugins.

Besides, some plugin defines some own datatypes (eventually coming with some specific operations) which have to be visible from outside (usually because the plugin API uses them). In order to keep as small as possible the plugins database `Db`, these datatypes are put outside of it. For visibility purpose, they are also put outside their plugin owners. That is the *raison d'être* of plugins types.

Chapter 4

Advanced Plugin Development

This chapter details how to use services provided by Frama-C in order to be fully operational with the development of plugins. Each section describes technical points which a developer should be aware of. If you are not aware of, the result could be, from the less bad situation to the whorst one[1]:

1. reinventing the (Frama-C) whole;
2. being not able 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 plugins that use your own one;
5. breaking the kernel consistency and so potentially breaking all the Frama-C plugins (*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 plugin like `hello` described in chapter 2 and knows about the software architecture of Frama-C 3. Moreover plugin 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).

A reader can obviously read this chapter from the beginning to the end. He can yet pick the wished information in any section in any order. Pointers to reference manuals (Chapter 5) are also provided for readers who want full details about specific parts.

4.1 File Tree Overview

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 [10] extended with 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 plugin.

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 worse situation by mistake if you are not a kernel developer.

4.2 Configure.in

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

In this Section, we detail how to modify the file `configure.in` in order to configure plugins (Frama-C configuration has been introduced in Section 2.1 and 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 plugin without any dependency. Next we show how to exhibit dependencies with external libraries and tools (Section 4.2.3) and with other plugins (Section 4.2.4). Finally Section 4.2.5 presents the configuration of external libraries and tools needed by a new plugin 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 plugin which can be:

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

The important notion in the above definitions is *dependency*. A dependency of a plugin p is either an external library/tool or another Frama-C plugin. 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 plugin developer, the main role of `configure.in` is to define the optional and mandatory dependencies of each plugin. Another standard job of `configure.in` is the addition of options `--enable-`p and `--disable-`p to `configure` for a plugin 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 plugin in Frama-C.

4.2.2 Addition of a simple plugin

In order to add a new plugin, there are three actions to perform:

1. first add a new subsection for the new plugin to Section *Plugin wished*;
2. then add a new substitution in Section *Substitutions to perform*; and
3. finally add a new entry in Section *Summary*.

All these actions are very easy to perform by copy/paste from another existing plugin (*e.g.* `occurrence`) and by replacing the plugin name (here `occurrence`) by the new plugin name in the pasted part. In these sections, plugins are sorted in a lexicographic ordering. In order to well understand what is defined by this copy/paste, we explain how `occurrence` is defined in these sections.

First Section *Wished Plugin* introduces a new subsection for this plugin in the following way.

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

The first argument is the plugin name, the second one is the name of directory containing the source files of the plugin, the third one is a help message for the `-enable-occurence` option of configure and the last one indicates if the plugin 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 plugins in order to handle their dependencies (see Section 4.2.4). Finally `ENABLE_OCCURRENCE` indicates the plugin status (available, partially available or not available). At the end of these lines of code, it says if the plugin should be compiled: if `--enable-occurrence` is set, then `ENABLE_OCCURRENCE` is `yes` (plugin available); if `--disable-occurrence`, then its value is `no` (plugin 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])
```

Thus the value `@ENABLE_OCCURRENCE` is usable in `Makefile.in` in order to know whether the plugin has to be compiled or not (see Section 4.3) and a notification is displayed to the user which indicates this value and an optional informative message (contained in `$INFO_OCCURRENCE`).

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 plugin names (separated by spaces). It contains the plugins 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 plugin *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 *Plugin* `gui` *requires* Lablgtk2 *[6]. 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
```

They say that Lablgtk2 *must be available on the system if the user wants to compile* `gui`.

4.2.4 Addition of plugin dependencies

Adding a dependency with another plugin 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 plugin names for which *plugin* is respectively a mandatory dependency or an optional dependency).

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

Example 4.2 *Plugin* `value` *requires plugin* `from` *and may use plugin* `gui` *(for* ValViewer *[2]). So lists* `REQUIRE_FROM` *and* `USE_GUI` *contain* `value`. *Moreover, as many plugins require* `value`, *list* `REQUIRE_VALUE` *is quite big. In particular, it contains plugin* `from`: *both plugins* `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 the same: so, in this section, we only 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 Plugin 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 script called `configure_library` helps the plugin 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. 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 plugins requiring (resp. optionally using) it[3].

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.cma \
   $OCAMLLIB/lablgtksourceview/lablgtksourceview.cma" \
  "lablgtksourceview not found"
```

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

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

4.3 Makefile.in

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

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

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 plugin, you have to extend section "Plugins". For this purpose, you have to include `Makefile.plugin` for each new plugin (hence there are as many lines `include Makefile.plugin` as plugins). `Makefile.plugin` is a generic makefile dedicated to plugin compilation. Before its inclusion, a plugin 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 plugin `Value`, the following lines are added into `Makefile.in`.*

```
##################
# 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-specific variables once `Makefile.plugin` has been included. These variables are detailed in Section 5.3.2.

[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 [5]).

One other variable has to be modified by a plugin developer if he uses files which do not belong to the plugin directory (that is if variable `PLUGIN_TYPES_CMO` is set). This variable is `UNPACKED_DIRS` and corresponds to the list of non-plugin directories containing source files.

A plugin 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.

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 plugin `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 plugin* `plugin` *has changed, a typical sequence of tests is the following one.*

```
$ ./bin/ptests.byte plugin
$ ./bin/ptests.byte -update plugin
$ make tests
```

So we first run the tests suite corresponding to `plugin` *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
```

4.5 Exporting Datatypes

If a plugin has to export some datatypes usable by other plugins, such datatypes have to be visible from module `Db`. Thus they cannot be declared in the plugin implementation itself like any other plugin 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 plugin 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 plugin 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 4.8 *Suppose you are writing a plugin `plugin` which exports a specific type `t` corresponding to the result of the plugin 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 plugin code outside plugin 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).

4.6 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 [8, 9] 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.

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

4.6.2 Using Projects

Actually Frama-C maintains a default project (`Project.current ()`) and a default AST (`Cil_state.file ()`) which all operations are automatically performed on. But sometimes a plugin 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 4.9 *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
  Project.set_current foo;
  !Db.Valsue.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`.

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

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 4.10 *The following code is equivalent to the one given in Example 4.9.*

```
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 4.11 *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]. Section 4.6.5 details how to register a new computation.

The registration of a data of type τ requires to register the type τ itself as a *datatype* using functor `Project.Datatype.Register`. A datatype is a type that is aware of projects. Similarly to computations, module `Datatype` (resp. `Kernel_datatype`) provides pre-defined datatypes and datatypes-builder for elaborated types[8]. Section 4.6.4 details how to register a new datatype.

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

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

[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 [4, 7, 3].

provide a copy function for imperative (*i.e.* mutable) datatypes. This copy function is only used by `Project.copy`.

Example 4.12 *For registering a couple composed of an integer and a boolean, just apply*

```
Project.Datatype.Persistent(struct type t = int × bool end)
```

For registering a type `a` containing a mutable field `a`, just do

```
type a = { mutable a : int }
Project.Datatype.Imperative
  (struct
     type t = a
     let copy x = { a = x.a }
   end)
```

Using predefined datatypes or datatype builders For most useful types, the corresponding datatypes are already provided in modules `Datatype` (*e.g.* `Datatype.Int` for type `int`) and `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 4.13 *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).
- Functions `before_load` and `after_load` which are applied at load time (when function `Project.load` or `Project.load_all` is applied). If there is no action to apply, just include module `Datatype.Nop` instead.
- A name for the datatype. The type of this value is `Project.Datatype.Name.t` which is isomorphic to `string`. The only difference is name unicity: if the same name is used for two different registrations, an exception is occurring at the module initialisation time of Frama-C (*i.e.* at runtime, but before mostly anything else, see Section 4.7).

- A list of dependencies which should be the list of datatypes used by the datatype under registration. Datatypes are build by functor applications which provide a value `self` in the output module. This value is called the *kind* of the datatype and can be used for this purpose. Roughly speaking, it represents the type itself.

Example 4.14 *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
     include Datatype.Nop (* before_load and after_load do nothing. *)
     let name = Project.Datatype.Name.make "postdominator"
     let dependencies = [ Kernel_datatype.StmtSet.self ]
   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 = Project.Computation.Name.make "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` and `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. Name and dependencies managements are similar to those for datatypes (see Section 4.6.4). The only difference concerning dependencies is that one has to provide each state requiring by the state computation: here that is the state of the value analysis.

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 plugin p may want to export the kind of its internal state (also called *state kind*; in the previous example, that is value `State.self`). This exportation offers the possibility to other plugins 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). 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 plugin initialisation time (see Section 4.7). Now if another plugin 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 plugin initialisation is not specified (see Sec-

tion 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 4.15 *Plugin `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 = Project.Computation.Name.make "functionwise_from"
       let size = 97
       let dependencies = [ Value.self ]
     end)
let () = Db.From.self := Functionwise_Dependencies.self
                (* performed at module initialisation runtime. *)
```

Plugin `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 = Project.Computation.Name.make "Pdg.State"
       let dependencies = [] (* postponed *)
       let size = 97
     end)
let () =
  Options.register_plugin_init
    (fun () → Project.Computation.add_dependency Tbl.self !Db.From.self)
```

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 this 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]

$$\texttt{create () != create ()} \tag{4.1}$$

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

$$\forall \texttt{p of type t}, \texttt{set p; p == get ()} \tag{4.3}$$

$$\forall \texttt{p1, p2 of type t}, \texttt{set p1; let p = get () in set p2; p != get ()} \tag{4.4}$$

The first invariant ensures that there is no sharing between fresh values of a same internal state: so each new project has got its own fresh internal state. The second invariant ensures that cleaning a state resets it to its initial value. The third invariant ensures that the version of the global project remains physically equal to the local version when the current project changes. Last the fourth invariant 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. The two last invariants usually require an additional indirection in order to be safely (and efficiently) implemented.

Example 4.16 *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.*

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

```
  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
```

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

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

```
Project.clear
  ~only:(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 4.18 *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: *Knowledges 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 plugin files
   (f) gui non-plugin files (in `src/gui`)[11]
   (g) gui plugin files[11]

[11] If the graphical user interface is compiled.

(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` (in an unspecified order). Usually these functions are used for plugin initialisations.
4. Parsing the Frama-C command line.
5. Running each function registered through argument `init` (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).

4.8 Command Line Options

Prerequisite: *Knowledges 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`. First Section 4.8.1 introduces how to store new option values. Second Section 4.8.2 presents how to register new options.

4.8.1 Storing new 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 4.19 *Value for option* `-slevel` *is 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 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 plugin* `plugin` *should belong to a same module* `PluginOptions` *inside* `Cmdline`*.*

4.8.2 Registering new options

You have to use function `Options.add_plugin` for registering all options of a plugin. 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 plugin 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 4.20 *For illustrating the use of this function, we show how two plugins use it. First consider plugin* `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 plugin ***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 plugin (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 plugin ***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 plugin: 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;

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

- `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.

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 `behavior`, which comes in two flavors: `inplace_behavior` and `copy_behavior`. 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 as follows:

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

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.

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 plugin 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 plugin 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 main widgets of a Frama-C gui.

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

Mainly that's all! The gui implementation uses Lablgtk2 [6]: so you can use any Lablgtk2-compatible code in your gui extension.

Example 4.21 *We illustrate the principle with plugin* `occurrence`*. This plugin computes all the places where a variable declaration is used: for a variable* `v`*, function* `!Db.Occurrence.get` *provides a list of occurrences of* `v` *which are couples containing a statement* `s` *and a left-value in* `s` *which uses some part of the memory location corresponding to* `v`*.*

In the gui, this plugin adds item "Occurrence" in the menu displayed when an user clicks with the right mouse button on a variable declaration `v`*. The selection of this item highlights each occurrence corresponding to* `v`*. Below is the annotated code implementing this feature*[13]*.*

First we open some useful modules.

```
open Pretty_source (* module coming with the Frama-C GUI *)
open Gtk_helper    (* module coming with the Frama-C GUI *)
open Db
open Cil_types
```

Next we extend the Frama-C gui with function `main` *below.*

```
let main main_ui = main_ui#register_source_selector occurrence_selector
let () = Design.register_extension main
```

This function takes the main Frama-C gui as parameter, using it in order to register an action performed when a button is released on some predefined place.

This action is provided by function `occurrence_selector` *below.*

```
let occurrence_selector
    (popup_factory:GMenu.menu GMenu.factory) main_ui ~button localizable =
  if button = 3 then
    (* right mouse button pressed *)
    match localizable with
    | PVDecl (_,vi) → (* variable declaration selected *)
        (* ignore variables which are functions *)
        if not (Cil.isFunctionType vi.vtype) then
          let callback () =
            (* compute occurrences of the selected variable *)
            let lvals = !Db.Occurrence.get vi in
            (* next highlight them (in yellow) *)
            apply_tag main_ui "occurrence" "yellow" lvals
          in
          (* add item "Occurrence" associated to callback in the popup *)
          ignore (popup_factory#add_item "_Occurrence" ~callback)
    | _ → ()
```

*This function takes four arguments. The first one is the factory corresponding to the popup displayed when an user clicks with the right mouse button, the second one is the main Frama-C gui, the third one is the pressed mouse button and the four one is the selected localizable. The localizable is a Frama-C variant indicating which piece of code in the viewer the user has selected (*e.g. *a statement or a left value).*

Here the function body adds item "Occurrence" if the right mouse button is pressed on a variable declaration (which is not a function). The action associated to this new item computes the occurrences of the selected variables and highlights them in yellow.

[13] File `src/occurrence/occurrence_gui.ml` contains the original source.

Function `apply_tag` *implements this highlighting. Its code is below.*

```
exception Highlight of Db_types.kernel_function × stmt
let apply_tag main_ui name color occurrences =
  (* create a new tag for the highlighting (if not previously built) *)
  let view = main_ui#source_viewer in
  let tag = make_tag view#buffer name [‘BACKGROUND color ] in
  (* erase previous tags in the buffer if any *)
  cleanup_tag view#buffer tag;
  (* highlight each occurrences *)
  List.iter (fun (ki, lv) → highlighting) occurrences
```

Badly the highlighting itself is a quite ugly because we try different solutions to convert an occurrence to a proper highlightable localizable. The code is below.

```
(* first compute the statement and the kernel function of the localizable *)
let skf, kf = match ki with
  | Kglobal → None, None
  | Kstmt s →
      let s, kf = Kernel_function.find_from_sid s.sid in
      Some (s, kf), Some kf
in
(* next try different highlighting solutions *)
let highlight = main_ui#highlight ~scroll:true tag in
let try_and_highlight loc =
  match Pretty_source.locate_localizable loc, skf with
  | None, None → invalid_arg "some occurrence cannot be highlighted"
  | None, Some (s, kf) → raise (Highlight(kf, s))
  | Some _, _ → highlight loc
in
(* try to highlight as a lval *)
try try_and_highlight (PLval (kf, ki, lv))
with Invalid_argument _ | Highlight _ →
  (* if that doesn't work, try to highlight as a term_lval *)
  try
    try_and_highlight
     (PTermLval (kf, ki, Logic_const.lval_to_term_lval lv))
  with
  | Highlight(kf, s) →
      (* possible to highlight the whole stmt *)
      highlight (PStmt (kf, s))
  | Invalid_argument msg →
      (* cannot highlight *)
      Format.printf "%s@." msg)
```

Potential problems All the gui plugin 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 plugin 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 plugin. First Section 4.12.1 introduces a quick general overview about the documentation process. Next Section 4.12.2 focus on the plugin 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 plugin) while `make $(PLUGIN_NAME)_DOC` (by substituting the right value for `$(PLUGIN_NAME)`) generates the documentation for a single plugin.

4.12.2 Plugin source documentation

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

First of all, a plugin should export itself no function: the only visible plugin interface should be in `Db`.

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

The interface name of a plugin `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 plugin documentation which may be useful for the plugin developer itself. This internal documentation is available *via* file `doc/code/`*plugin*`/index.html` for each plugin *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 plugin in the plugin list.

4.12.3 Website

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

For each plugin 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 plugin</h1>

··· Plugin 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`.

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

Prerequisite: *Knowledge of* make.

If you want to redistribute a plugin 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 plugin. Next run `make headers`.

  Example 4.22 *Plugin `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 licence is unknown inside** Frama-C , you have to:
  1. Add a new variable v corresponding to it and assign files of your plugin;
  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 plugin (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 plugin 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 plugin 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 plugins 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 plugins. You should never add yourself any file in these directories.
- Documentations (including plugin-specific, source code and ACSL documentations) are provided in directory `doc`. Directories `headers` and `licences` 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 plugin 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_type` 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 plugin developer.
- `src/logic` is the ACSL frontend which converts logic code to the corresponding Cil AST. The only useful modules for a Frama-C plugin 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 plugin implementation or some parts of the Frama-C kernel.

Each plugin 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) 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.
- 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 plugins. 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 by a plugin developer. More details are provided below.

[1]From the outside, `gui` and `toplevel` may be seen as plugins 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`	Plugin database	Section 4.5
	`Db_types`	Type declarations required by `Db`	
	`Kui`	High-level Frama-C front-end	
Base Modules	`Version`	Information about Frama-C version	Section 4.8
	`Cmdline`	Command line options	
	`CilE`	Useful Cil extensions	
	`Alarms`	Alarm management	
	`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.

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 plugin	YES	Sections 4.2.2 and 4.2.3
7	Configuration of plugin tools/libraries	YES	Section 4.2.5
8	Checking plugin 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 Plugins** sets which Frama-C plugins the user wants to compile.
7. **Configuration of plugin tools/libraries** checks the availability of external tools and libraries required by plugins for compilation and execution.
8. **Checking Plugin Dependencies** sets which plugins have to be disable (at least partially) because they depend on others plugins 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 plugin 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 plugin developer. More details are provided below.

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.

Id	Name	Mod.	Reference
1	Global variables from `configure`	no	
2	Shell commands	no	
3	Command names	no	
4	Global plugin 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	Plugins	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`.

4. **Global plugin variables** declares some plugin-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 plugin developer if he uses files which do not belong to the plugin 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 **plugins** (see Section 5.3.2). This is the part that a plugin developer has to modify in order to add compilation directives for its plugin.

11. After plugin 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 plugin.

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

- Variable `PLUGIN_DIR` is the directory containing plugin source files. It is usually set to `src/`*plugin* where *plugin* is the plugin name.
- Variable `PLUGIN_ENABLE` must be set to `yes` if the plugin has to be compiled. It is usually set to `@`*plugin*`_ENABLE@` provided by `configure.in` where *plugin* is the plugin name.
- Variable `PLUGIN_HAS_MLI` must be set to `yes` if plugin *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 plugin is used through `Db` (see Chapter 3).
- Variables `PLUGIN_CMO` and `PLUGIN_CMI` are respectively `.cmo` plugin files and `.cmi` files without corresponding `.cmo` plugin files. For each of them, do not write their file path nor their file extension: they are automatically added (`$(PLUGIN_DIR)/`*f*`.cm[io]` for a file *f*).
- Variable `PLUGIN_TYPES_CMO` is the `.cmo` plugin files which do not belong to `$(PLUGIN_DIR)`. They usually belong to `src/`*plugin*`_types` where *plugin* is the plugin name (see Section 4.5). Do not write file extension (which is `.cmo`): it is automatically added.
- Variable `PLUGIN_GUI_CMO` is the `.cmo` plugin files which have to be linked with the GUI (see Section 4.11). As for variable `PLUGIN_CMO`, fo not write their file path nor their file extension.

Kind	Name	Specification
Usual information	`PLUGIN_NAME`	Module name of the plugin
	`PLUGIN_DIR`	Directory containing plugin source files
	`PLUGIN_ENABLE`	Whether the plugin has to be compiled
	`PLUGIN_HAS_MLI`	Whether the plugin gets an interface
Source files	`PLUGIN_CMO`	`.cmo` plugin files
	`PLUGIN_CMI`	`.cmi` plugin files without corresponding `.cmo`
	`PLUGIN_TYPES_CMO`	`.cmo` plugin files not belonging to `$(PLUGIN_DIR)`
	`PLUGIN_GUI_CMO`	`.cmo` plugin files not belonging to `$(PLUGIN_DIR)`
Compilation flags	`PLUGIN_BFLAGS`	Plugin-specific flags for `ocamlc`
	`PLUGIN_OFLAGS`	Plugin-specific flags for `ocamlopt`
Dependencies	`PLUGIN_DEPFLAGS`	Plugin-specific flags for `ocamldep`
	`PLUGIN_GENERATED`	Plugin files to compiled before running `ocamldep`
	`PLUGIN_DEPENDS`	Other plugins to compiled before the considered one
Documentation	`PLUGIN_DOCFLAGS`	Plugin-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 plugin introduction
Testing	`PLUGIN_NO_TESTS`	Whether there is no plugin-specific test directory
	`PLUGIN_TESTS_DIRS`	Directories containing plugin tests
	`PLUGIN_TESTS_LIBS`	Specific `.cmo` files used by plugin tests

Figure 5.7: Parameters of `Makefile.plugin`.

- Variables `PLUGIN_BFLAGS`, `PLUGIN_OFLAGS`, `PLUGIN_DEPFLAGS` and `PLUGIN_DOCFLAGS` are plugin-specific flags for respectively `ocamlc`, `ocamlopt`, `ocamldep` and `ocamldoc`.
- Variable `PLUGIN_GENERATED` is files which must be generated before computing plugin dependencies. In particular, this is where `.ml` files generated by `ocamlyacc` and `ocamllex` must be placed if needed.
- Variable `PLUGIN_DEPENDS` is the other plugins which must be compiled before the considered plugin. Note that, in a normal context, it should not be used because a plugin 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 `$(PLUGIN_DIR)`.
- Variable `PLUGIN_TYPES_TODOC` is the additional source files to document with the plugin. They usually belong to `src/`*plugin*`_types` where *plugin* is the plugin name (see Section 4.5).
- Variable `PLUGIN_INTRO` is the text file to append to the plugin documentation introduction. Usually this file is `doc/code/intro_`*plugin*`.txt` for a plugin *plugin*. It can contain any text understood by `ocamldoc`.
- Variable `PLUGIN_NO_TEST` must be set to `yes` if there is no specific test directory for the plugin.
- Variable `PLUGIN_TESTS_DIRS` is the directories containing plugin tests. Its default value is `tests/$(notdir $(PLUGIN_DIR))`).

- Variable `PLUGIN_TESTS_LIB` is the `.cmo` plugin-specific files used by plugin tests. Do not write its file path (which is `$(PLUGIN_TESTS_DIRS)`) nor its file extension (which is `.cmo`).

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 plugin 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 plugin 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 plugin. 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.

- The default value for directive `CMD` is `./bin/toplevel.opt`. Here the default directory considered for `.` is always the parent one of directory `tests`.
- The default value for directive `OPT` is `-val -out -input -deps`). If there are several directives `OPT` in the same configuration, they correspond to different test cases.
- The syntax for directive `EXECNOW` is the following.

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

[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 plugin (*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
Command	`CMD`	Program name to run
	`OPT`	Options to be used by command specified through `CMD`
	`EXECNOW`	Run a command before running the test itself
	`FILTER`	Command name used to filter results
Test suite	`DONTRUN`	Do not execute this test
	`FILEREG`	Regular expression indicating which files have to be tested
Miscellaneous	`COMMENT`	Comment in the configuration
	`GCC`	Unused (only present for compatibility reasons)

Figure 5.10: Directives in configuration headers of test files.

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`.

- Directive `FILEREG` is only usable in a configuration file `test_config` (see Section 4.4.2). It is default value is `.*\.\(c|i\)`.

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, September 2006.

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

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

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

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

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

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

[10] 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