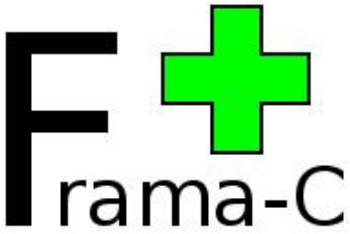

Version 2008.01.1+alpha

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

Chapter 1

Introduction

This guide aims at helping any developer which would like to program within the Frama-C platfom, in particular if he wishes to develop a new analysis or a new source-to-source transformation through a new plugin.

It is organised in two parts. The first one, Chapter 2, is a tutorial which shows step-by-step how to develop 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 both from other plugins (*via* the programatic interface) and from the Frama-C toplevel (*via* the command line). The second part, Chapter 3, details how to use the services provided by Frama-C in order to be fully operational with the development of Frama-C plugins.

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 both from others plugins (*via* the programatic interface) and from the Frama-C toplevel (*via* the command line). With this goal in mind, this chapter was written to explain step-by-step how to proceed.

The only requirement for this chapter is some shallow knowledge of Objective Caml, make and autoconf.

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.

```
$ cvs -d :ext:login@eccvs-srv1.partenaires.cea.fr/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
```

in order to generate a proper makefile and to see the plugins that are available. Now you are able to compile sources with `make`.

```
$ make -j
```

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);

[1] Character '$' (dollar) represents a shell prompt in all commands.

- `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 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 obviously possible to program such an option without the addition of a new plugin, but we use it 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[2]. 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\t%a@."
    (Pretty_utils.pp_list ~sep:"@\n\t" Format.pp_print_string)
    (Cmdline.Files.get ());
```

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 ()`. It also uses the utility function `Pretty_utils.pp_list` provided by Frama-C which helps to pretty-print a list of elements (here filenames of type `string`) separated by `sep` (here a newline followed by a tabulation).

More generally, the Frama-C modules `Cmdline` and `Pretty_utils` respectively provide information about the Frama-C command line (options and files set by the user, see Section 2.5) and helping functions for pretty-printing.

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

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

2.3 Configuration and Compilation

Here we explain how to compile the plugin `hello`. Section 3.2 and 3.3 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`[3]. They indicate how to configure `hello`, especially whether it has to be compiled or not.

```
=== configure.in ===
# ... Add the following lines after other plugins configurations.
# hello
#######
default=yes
FORCE_HELLO=no
REQUIRE_HELLO=
USE_HELLO=
AC_ARG_ENABLE(
  hello,
  [  --enable-hello support for hello plugin (default: $default) ],
  ENABLE_HELLO=$enableval; FORCE_HELLO=$enableval,
  ENABLE_HELLO=$default
)
echo "hello... $ENABLE_HELLO"
# ... Add the following line among others lines AC_SUBST
AC_SUBST(ENABLE_HELLO)
# ... Add the following line among others lines AC_MSG_NOTICE
AC_MSG_NOTICE([hello           : $ENABLE_HELLO$INFO_HELLO])
```

These lines correspond to the standard scheme for configuring a new plugin. Mostly, to add a new plugin, just copy these lines and replace `hello` by the right plugin name. Now we briefly explain these lines.

- The first line (after comments) says that the plugin is available by default (*i.e.* no special option on the command line of `configure` is required to compile `hello`).
- The second line says that it is not mandatory to compile the plugin within Frama-C.
- The third and fourth lines declare variables `REQUIRE_HELLO` and `USE_HELLO` which may be used by others plugins in order to indicate that they depend of `hello`. Indeed they are not directly used in this tutorial but some configuration scripts automatically use them. So we have to provide them.
- The call to `AC_ARG_ENABLE` first indicates the plugin name (here `hello`). Then it adds an option `--enable-hello` on the command line of `configure`. This option may be used in order to explicitly require the compilation of `hello` within Frama-C. As a counterpart,

[3] 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.

another option named `--disable-hello` also exists. This call to `AC_ARG_ENABLE` also declares and initialises the variable `ENABLE_HELLO` which says whether the plugin should be compiled or not.

- The last line of the plugin configuration echos whether the user want to compile `hello` or not.
- At the end of `configure.in`, we have to use `AC_SUBST` in order to replace in `Makefile.in` the variable `@ENABLE_HELLO@` by the value of `$ENABLE_HELLO` set by `configure.in`.
- The last line added to `configure.in` notifies the user on the way that `hello` is going to be compiled.

Now we are ready to execute

```
$ ./configure
```

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

```
configure: hello           : yes
```

in the configuration summary.

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

```
=== 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 developper 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 2.6 about plugin testing).

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

```
$ make -j
```

2.4 Connection with the Frama-C World

The plugin `hello` is now compiled but it is not strongly connected with the rest of Frama-C. In particular, our plugin should be added in the plugin database `Db` in order to be used by other plugins (see the architecture document [8] for details).

Extention 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 raise `Extlib.NotYetImplemented` if someone dereferences `run` before it is instantiated appropriately. This instantiation is done by the plugin itself: so we have to modify `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`. 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
```

2.5 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 3.8 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 of 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 we have to print the input files

of Frama-C or not). In order to implement this option, we use a predefined functor, called `False`, 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"
    ~descr:"Hello World plugin"
    [ "-hello",
      Arg.Unit Cmdline.Hello.Print.on,
      ": print input files of Frama-C" ]
```

We use the 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 in order to execute the new plugin when its option is set.

```
=== file src/toplevel/main.ml ===
(* ... *)
if Cmdline.Hello.Print.get () then !Db.Hello.run ();
(* ... *)
```

At this point, the plugin properly works: all the programming work is done.

2.6 Testing

Frama-C provides a tool, called ptests, in order to perform non-regression and unit tests. This tool is detailled in Section 3.4. 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 one of them, it also uses directives following `run.config`. Here, for the 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 just have to execute

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

in order to create oracles. So, each time one executes `ptests.byte`, differences with these 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

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

one more time, we obtain the result

```
% Diffs finished. Summary:
Run = 2
Ok  = 2 of 2
```

which indicates that everything is allright.

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.

2.7 Copyright your Work

If you want to redistribute the plugin `hello`, you have to choose a licence policy for it (compatible with the rest of Frama-C). Section 3.12 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: licency policy" of `Makefile.in` with the following line:

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

Now executing

```
$ make headers
```

adds an header on files of the plugin `hello` in order to indicate that they are under the desired licence.

Chapter 3

Frama-C-Aware Programming

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 better situation to the worse one[1]):

1. reinventing the (Frama-C) whole;
2. being not able to do some specific things;
3. introducing bugs in some of your pieces of code;
4. introducing bugs in pieces of code which use your own code;
5. breaking the kernel consistency and so potentially breaking all the Frama-C plugins.

In this chapter, we suppose that the reader well knows the software architecture of Frama-C [8] and is able to write a minimal plugin like `hello` described in chapter 2. Moreover plugin development requires to handle with autoconf, make and some advanced features of OCaml (module system, classes and objects, *etc*). Each section summarizes its own prerequisites at its beginning.

A reader can obviously read this chapter from the beginning to the end. He can yet use it as a reference manual and picks the wished information in any section in any order.

3.1 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 the architecture document [8]) nor to detail each module (that is the purpose of the source documentation generated by `make doc`). Directory containing Cil implementation is detailled in Section 3.1.1 while directory containing the Frama-C implementation itself is presented in Section 3.1.2.

Here are the main directories useful for a plugin developer:

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

- `.` is the Frama-C root directory. It contains the configuration files, makefiles and some information files (in uppercase).
- `bin` contains binaries (mainly produced by Frama-C compilation).
- `cil` contains the Cil source files. See Section 3.1.1 for details.
- `doc` is the documentation directory. It contains plugin-specific documentations, source code documentation, ACSL documentation and and some others.
- `external` contains source of external free libraries included in the Frama-C distribution.
- `headers` contains headers for Frama-C source files (see Section 3.12).
- `lib` contains compiled files (usually `.cm[iox]`) produced by Frama-C compilation. You should never add directly any file in this directory. In particular `lib/plugins` receives the compiled plugins.
- `licenses` contains licencies used by Frama-C plugins and kernel (see Section 3.12).
- `ptests` contains ptests implementation.
- `share` contains shared source files (*e.g.* Frama-C malloc in file `share/malloc.c`).
- `src` contains Frama-C implementation. See Section 3.1.2 for details.
- `tests` contains Frama-C test suites (see Section 3.4).

3.1.1 Directory `Cil`

The source files of Cil belong to five 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 analyses provided by Cil. For example, module `Cfg` provides control flow graph, module `Callgraph` provides a syntactic callgraph and module `Dataflow` provides parameterized forward/backward data flow analysis.
- `src/frontc` is the C frontend which converts C code to the corresponding Cil AST. It should not be used by a Frama-C 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.

3.1.2 Directory `Src`

The source files of Frama-C are splitted 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 splitted into two different sub-directories, one for exported type declarations and related implementations visible from `Db` (see the architecture documentation [8] and Section 3.5) and oneother for the implementation provided in `Db`.

Kernel directories are described below.

- `ai` is the abstract interpretation toolbox. In particular, module `Abstract_interp` defines useful generic lattices and module `Ival` defines some pre-instantiated arithmetic lattices.
- `gui`[2] contains the `gui` implementation part common to all plugins. See Section 3.11 for more details.
- `kernel` contains the kernel toolbox over Cil. Main kernel modules are described below.
  - `Alarms` manages alarms.
  - `Annotations` manages annotations associated with a `Cil_types.kinstr`.
  - `Ast_info` provides useful operations over the Cil AST.
  - `Boot` is the last modules linked with Frama-C (see Section 3.7).
  - `CilE` contains some useful Cil extensions.
  - `Cil_state` contains the Cil AST seen from the Frama-C world through `Cil_state.file ()`.
  - `Cmdline` contains recognized options of the Frama-C command line (see Section 3.8).
  - `Db` is the plugin database.
  - `Db_types` contains some type declarations used in `Db`. In particular, type `kernel_function` is the representation of C functions used everywhere in the Frama-C world.
  - `File` contains AST initialisers and accesses to C files corresponding to the AST.
  - `Globals` defines operations on globals splitted into operations on (global) variables, on functions and on entry points of analyses
  - `Kernel_computation` contains high-level internal state builders for most useful Frama-C datastructures (see Section 3.6.5).
  - `Kernel_datatype` contains high-level datatype builders for most useful Frama-C datatypes (see Section 3.6.4).
  - `Kernel_function` provides useful operations on `Db_types.kernel_function`.
  - `Kui` provides a high-level frontend of the Frama-C kernel.
  - `Loop` provides useful operations on loops.
  - `Printer` provides a class for pretty-print annotations.
  - `Stmts_graph` provides accessibility checks using the control flow graph.
  - `Version` provides general information on the Frama-C version.
  - `Visitor` provides Frama-C visitors subsuming the Cil ones (see Section 3.10).

[2] 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.

- `lib` contains datastructures and operations used in Frama-C. In particular, module `Extlib` is the Frama-C extension of the OCaml standard library.
- `memory_states` is the memory-state toolbox (see section 3.9).
- `misc` provides some additional useful operations.
- `project` is the project library (see Section 3.6).
- `toplevel`[2]contains the Frama-C toplevel. In particular, module `Main` defines the main Frama-C entry point (see Section 3.7) and module `Options` manages the Frama-C command line (see Section 3.8).

3.2 Configure.in

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 2.3).

First Section 3.2.1 introduces the general principle and organisation of `configure.in`. Then Section 3.2.2 explains how to configure a new simple plugin without any dependency. Next we show how to exhibit dependencies with external libraries and tools (Section 3.2.3) and with other plugins (Section 3.2.4). Finally Section 3.2.5 presents the configuration of external libraries and tools needed by a new plugin but not used anywhere else in Frama-C.

3.2.1 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 Frama-C 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 by Frama-C, or a mandatory dependency of the plugin is not available.

One 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).

So, for the plugin developer, the major job 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 below. Stars (characters '$\star$') exhibit sections which may be modified by a plugin developper.

1. *Configuration of make* : check if the version of make is correct. An useful feature is option `-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* : check if the version of OCaml is correct.
3. *Configuration of other mandatory tools/libraries*: check if all the external mandatory tools and libraries required by the Frama-C kernel are present.
4. *Configuration of other non-mandatory tools/libraries*: check what external non-mandatory tools and libraries used by the Frama-C kernel are present.
5. *Platform Configuration*: check some specific platform characteristics (operating system, specific features of gcc, *etc*) which are important to know/have got in Frama-C.
6. *Wished Frama-C Plugins**: check what Frama-C plugins the user wants to compile.
7. *Configuration of Plugin Tools/Libraries**: check what external tools and libraries only required by some plugins are available and so what plugins have to be disable (at least partially).
8. *Checking Plugin Dependencies**: check what plugins have to be disable (at least partially) because they depend of others plugins which are not available (at least partially).
9. *Makefile Creation**: create `Makefile` from `Makefile.in` including information provided by this configuration.
10. *Summary**: display on the user console the availability status of each plugin.

3.2.2 Addition of a simple plugin

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

1. add a new subsection for the new plugin to Section *Plugin whised*;
2. add a new substitution in Section *Substitutions to perform*; and
3. 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
#############
default=yes
FORCE_OCCURRENCE=no
REQUIRE_OCCURRENCE=
```

```
USE_OCCURRENCE=
AC_ARG_ENABLE(
  occurrence,
  [  ---enable-occurrence support for occurrence analysis (default: $default) ],
  ENABLE_OCCURRENCE=$enableval; FORCE_OCCURRENCE=$enableval,
  ENABLE_OCCURRENCE=$default
)
echo "occurrence... $ENABLE_OCCURRENCE"
```

These lines first set the variable `default` to `yes`, saying that the plugin is enable by default. This variable has to be set by each plugin to `yes` or `no`. Then the above lines introduce four plugin-specific variables: `FORCE_OCCURRENCE`, `REQUIRE_OCCURRENCE`, `USE_OCCURRENCE` and `ENABLE_OCCURRENCE`. These variables *must* be introduce by each plugin.

The first one indicates if the user explicitely 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 3.2.4). 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`).

Sections *Substitutions to perform* and *Summary* add respectively a new substitution in `Makefile.in` thanks to the line

```
AC_SUBST(ENABLE_OCCURRENCE)
```

and a new entry in the summary thanks to the line

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

So the value `@ENABLE_OCCURRENCE` is usable in `Makefile.in` in order to know whether the plugin has to be compiled or not (see Section 3.3) and a notification is displayed to the user which indicates this value and an optional informative message (contained in `$INFO_OCCURRENCE`).

3.2.3 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 3.2.5 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 3.3). 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 3.1 *The best place to perform such extensions is just after the addition of p which sets the value of* `ENABLE_`*p*.

Example 3.1 *Plugin* `gui` *requires Lablgtk2 [4]. 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`.

3.2.4 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 3.2.3). 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 developper 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 3.2 *Plugin* `value` *requires plugin* `from` *and may use plugin* `gui` *(for ValViewer [1]). 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.*

3.2.5 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 3.2.3 explains how to depend of an 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[3]. `configure_library`

[3]For tools, there is a script `configure_tool` which works in the same way as `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[4].

Example 3.3 *The library* `gtksourceview` *(used to have a better rendering of C sources in the GUI) can be found either as part of* `lablgtk` *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* `lablgtk` *since. 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
```

3.3 Makefile.in

In this section, we detail the use of `Makefile.in` dedicated to Frama-C compilation. First Section 3.3.1 introduces a general overview of architecture of `Makefile.in`. Next Section 3.3.2 details how to add a new plugin.

3.3.1 Overview

Here we present a general overview of `Makefile.in`. This file is splitted in several sections. Below we introduce each of them. So a plugin developer can easily search in the right section its required features (variable and rule declarations).

- **Global variables from configure** contains variable declarations from variables defined in `configure.in` (see Section 3.2). 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
  ```

[4] As plugin dependencies are checked after this check, plugins are not recursively disabled here.

- **Shell commands** defines shortcuts which should be used in the makefile.
- **Command names** defines command names displayed on the console in the non-verbose mode.
- **Global plugin variables** declares some plugin-specific variables used throughout the makefile.
- **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 3.3.2).
- **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.
- **External libraries to compile** provides variables and rules for external libraries required by Frama-C. Each library is in a specific sub-section.
- **Internal miscellaneous libraries** provides variables and rules for Frama-C internal libraries (`Cil` and `Project`), each described in a specific sub-section.
- **Kernel** provides variales and rules for the Frama-C kernel. Each part is described in specific sub-sections.
- After Section "Kernel", there are several sections corresponding to **plugins** (see Section 3.3.2).
- After plugin sections, there are sections corresponding to different Frama-C frontends (in particular, Sections **toplevel**, **gui** and **obfuscator**).
- **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)
- **Tests** provides rules to execute tests (see Section 3.4).
- **Emacs tags** provides rules which generate emacs tags (useful for a quick search of OCaml definitions).
- **Documentation** provides rules generating Frama-C source documentation (see Section 3.13).
- **Distribution** provides rules which install the Frama-C distribution.
- **File headers: license policy** provides variables and rules to manage the Frama-C license policy (see Section 3.12).
- **Makefile rebuilting** provides rules in order to automatically rebuild `Makefile` and `configure` when required.
- **Cleaning** provides rules in order to remove files generated by makefile rules.
- **Depend** provides rules which compute Frama-C source dependencies.
- **Ptests** provides rules in order to build ptests (see Section 3.4).

3.3.2 Addition of a new Plugin

In order to add a new plugin, you have to include `Makefile.plugin` in `Makefile.in` among other plugins. `Makefile.plugin` is a generic makefile dedicated to plugin compilation. A plugin developer can set some variables before including it in order to control its behaviour. Below is the list of variables which can be set.

- `PLUGIN_NAME`: Module name of the plugin. So must be capitalised (as each OCaml module name).
- `PLUGIN_ENABLE`: Set it 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).
- `PLUGIN_DIR`: Directory containing plugin source files (usually `src/`*plugin* where *plugin* is the plugin name).
- `PLUGIN_CMO`: Plugin `.cmo` files. Do not write its file path (which is `$(PLUGIN_DIR)`) nor its file extension (which is `.cmo`): they are automatically added.
- `PLUGIN_CMI`: Plugin `.cmi` files for which there is no corresponding `.cmo`. Do not write its file path (which is `$(PLUGIN_DIR)`) nor its file extension (which is `.cmi`): they are automatically added.
- `PLUGIN_TYPES_CMO`: Plugin `.cmo` files which do not belong to `$(PLUGIN_DIR)` (they usually belong to `src/`*plugin*`_types` where *plugin* is the plugin name, see Section 3.5). Do not write file extension (which is `.cmo`): it is automatically added.
- `PLUGIN_GUI_CMO`: Plugin `.cmo` files which have to be linked with the GUI. Must be a subset of `$(PLUGIN_CMO)`.
- `PLUGIN_HAS_MLI`: Set it to `yes` if plugin *plugin* gets a file `.mli` (which must be capilalised: `Plugin.mli`) 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 the architecture document [8]).
- `PLUGIN_BFLAGS`: Specific plugin flags for `ocamlc`.
- `PLUGIN_OFLAGS`: Specific plugin flags for `ocamlopt`.
- `PLUGIN_DEPFLAGS`: Specific plugin flags for `ocamldep`.
- `PLUGIN_GENERATED`: Files which must be generated before computing plugin dependencies.
- `PLUGIN_DOCFLAGS`: Specific plugin flags for `ocamldoc`.
- `PLUGIN_UNDOC`: Source files for which no documentation is provided. Do not write its file path (which is `$(PLUGIN_DIR)`).
- `PLUGIN_TYPES_TODOC`: Additional source files to document with the plugin (they usually belong to `src/`*plugin*`_types` where *plugin* is the plugin name, see Section 3.5).
- `PLUGIN_INTRO`: Text file to add at the end of 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`.
- `PLUGIN_NO_TEST`: Set it to `yes` if there is no specific test directory for the plugin.

- `PLUGIN_TESTS_DIRS`: Directories containing plugin tests (default value is `tests/$(notdir $(PLUGIN_DIR))`).
- `PLUGIN_TESTS_LIB` Specific `.cmo` files used by plugin tests. Do not write its file path (which is `$(PLUGIN_TESTS_DIRS)`) nor its file extension (which is `.cmo`).
- `PLUGIN_DEPENDS`: Other plugins which must be compiled before the considered plugin.

For setting variables, you must use `:=` and not `=`.

Example 3.4 *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
```

Moreover the above variables must not be used anywhere else in `Makefile.in`.

For each plugin *plugin*, you can yet use the following variables once `Makefile.plugin` has been included for it.

- *plugin*`_DIR`
- *plugin*`_CMO`
- *plugin*`_CMX`
- *plugin*`_CMI`
- *plugin*`_TYPES_CMO`
- *plugin*`_TYPES_CMX`
- *plugin*`_TYPES_TODOC`
- *plugin*`_BFLAGS`
- *plugin*`_OFLAGS`
- *plugin*`_DEPFLAGS`
- *plugin*`_DOCFLAGS`
- *plugin*`_GENERATED`
- *plugin*`_TESTS_DIRS`

- *plugin*_TESTS_LIB

Semantics of each variable *plugin_variable* is the same as semantics of PLUGIN_*variable*.

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.

3.4 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 3.4.1 shows how to use ptests. Next Section 3.4.2 explains how to configure tests through directives.

3.4.1 Use of 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 3.5 *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
```

Below we detail options of ptests.

- `-add-options opts`: Add additional options `opts` to be passed to the toplevels that will be launched.

- `-byte`: Use bytecode toplevel `bin/toplevel.byte`.
- `-diff cmd`: Use command `cmd` in order to compare results and oracles (default is `diff -u`).
- `-examine`: Only examine the current results that are different from oracles (do not run tests).
- `-j n`: Use non-negative integer `n` for level of parallelism (default is `4`).
- `-opt`: Use native toplevel `bin/toplevel.opt`. This option is set by default.
- `-run`: Delete the current results, then run the tests and finally examine the results that are different from the oracles. This option is set by default.
- `-show`: Run the tests and show their results. Also automatically set option `-byte`.
- `-update`: Take the current results as oracles. Do not run tests.
- `-v`: increase verbosity (up to twice) (default is `0`).

Example 3.6 *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.*

3.4.2 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:

- `COMMENT`: Comment in the configuration.
- `CMD`: Program name to run (default is `./bin/toplevel.opt` and default directory is the parent one of `tests`).
- `DONTRUN`: Do no execute this test (or test suite).
- `EXECNOW`: Run a command before running the test itself. The syntax is the following.

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

  Files after `LOG` are log files generated by command `cmd` and compared from oracles, whereas files after `BIN` are binary files also generated by `cmd` but not compared from oracles. Full access path to these files have to be specified only in `cmd`.
- `FILEREG`: Should be used only in a configuration file `test_config`. It is a regular expression indicating which files have to be tested (default is `.*\.\(c|i\)`).
- `FILTER`: Set the command name used to filter results before their comparison from oracles (default is no filter).
- `GCC`: Unused (only present for compatibility reasons).
- `OPT`: Options to be used by command specified through `CMD` (default is `-val -out -input -deps`). If there are several directives `OPT` in the same configuration, they correspond to different test cases.

Example 3.7 *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. So

```
$ ./bin/ptests.byte tests/sparecode/calls.c
```

executes two test cases and displays the following output.

```
% Dispatch finished, waiting for workers to complete
% Comparisons finished, waiting for diffs to complete
% Diffs finished. Summary:
Run = 2
Ok  = 4 of 4
```

3.5 Exporting Datatypes

If a plugin has to export some datatypes usable by other plugins, they have to be visible by `Db`. So they cannot be declared in the plugin implementation itself like any other plugin declaration because postponed declarations are not possible for types.

The solution is to put these datatype declarations in files linked before `Db` and 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 3.2 *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 3.3.2).

Example 3.8 *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 sad 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 3.3.2).

3.6 Project Management System

In Frama-C, a key notion detailled in this Section is *project*. Section 3.6.1 first introduces its general principle. Next Section 3.6.2 explains how to simply use them. Then Section 3.6.3 introduces the principle of so-called *internal states* registration which is detailled in Sections 3.6.4, 3.6.5 and 3.6.6. The first one is dedicated to so-called *datatypes*. The second one is dedicated to the internal states themself. The third one is dedicated to low-level registration. Finally Section 3.6.7 shows how to shrewdly handle projects and internal states.

[5] A direct consequence is that you cannot use the whole Frama-C functionalities inside this code.

3.6.1 General 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 3.8) and tables containing results of analyses (Frama-C extensively uses memoisation [6, 7] in order to avoid re-computation of analyses). The set of all these data is called a *project*.

Several ASTs can exist at the same time in Frama-C and so several projects as well: the number of ASTs is exactly the same than the number of projects. Besides a data has one value by AST: usually the table containing results of one particular analysis is different but always exists for two different ASTs.

The set of all the projects stands for *the internal state of Frama-C* : that is all the ASTs defined in Frama-C and, for each of them, the corresponding value of all the attached data. So it contains all the important data of Frama-C. Hence it is the only value saved on the disk and restored by loading.

A related notion is *internal state* of a data d. That is the set of all the values of d: so, 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 3.1.

Projects / **Internal states**	Project p_1	Project p_2 ...	Project p_n
AST a	value of a in p_1	value of a in p_2 ...	value of a in p_n
data d_1	value of d_1 in p_1	value of d_1 in p_2 ...	value of d_1 in p_n
...	...	... ...	...
data d_m	value of d_m in p_1	value of d_m in p_2 ...	value of d_m in p_n

Figure 3.1: Matricial Representation of the Frama-C Internal State

3.6.2 Use of project

Actually projects are mostly unshown in Frama-C: there is a default project `Project.current ()` and a default AST `Cil_state.file ()` which all operations are performed on. But sometimes a plugin developer have to explicitely use them. That is when the AST is modified (usually through the use of a copy visitor, see Section 3.10) 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`. Each project has got type `Project.t`. Among other operations described in the interface of `Project`, an important function is `Project.set_current` which sets the current project and hence switchs the Frama-C context: after its application, all operations are implicitely performed on the new current project.

Example 3.9 *Suppose that you do not perform the value analysis and save the current project into file `foo.sav` in a previous Frama-C session*[6] *thanks to the following instruction.*

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

```
Project.save "foo.sav"
```

So, in a new Frama-C session, executing the following lines of code (assuming the value analysis is not computed)

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

displays

```
false
true
false
```

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

An alternative to the use of `Project.set_current` is the use of `Project.on` which applies an operation on a given project without changing the current project (*i.e.* locally switch the current project in order to apply the given operation and, after, restore the initial context).

Example 3.10 *The following code is equivalent to the one given in Example 3.9.*

```
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
```

So it displays

```
false
true
false
```

3.6.3 Internal State Registration: Principle

If a data should be part of the internal state of Frama-C (*e.g.* it should be saved on the disk), you must register it as an internal state (aka a *computation* because it is often related to memoisation).

For this purpose, functor `Project.Computation.Register` is provided. Actually it is quite a low-level functor and higher-level functors inside modules `Computation` and `Kernel_computation` register internal states in a simpler way by wraping the low-level functor: there is no direct use of the low-level functor in the Frama-C kernel. 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 index by statements of the AST)[7].

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 aware of projects. As for computations, module `Datatype` (resp. `Kernel_datatype`) provides pre-defined datatypes and datatypes-builder for OCaml (resp. Frama-C) datatypes like hashtables (resp. hashtables index by statements of the AST)[8].

3.6.4 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 mostly cases, applying this functor is actually not required. We explain here the three different possible situations.

Simple registration If the datatype to register is not hash-consed[9] or does not contain hash-consed ones (*i.e.* it is not itself hash-consed or composed of `Cil_types.fundec`, or any Frama-C abstract interpretation type), the easiest way of registering a new datatype d is to apply one of functors `Persistent` or `Imperative` of module `Project.Datatype`, depending on the nature of d (whether it is persistent). The only difference between both functors is that you have to provide a copy function for imperative (*i.e.* mutable) datatypes. This copy function is only used by `Project.copy`.

Example 3.11 *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 `t` *containing a mutable field* `a`*, just do*

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

[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 [3, 5, 2].

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.

Example 3.12 *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))
```

Sad cases 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 isomorph to `string`. The only difference is name unicity: if the same name is used for two different registrations, an exception is occuring at the module iniatialisation time of Frama-C (*i.e.* at runtime, but before mostly anything else, see Section 3.7).
- 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 3.13 *Plugin value registers a complex tuple in the following way (see file `src/value/eval.ml`).*

```
Project.Datatype.Register
  (struct
     module V_Offsetmap_option = Datatype.Option(V_Offsetmap.Datatype)
     type t =
       Relations_type.Model.t
       * (V_Offsetmap_option.t * Relations_type.Model.t)
       * Locations.Zone.t (* in *)
       * Locations.Zone.t (* out *)
     let copy _ = assert false (* TODO: deep copy *)
     let rehash (generic_state, (result, result_state), ins, outs) =
       Relations_type.Model.Datatype.rehash generic_state,
       (V_Offsetmap_option.rehash result,
        Relations_type.Model.Datatype.rehash result_state),
```

```
       Locations.Zone.Datatype.rehash ins,
       Locations.Zone.Datatype.rehash outs
      include Datatype.Nop
      let name = Project.Datatype.Name.make "Mem_Exec_tuple"
      let dependencies =
        [ Relations_type.Model.Datatype.self;
          Cvalue_type.V.Datatype.self;
          V_Offsetmap_option.self;
          Locations.Zone.Datatype.self ]
  end)
```

3.6.5 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 posponed in Section 3.6.6 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 stock results. 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 s = ...
let memoise s =
  try StmtHashtbl.find state s
  with Not_found -> StmtHashtbl.add state s (compute_info s)
let run () = ... !Db.Value.compute (); ... memoise s ...
```

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 s = ...
let memoise = State.memo compute_info
let run () = ... !Db.Value.compute (); ... memoise s ...
```

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`. 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 3.6.4). 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 3.6.4). 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. So operations on states implementing hashtable does not take an hashtable in argument. Here, a predefined memo function is used in order to memoise the computation of `compute_info`. This function implicetely operates on an hashtable hiden in the internal representation of `State`.

TODO: Computation.apply_once (?)

Postponed dependencies A plugin p may want to export the kind of its internal state (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 3.5). 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 3.7). 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 (also see Section 3.7). So you have to postpone the addition of this dependency (usually by using function `Options.register_plugin_init`, see Section 3.8).

Example 3.14 *Plugin from postpones its internal state in the following way.*

```
=== src/kernel/db.mli ===
module From = struct
  ...
  val self: Project.Computation.t ref
end
=== src/kernel/db.ml ===
module From = struct
  ...
  val self = ref Project.Computation.dummy (* postponed *)
end
=== 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. *)
```

Besides plugin pdg uses from for computing its own internal state. So it declares this dependency as follow.

```
=== 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)
```

3.6.6 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 3.6.5).

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* switchs 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]

$$\texttt{create () != create ()} \tag{3.1}$$

$$\forall \texttt{p of type t}, \texttt{create () = (clear p; set p; get ())} \tag{3.2}$$

$$\forall \texttt{p of type t}, \texttt{set p; p == get ()} \tag{3.3}$$

$$\forall \texttt{p1, p2 of type t}, \texttt{set p1; let p = get () in set p2; p != get ()} \tag{3.4}$$

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 independance 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 undirection in order to be safely (and efficiently) implemented.

Example 3.15 *To illustrate this, we show how functor* `Computation.Ref` *(registering a state corresponding to a reference) is implemented.*

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

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

```
  include Project.Computation.Register
    (Datatype.Ref(Data))
    (struct
      type t = data ref (* we register a reference on the given type *)
      let create = create
      let clear tbl = tbl := Data.default
      let get () = !state
      let set x = state := x
    end)
   (Info)
```

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

```
  let set v = !state := v
  let get () = !(!state)
  let clear () = !state := Data.default
end
```

3.6.7 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 what internal states the operation applies on:

- If `only` is specified, the operation is *only* applied on the selected states.

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

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

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

Example 3.16 *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 wants to clear all the states which depend of the value analysis.

In some cases, using selection may be quite dangerous because the Frama-C state may become lost and inconsistent. So use it carefully.

Example 3.17 *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 3.8).*

```
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.

3.7 Initialisation Steps

In a standard way, Frama-C modules are initialiased in the link order. Mostly the link order remains 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 3.6.5) 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 3.8). 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]
(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 3.6.5).
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 of 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).

3.8 Command Line Options

Values associated with command line options are stored in module `Cmdline` while command line options themself are registered through module `Options`. First Section 3.8.1 introduces how to store new option values. Second Section 3.8.2 presents how to register new options.

3.8.1 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 3.6.5). 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.

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`).

[11] If the graphical user interface is compiled.

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 3.6.5).

Example 3.18 *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 3.3 *Options of a same plugin `plugin` should belong to a same module `PluginOptions` inside `Cmdline`.*

3.8.2 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 3.7). 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 3.7).

Example 3.19 *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_cmdline () 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 3.7) 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.*

3.9 Memory States

Locations_Bytes.t = Association between varids and offsets in byte.

Locations_Bits.t = Association between varids and offsets in bits.

Zone.t = Association between varids and ranges of bits.

location = A Location_Bits.t and a size in bits.

Lmap = roughly map indexed by location

Lmap_bitwise = roughly map indexed Zone.t

In both cases, handle overlaps

3.10 Visitors

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 projects and the internal state of Frama-C (see Section 3.6). Hence, there exists another visitor, `Visitor.generic_frama_c_visitor`, which deals with that 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 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 3.10.5, which the reader already familiar with Cil is invited to read carefully.

3.10.1 Entry points

Cil offers a certain number of entry points for the visitor. They are functions called `Cil.visitCil`*AstType* where *astType* is a type of 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.

3.10.2 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`.

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

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`, `vfile_aux` and `vglob_aux` instead.

3.10.3 Action performed

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

- `SkipChildren` the visitor do not visit the children;
- `ChangeTo v` the old node is replaced by `v` and the visit stops;
- `DoChildren` the visit goes on with the children; this is the default behavior;
- `DoChildrenPost(v,f)` the old node is replaced by `v`, the visit goes on with the children of `v`, and when it is finished, `f` is applied to the result.

3.10.4 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 explicitely 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.

3.10.5 Differences between Cil and Frama-C visitors

The Frama-C visitor takes an additional argument, which is the project in which the transformed AST should be put in. Note that an in-place visitor (see previous 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.

Moreover, as said in Section 3.10.2, `vstmt`, `vfile`, and `vglob` should not be redefined. Use `vstmt_aux`, `vfile_aux` and `vglob_aux` instead. For `vstmt_aux` and `vglob_aux`, 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 3.10.3), it is assumed that it has taken care of updating the table 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.

3.10.6 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
```

3.11 GUI Extension

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 3.3.2).

Mainly that's all! The gui implementation uses Lablgtk2 [4]: so you can use any Lablgtk2-compatible code in your gui extension.

Example 3.20 *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 (* comming with the Frama-C GUI *)
open Gtk_helper    (* comming 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

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

*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 occurences of the selected variables and highlights them in yellow.

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 hightlighting (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.

3.12 License Policy

3.13 Documentation

Bibliography

[1] CEA LIST, Laboratoire pour la Sûreté du Logiciel. *Documentation de l'outil d'analyse ValViewer*, February 2008. In French.

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

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

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

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

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

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

[8] Julien Signoles. *Frama-C Software Architecture*, February 2008.