frama C

Software Analyzers

User Manual

Frama-C User Manual

Release 21.0 (Scandium)

Loïc Correnson, Pascal Cuoq, Florent Kirchner, André Maroneze, Virgile Prevosto, Armand Puccetti, Julien Signoles and Boris Yakobowski

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

This work has been supported by the ANR project CAT (ANR-05-RNTL-00301) and by the ANR project U3CAT (08-SEGI-021-01).

Contents

Foreword

This is the user manual of Frama-C[1]. The content of this document corresponds to the version 21.0 (Scandium)(June 10, 2020) of Frama-C. However the development of Frama-C is still ongoing: features described here may still evolve in the future.

Acknowledgements

We gratefully thank all the people who contributed to this document: Patrick Baudin, Mickaël Delahaye, Philippe Hermann, Benjamin Monate and Dillon Pariente.

[1] http://frama-c.com

Chapter 1

Introduction

This is Frama-C's user manual. Frama-C is an open-source platform dedicated to the analysis of source code written in the C programming language. The Frama-C platform gathers several analysis techniques into a single collaborative framework.

This manual gives an overview of Frama-C for newcomers, and serves as a reference for expert users. It only describes those platform features that are common to all analyzers. Thus it *does not cover* the use of the analyzers provided in the Frama-C distribution (Value Analysis, Slicing, ...). Each of these analyses has its own specific documentation [4, 7, 9, 13]. Furthermore, research papers [6, 10] give a synthetic view of the platform, its main and composite analyses, and some of its industrial achievements, while the development of new analyzers is described in the Plug-in Development Guide [12].

1.1 About this document

Appendix A references all the changes made to this document between successive Frama-C releases.

In the index, page numbers written in bold italics (e.g. ***1***) reference the defining sections for the corresponding entries while other numbers (e.g. 1) are less important references.

The most important paragraphs are displayed inside gray boxes like this one. A plug-in developer **must** follow them very carefully.

1.2 Outline

The remainder of this manual is organized in several chapters.

Chapter 2 provides a general overview of the platform.

Chapter 3 describes the basic elements for starting the tool, in terms of installation and commands.

Chapter 4 explains the basics of plug-in categories, installation, and usage.

Chapter 5 presents the options of the source code pre-processor.

Chapter 6 gives some general options for parameterizing analyzers.

Chapter 7 touches on the topic of code properties, and their validation by the platform.

Chapter 8 introduces the general services offered by the platform.

Chapter 9 gives a detailed description of the graphical user interface of Frama-C.

Chapter 10 describes the `Report` plug-in, used for textual consolidation and export of warnings, errors and properties.

Chapter 11 presents the `Variadic` plug-in, used to help other plug-ins handle code containing variadic functions, such as `printf` and `scanf`.

Chapter 12 details several scripts used to help setup and run analyses on large code bases.

Chapter 13 explains how to report errors *via* Frama-C's public Gitlab repository.

Chapter 2

Overview

2.1 What is Frama-C?

Frama-C is a platform dedicated to the analysis of source code written in C. The Frama-C platform gathers several analysis techniques into a single collaborative extensible framework. The collaborative approach of Frama-C allows analyzers to build upon the results already computed by other analyzers in the framework. Thanks to this approach, Frama-C can provide a number of sophisticated tools such as a slicer [3], and a dependency analyzer [7, Chap. 6].

2.2 Frama-C as a Static Analysis Tool

Static analysis of source code is the science of computing synthetic information about the source code without executing it.

To most programmers, static analysis means measuring the source code with respect to various metrics (examples are the number of comments per line of code and the depth of nested control structures). This kind of syntactic analysis can be implemented in Frama-C but it is not the focus of the project.

Others may be familiar with heuristic bug-finding tools. These tools take more of an in-depth look at the source code and try to pinpoint dangerous constructions and likely bugs (locations in the code where an error might happen at run-time). These heuristic tools do not find all such bugs, and sometimes they alert the user for constructions which are in fact not bugs.

Frama-C is closer to these heuristic tools than it is to software metrics tools, but it has two important differences with them: it aims at being correct, that is, never to remain silent for a location in the source code where an error can happen at run-time. And it allows its user to manipulate *functional specifications*, and to *prove* that the source code satisfies these specifications.

Frama-C is not the only correct static analyzer out there, but analyzers of the *correct* family are less widely known and used. Software metrics tools do not guarantee anything about the behavior of the program, only about the way it is written. Heuristic bug-finding tools can be very useful, but because they do not find all bugs, they can not be used to prove the absence of bugs in a program. Frama-C on the other hand can guarantee that there are no bugs in a program ("no bugs" meaning either no possibility of a run-time error, or even no deviation from the functional specification the program is supposed to adhere to). This

of course requires more work from the user than heuristic bug-finding tools usually do, but some of the analyses provided by Frama-C require comparatively little intervention from the user, and the collaborative approach proposed in Frama-C allows the user to get results about complex semantic properties.

2.2.1 Frama-C as a Lightweight Semantic-Extractor Tool

Frama-C analyzers, by offering the possibility to extract semantic information from C code, can help better understand a program source.

The C language has been in use for a long time, and numerous programs today make use of C routines. This ubiquity is due to historical reasons, and to the fact that C is well adapted for a significant number of applications (*e.g.* embedded code). However, the C language exposes many notoriously awkward constructs. Many Frama-C plug-ins are able to reveal what the analyzed C code actually does. Equipped with Frama-C, you can for instance:

- observe sets of possible values for the variables of the program at each point of the execution;
- slice the original program into simplified ones;
- navigate the dataflow of the program, from definition to use or from use to definition.

2.2.2 Frama-C for Formal Verification of Critical Software

Frama-C can verify that an implementation complies with a related set of formal specifications.

Specifications are written in a dedicated language, ACSL (*ANSI/ISO C Specification Language*) [2]. The specifications can be partial, concentrating on one aspect of the analyzed program at a time.

The most structured sections of your existing design documents can also be considered as formal specifications. For instance, the list of global variables that a function is supposed to read or write to is a formal specification. Frama-C can compute this information automatically from the source code of the function, allowing you to verify that the code satisfies this part of the design document, faster and with less risks than by code review.

2.3 Frama-C as a Tool for C programs

The C source code analyzed by Frama-C is assumed to follow the C99 ISO standard. C comments may contain ACSL annotations [2] used as specifications to be interpreted by Frama-C. The subset of ACSL currently interpreted in Frama-C is described in [1].

Each analyzer may define the subsets of C and ACSL that it understands, as well as introduce specific limitations and hypotheses. Please refer to each plug-in's documentation.

2.4 Frama-C as an Extensible Platform

Frama-C is organized into a modular architecture (comparable to that of the Gimp or Eclipse): each analyzer comes in the form of a *plug-in* and is connected to the platform itself, or *kernel*, which provides common functionalities and collaborative data structures.

Several ready-to-use analyses are included in the Frama-C distribution. This manual covers the set of features common to all plug-ins, plus some plug-ins which are used by several others, such as the graphical user interface (GUI) and reporting tools (Report). It does not cover use of the plug-ins that come in the Frama-C distribution: value analysis (Eva), functional dependencies (From), functional verification (WP), program slicing (Slicing), *etc.* Each of these analyses has its own specific documentation [7, 4, 3].

Additional plug-ins can be provided by third-party developers and installed separately from the kernel. Frama-C is thus not limited to the set of analyses initially installed. For instance, it may be extended with the E-ACSL plug-in [11, 8] which instruments the program in order to check annotations at runtime. In this way, Frama-C is not restricted to static analysis of source code, but also provides dynamic analysis.

2.5 Frama-C as a Collaborative Platform

Frama-C's analyzers collaborate with each other. Each plug-in may interact with other plug-ins of his choosing. The kernel centralizes information and conducts the analysis. This makes for robustness in the development of Frama-C while allowing a wide functionality spectrum. For instance, the Slicing plug-in uses the results of the Eva (value analysis) plug-in and of the From (functional dependencies) plug-in.

Analyzers may also exchange information through ACSL annotations [2]. A plug-in that needs to make an assumption about the behavior of the program may express this assumption as an ACSL property. Because ACSL is the *lingua franca* of all plug-ins, another plug-in can later be used to establish the property.

With Frama-C, it is possible to take advantage of the complementarity of existing analysis approaches. It is possible to apply the most sophisticated techniques only on those parts of the analyzed program that require them. The low-level constructs can for instance effectively be hidden from them by high-level specifications, verified by other, adapted plug-ins. Note that the sound collaboration of plug-ins on different parts of a same program that require different modelizations of C is a work in progress. At this time, a safe restriction for using plug-in collaboration is to limit the analyzed program and annotations to those C and ACSL constructs that are understood by all involved plug-ins.

2.6 Frama-C as a Development Platform

Frama-C may be used for developing new analyses. The collaborative and extensible approach of Frama-C allows powerful plug-ins to be written with relatively little effort.

There are a number of reasons for a user of Frama-C to be also interested in writing their own plug-in:

- a custom plug-in can emit very specific queries for the existing plug-ins, and in this way obtain information which is not easily available through the normal user interface;
- a custom plug-in has more flexibility for finely tuning the behavior of the existing analyses;
- some analyses may offer specific opportunities for extension.

If you are a researcher in the field of program analysis, using Frama-C as a testbed for your ideas is a choice to consider. You may benefit from the ready-made parser for C programs with ACSL annotations. The results of existing analyses may simplify the problems that are orthogonal to those you want to consider (in particular, the value analysis provides sets of possible targets of every pointer in the analyzed C program). And lastly, being available as a Frama-C plug-in increases your work's visibility among existing industrial users of Frama-C. The development of new plug-ins is described in the Plug-in Development Guide [12].

2.7 Frama-C as an Educational Platform

Frama-C is being used as part of courses on formal verification, program specification, testing, static analysis, and abstract interpretation, with audiences ranging from Master's students to active professionals, in institutions world-wide. Frama-C is part of the curriculum at several universities in France, England, Germany, Portugal, Russia and in the US; at schools such as Ecole Polytechnique, ENSIIE, ENSMA, or ENSI Bourges; and as part of continuing education units at CNAM, or at Fraunhofer FOKUS.

If you are a teacher in the extended field of software safety, using Frama-C as a support for your course and lab work is a choice to consider. You may benefit from a clean, focused interface, a choice of techniques to illustrate, and a in-tool pedagogical presentation of their abstract values at all program points. A number of course materials are also available on the web, or upon simple inquiry to the Frama-C team.

Chapter 3

Getting Started

This chapter describes *how* to install Frama-C and *what* this installation provides.

3.1 Installation

The Frama-C platform is distributed as source code. Binaries are also available for popular architectures. All distributions include the Frama-C kernel and a base set of open-source plug-ins.

The recommended way to install Frama-C is by using the opam [1] package manager to install the `frama-c` package, which should always point to the latest release compatible with the configured OCaml compiler. opam is able to handle the dependencies required by the Frama-C kernel.

Frama-C can also be installed via pre-compiled binaries, which include many of the required libraries and other dependencies, although there is a delay between each new Frama-C release and the availability of a binary package for the considered platform.

Finally, Frama-C can be compiled and installed from the source distribution, as long as its dependencies have already been installed.

The dependencies of the Frama-C kernel are as follows. Each plug-in may define its own set of additional dependencies. Instructions for installing Frama-C from source may be found in the file `INSTALL.md` of the source distribution.

A C pre-processor is required for *using* Frama-C on C files. By default, Frama-C tries to use `gcc -C -E -I.` as pre-processing command, but this command can be customized (see Section 5.2). If you do not have any C pre-processor, you can only run Frama-C on already pre-processed `.i` files.

A C compiler is required to compile the Frama-C kernel.

A Unix-like compilation environment is mandatory and shall have at least the tool `GNU make`[2] version 3.81 or higher, as well as various POSIX commands, libraries and header files.

[1] http://opam.ocaml.org
[2] http://www.gnu.org/software/make

The OCaml compiler [3] is required both for compiling Frama-C from source *and* for compiling additional plug-ins. Version greater or equal than 4.05.0 of the compiler must be used.

Gtk-related packages: GTK+[4] version 2.4 or higher, GtkSourceView[5] version 2.x, GnomeCanvas[6] version 2.x as well as LablGtk[7] version 2.18.2 or higher are required for building the Graphical User Interface (GUI) of Frama-C.

OcamlGraph package: Frama-C needs the OcamlGraph[8] package, version 1.8.8.

Zarith package: Frama-C requires the Zarith[9] package, used for large integer computations.

3.2 One Framework, Several Executables

Frama-C installs some executables[10], namely:

- `frama-c`: natively-compiled batch version;
- `frama-c.byte`: bytecode batch version;
- `frama-c-gui`: natively-compiled interactive version;
- `frama-c-gui.byte`: bytecode interactive version;
- `frama-c-config`: auxiliary batch version for quickly retrieving configuration information (e.g. installation path);
- `frama-c-script`: contains several utilities related to source preparation, results visualization and analysis automation. Run it without arguments to obtain more details.

The differences between these versions are described below.

***native-compiled* vs *bytecode*:** native executables contain machine code, while bytecode executables contain machine-independent instructions which are run by a bytecode interpreter.

The native-compiled version is usually ten times faster than the bytecode one. The bytecode is sometimes able to provide better debugging information. Use the native-compiled version unless you have a specific reason to use the bytecode one.

***batch* vs *interactive*:** The interactive version is a GUI that can be used to select the set of files to analyze, specify options, launch analyses, browse the code and observe analysis results at one's convenience (see Chapter 9 for details).

With the batch version, all settings and actions must be provided on the command-line. This is not possible for all plug-ins, nor is it always easy for beginners. Modulo the

[3] http://ocaml.org
[4] http://www.gtk.org
[5] http://projects.gnome.org/gtksourceview
[6] http://library.gnome.org/devel/libgnomecanvas
[7] http://lablgtk.forge.ocamlcore.org
[8] http://ocamlgraph.lri.fr
[9] http://forge.ocamlcore.org/projects/zarith
[10] On Windows OS, the usual extension `.exe` is added to each file name.

limited user interactions, the batch version allows the same analyses as the interactive version[11]. A batch analysis session consists in launching Frama-C in a terminal. Results are printed on the standard output.

The task of analyzing some C code being iterative and error-prone, Frama-C provides functionalities to set up an analysis project, observe preliminary results, and progress until a complete and satisfactory analysis of the desired code is obtained.

3.3 Frama-C Command Line and General Options

3.3.1 Getting Help

The option `-help` or `-h` or `--help` gives a very short summary of Frama-C's command line, with its version and the main help entry points presented below.

The other options of the Frama-C kernel, *i.e.* those which are not specific to any plug-in, can be printed out through either the option `-kernel-help` or `-kernel-h`.

The list of all installed plug-ins can be obtained via `-plugins`, while `-version` prints the Frama-C version only (useful for installation scripts).

The options of the installed plug-ins are displayed by using either the option `-<plug-in shortname>-help` or `-<plug-in shortname>-h`.

3.3.2 Frama-C Configuration

The complete configuration of Frama-C can be obtained with various options, all documented with `-kernel-h`:

`-print-version`	version number only
`-print-share-path`	directory for shared resources
`-print-lib-path`	directory for the Frama-C kernel
`-print-plugin-path`	directory for installed plug-ins
`-print-config`	summary of the Frama-C configuration
`-plugins`	list of installed plug-ins

There are many aliases for these options, but for backward compatibility purposes only. Those listed above should be used for scripting.

3.3.3 Options Outline

The batch and interactive versions of Frama-C obey a number of command-line options. Any option that exists in these two modes has the same meaning in both. For instance, the batch version can be made to launch the value analysis on the `foo.c` file with the command `frama-c -eva foo.c`. Although the GUI allows to select files and to launch the value analysis interactively, the command `frama-c-gui -eva foo.c` can be used to launch the value analysis on the file `foo.c` and immediately start displaying the results in the GUI.

Any option requiring an argument may use the following format:

```
-option_name value
```

[11] For a single analysis project. Multiple projects can only be handled in the interactive version or programmatically. See Section 8.1

Parameterless Options Most parameterless options have an opposite option, often written by prefixing the option name with `no-`. For instance, the option `-unicode` for using the Unicode character set in messages has an opposite option for limiting the messages to ASCII. Plug-in options with a name of the form `-<plug-in name>-<option name>` have their opposite option named `-<plug-in name>-no-<option name>`. For instance, the opposite of option `-wp-print` is `-wp-no-print`. When prefixing an option name by `-no` is meaningless, the opposite option is usually renamed. For instance, the opposite option of `-journal-enable` is `-journal-disable`. Use the options `-kernel-help` and `-<plug-in name>-help` to get the opposite option name of each parameterless option.

String Options If the option's argument is a string (that is, neither an integer nor a float, *etc*), the following format is also possible:

```
-option_name=value
```

This format **must be used** when `value` starts with a minus sign.

Set Options Some options (e.g. option `-cpp-extra-args`) accept a set of comma-separated values as argument. Each value may be prefix by `+` (resp. `-`) to indicate that this value must be added to (resp. deleted from) the set. When neither is specified, `+` is added by default.

As for string options, the extended format is also possible:

```
-option_name=values
```

This format **must be used** if your argument contains a minus sign.

For instance, you can ask the C preprocessor to search for header files in directory `src` by setting:

```
-cpp-extra-args="-I src"
```

Categories are specific values which describe a subset of values. Their names begin with an `@`. Available categories are option-dependent, but most set options accept the category `@all` which defines the set of all acceptable values.

For instance, you can ask the Eva plug-in to use the ACSL specification of each function but `main` instead of their definitions by setting:

```
-eva-use-spec="@all, -main"
```

If the first character of a set value is either `+`, `-`, `@` or \, it **must be escaped** with a \.

Map Options Map options are set options whose values are of the form `key:value`. For instance, you can override the default Eva's *slevel* [7] for functions `f` and `g` by setting:

```
-slevel-function="f:16, g:42"
```

3.3.4 Autocompletion for Options

The `autocomplete_frama-c` file in the directory of shared resources (see `-print-share-path` option above) contains a bash autocompletion script for Frama-C's options. In order to take advantage of it, several possibilities exist.

- In order to enable system-wide completion, the file can be copied into the directory `/etc/bash_completion.d/` where bash search for completion scripts by default

- If you only want to add completion for yourself, you can append the content of the file to `~/.bash_completion`
- You can `source` the file, e.g. from your `.bashrc` with the following command:

```
source $(frama-c -print-share-path)/.autocomplete_frama-c || true
```

There is also an autocompletion script for zsh, `_frama-c`, also in the shared resources directory. Look inside for installation instructions.

3.3.5 Splitting a Frama-C Execution into Several Steps

By default, Frama-C parses its command line in an *unspecified* order and runs its actions according to the read options. To enforce an order of execution, you have to use the option `-then`: Frama-C parses its command line until the option `-then` and runs its actions accordingly, *then* it parses its command line from this option to the end (or to the next occurrence of `-then`) and runs its actions according to the read options. Note that this second run starts with the results of the first one.

Consider for instance the following command.

```
$ frama-c -eva -ulevel 4 file.c -then -ulevel 5
```

It first runs the value analysis plug-in (option `-eva`, [7]) with an unrolling level of 4 (option `-ulevel`, Section 5.4). Then it re-runs the value analysis plug-in (option `-eva` is still set) with an unrolling level of 5.

It is also possible to specify a project (see Section 8.1) on which the actions are applied thanks to the option `-then-on`. Consider for instance the following command.

```
$ frama-c -semantic-const-fold main file.c -then-on propagated -eva
```

It first propagates constants in function `main` of `file.c` (option `-semantic-const-fold`) which generates a new project called `propagated`. Then it runs the value analysis plug-in on this new project. Finally it restores the initial default project, except if the option `-set-project-as-default` is used as follow:

```
$ frama-c -semantic-const-fold main file.c \
          -then-on propagated -eva -set-project-as-default
```

Another possibility is the option `-then-last` which applies the next actions on the last project created by a program transformer. For instance, the following command is equivalent to the previous one.

```
$ frama-c -semantic-const-fold main file.c -then-last -eva
```

The last option is `-then-replace` which behaves like `-then-last` but also definitively destroys the previous current project. It might be useful to prevent a prohibitive memory consumption. For instance, the following command is equivalent to the previous one, but also destroys the initial default project.

```
$ frama-c -semantic-const-fold main file.c -then-replace -eva
```

3.3.6 Verbosity and Debugging Levels

The Frama-C kernel and plug-ins usually output messages either in the GUI or in the console. Their levels of verbosity may be set by using the option `-verbose <level>`. By default, this level is 1. Setting it to 0 limits the output to warnings and error messages, while setting it to a number greater than 1 displays additional informative message (progress of the analyses, *etc*).

In the same fashion, debugging messages may be printed by using the option `-debug <level>`. By default, this level is 0: no debugging message is printed. By contrast with standard messages, debugging messages may refer to the internals of the analyzer, and may not be understandable by non-developers.

The option `-quiet` is a shortcut for `-verbose 0 -debug 0`.

In the same way that `-verbose` (resp. `-debug`) sets the level of verbosity (resp. debugging), the options `-kernel-verbose` (resp. `-kernel-debug`) and `-<plug-in shortname>-verbose` (resp. `-<plug-in shortname>-debug`) set the level of verbosity (resp. debugging) of the kernel and particular plug-ins. When both the global level of verbosity (resp. debugging) and a specific one are modified, the specific one applies. For instance, `-verbose 0 -slicing-verbose 1` runs Frama-C quietly except for the slicing plug-in.

It is also possible to choose which categories of message should be displayed for a given plugin. See section 6.2 for more information.

3.3.7 Copying Output to Files

Messages emitted by the logging mechanism (either by the kernel or by plug-ins) can be copied to files using the `-<plug-in shortname>-log` option (or `-kernel-log`), according to the following syntax: `kinds1:file1,kinds2:file2,...`

Its argument is a map from *kind* specifiers to output files, where each key is a set of flags defining the kind(s) of message to be copied, that is, a sequence of one or several of the following characters:

- `a`: all (equivalent to `defrw` or `dfiruw`)
- `d`: debug
- `e`: user or internal error (equivalent to `iu`)
- `f`: feedback
- `i`: internal error
- `r`: result
- `u`: user error
- `w`: warning

If `kinds` is empty (e.g. `:file1`), it defaults to `erw`, that is, copy all but debug and feedback messages.

`file1`, `file2`, etc. are the names of the files where each set of messages will be copied to. Each file will be overwritten if existing, or created otherwise. Note that multiple entries can be directed to a single file (e.g. `-kernel-log w:warn.txt -wp-log w:warn.txt`).

Here is an example of a sophisticated use case for this option:

```
frama-c -kernel-log ew:warn.log -wp-log ew:warn.log
     -metrics-log :../metrics.log [...] file.c
```

The command above will run some unspecified analyses ([...]), copying the error and warning messages produced by both the kernel and the `wp` plug-in into file `warn.log`, and copying all non-debug, non-feedback output from the `metrics` plug-in into file `../metrics.log` (note that there is a separator (`:`) before the file path).

This option does not suppress the standard Frama-C output, but only copies it to a file. Also, only messages which are effectively printed (according to the defined verbosity and debugging levels) will be copied.

3.3.8 Terminal Capabilities

Some plug-ins can take advantage of terminal capabilities to enrich output. These features are automatically turned off when the Frama-C standard output channel is not a terminal, which typically occurs when you redirect it into a file or through a pipe.

You can control use of terminal capabilities with option `-tty`, which is set by default and can be deactivated with `-no-tty`.

3.3.9 Getting time

The option `-time <file>` appends user time and date to the given log `<file>` at exit.

3.3.10 Inputs and Outputs of Source Code

The following options deal with the output of analyzed source code:

`-print` causes Frama-C's representation for the analyzed source files to be printed as a single C program (see Section 5.4). Note that files from the Frama-C standard library are kept by default under the form of `#include` directives, to avoid polluting the output. To expand them, use `-print-libc`.

`-ocode <file name>` redirects all output code of the current project to the designated file.

`-keep-comments` keeps C comments in-lined in the code.

`-unicode` uses unicode characters in order to display some ACSL symbols. This option is set by default, so one usually uses the opposite option `-no-unicode`.

A series of dedicated options deal with the display of floating-point and integer numbers:

`-float-hex` displays floating-point numbers as hexadecimal

`-float-normal` displays floating-point numbers with an internal routine

`-float-relative` displays intervals of floating-point numbers as `[lower bound ++ width]`

`-big-ints-hex <max>` prints all integers greater than `max` (in absolute value) using hexadecimal notation

3.4 Environment Variables

Different environment variables may be set to customize Frama-C.

3.4.1 Variable `FRAMAC_LIB`

External plug-ins (see Section 4.3) or scripts (see Section 4.4) are compiled against the Frama-C compiled library. The Frama-C option `-print-lib-path` prints the path to this library.

The default path to this library may be set when configuring Frama-C by using the `configure` option `--libdir`. Once Frama-C is installed, you can also set the environment variable `FRAMAC_LIB` to change this path.

3.4.2 Variable `FRAMAC_PLUGIN`

Dynamic plug-ins (see Section 4.4) are searched for in a default directory. The Frama-C option `-print-plugin-path` prints the path to this directory. It can be changed by setting the environment variable `FRAMAC_PLUGIN`.

3.4.3 Variable `FRAMAC_SHARE`

Frama-C looks for all its other data (installed manuals, configuration files, C modelization libraries, *etc*) in a single directory. The Frama-C option `-print-share-path` prints this path.

The default path to this library may be set when configuring Frama-C by using the `configure` option `--datarootdir`. Once Frama-C is installed, you can also set the environment variable `FRAMAC_SHARE` to change this path.

A Frama-C plug-in may have its own share directory (default is ‘`frama-c -print-share-path ‘/<plug-in shortname>`). If the plug-in is not installed in the standard way, you can set this share directory by using the option `-<plug-in shortname>-share`.

3.4.4 Variable `FRAMAC_SESSION`

Frama-C may have to generate files depending on the project under analysis during a session in order to reuse them later in other sessions.

By default, these files are generated or searched in the subdirectory `.frama-c` of the current directory. You can also set the environment variable `FRAMAC_SESSION` or the option `-session` to change this path.

Each Frama-C plug-in may have its own session directory (default is `.frama-c/<plug-in shortname>`). It is also possible to change a plug-in's session directory by using the option `-<plug-in shortname>-session`.

3.4.5 Variable `FRAMAC_CONFIG`

Frama-C may have to generate configuration files during a session in order to reuse them later in other sessions.

By default, these files are generated or searched in a subdirectory `frama-c` (or `.frama-c`) of the system's default configuration directory (*e.g.* `%USERPROFILE%` on Windows or `$HOME/.config` on Linux). You can also set the environment variable `FRAMAC_CONFIG` or the option `-config` to change this path.

Each Frama-C plug-in may have its own configuration directory if required (on Linux, default is `$HOME/.config/frama-c/<plug-in shortname>`). It is also possible to change a plug-in's config directory by using the option `-<plug-in shortname>-config`.

3.5 Exit Status

When exiting, Frama-C has one of the following status:

0 Frama-C exits normally without any error;

1 Frama-C exits because of invalid user input;

2 Frama-C exits because the user kills it (usually *via* `Ctrl-C`);

3 Frama-C exits because the user tries to use an unimplemented feature. Please report a "feature request" on the Bug Tracking System (see Chapter 13);

4,5,6 Frama-C exits on an internal error. Please report a "bug report" on the Bug Tracking System (see Chapter 13);

125 Frama-C exits abnormally on an unknown error. Please report a "bug report" on the Bug Tracking System (see Chapter 13).

Chapter 4

Setting Up Plug-ins

The Frama-C platform has been designed to support third-party plug-ins. In the present chapter, we present how to configure, compile, install, run and update such extensions. This chapter does not deal with the development of new plug-ins (see the Plug-in Development Guide [12]). Nor does it deal with usage of plug-ins, which is the purpose of individual plug-in documentation (see e.g. [7, 4, 3]).

4.1 The Plug-in Taxonomy

There are two kinds of plug-ins: *internal* and *external* plug-ins. Internal plug-ins are those distributed within the Frama-C kernel while external plug-ins are those distributed independently of the Frama-C kernel. They only differ in the way they are installed (see Sections 4.2 and 4.3).

4.2 Installing Internal Plug-ins

Internal plug-ins are automatically installed with the Frama-C kernel.

If you use a source distribution of Frama-C, it is possible to disable (resp. force) the installation of a plug-in of name `<plug-in name>` by passing the `configure` script the option `--disable-<plug-in name>` (resp. `--enable-<plug-in name>`). Disabling a plug-in means it is neither compiled nor installed. Forcing the compilation and installation of a plug-in against `configure`'s autodetection-based default may cause the entire Frama-C configuration to fail. You can also use the option `--with-no-plugin` in order to disable all plug-ins.

4.3 Installing External Plug-ins

To install an external plug-in, Frama-C itself must be properly installed first. In particular, `frama-c -print-share-path` must return the share directory of Frama-C (see Section 3.4.3), while `frama-c -print-lib-path` must return the directory where the Frama-C compiled library is installed (see Section 3.4.1).

The standard way for installing an external plug-in from source is to run the sequence of commands `make && make install`, possibly preceded by `./configure`. Please refer to each plug-in's documentation for installation instructions.

External plug-ins may also be configured and compiled at the same time as the Frama-C kernel by using the option `--enable-external=<path-to-plugin>` . This option may be passed several times.

4.4 Loading Plug-ins

At launch, Frama-C loads all plug-ins in the directories indicated by `frama-c -print-plugin-path` (see Section 3.4.2). Frama-C can locate plug-ins in additional directories by using the option `-add-path <paths>`. To prevent this behavior, you can use option `-no-autoload-plugins`.

Another way to load a plug-in is to set the `-load-module <files>` or `-load-script <files>` options, using in both cases a comma-separated list of specifications, which may be one of the following:

- an OCaml source file (in which case it will be compiled);
- an OCaml object file (`.cmo` or `.cma` if running Frama-C in bytecode, or `.cmxs` if running Frama-C in native code);
- a Findlib package.

File extensions for source and object files may be omitted, e.g. `-load-script file` will search for `file.ml` and `file.cm*` (where `cm*` depends if Frama-C is running in bytecode or native code).

In general, plug-ins must be compiled with the very same OCaml compiler than Frama-C was, and against a consistent Frama-C installation. Loading will fail and a warning will be emitted at launch if this is not the case.
These options require the OCaml compiler that was used to compile Frama-C to be available and the Frama-C compiled library to be found (see Section 3.4.1).

Chapter 5

Preparing the Sources

This chapter explains how to specify the source files that form the basis of an analysis project, and describes options that influence parsing.

5.1 Overview of source processing in Frama-C

For small projects and tests, processing the sources in Frama-C is as simple as running `frama-c *.c`. For more complex projects, however, some problems may arise when using this command, and the user must be aware of the several steps involved in Frama-C source processing to fix them.

The diagram in Figure 5.1 presents an overview of the steps described in this chapter. For comparison purposes, we add the equivalent process performed by a compiler such as GCC.

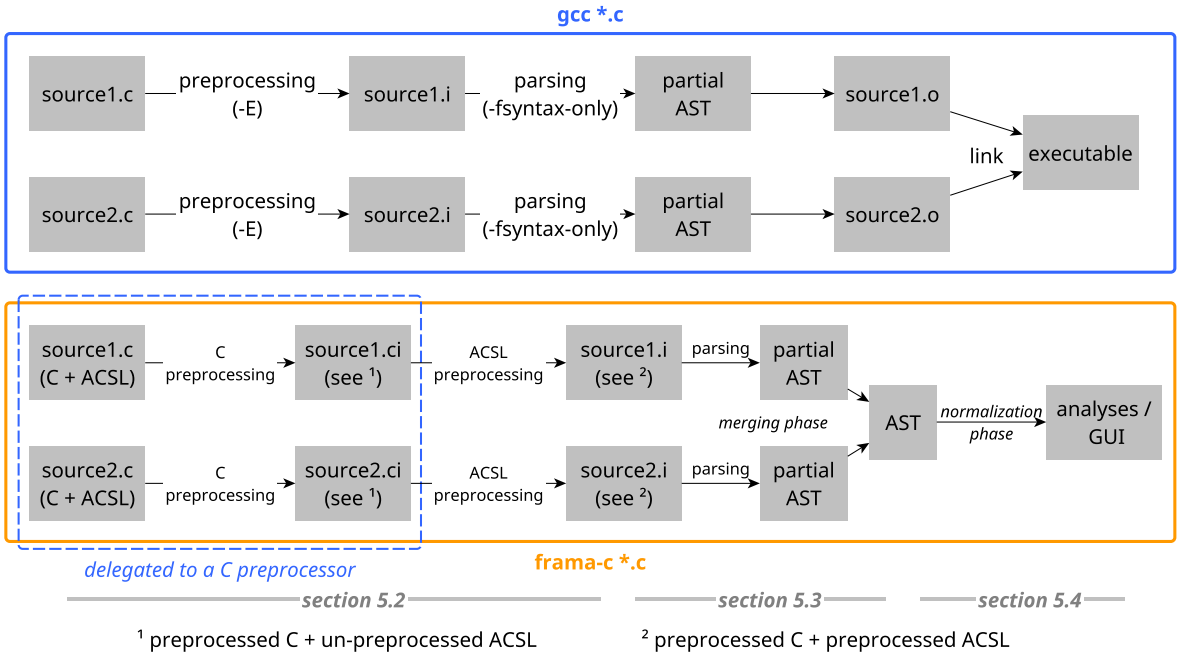

Figure 5.1: Overview of source preparation steps: as performed by GCC (top) and as performed by Frama-C (bottom).

The following sections describe various options related to the steps shown in the figure. Note that some plug-ins, such as Variadic (described in chapter 11), perform further AST transformations before most analyses are run.

5.2 Pre-processing the Source Files

The list of files to analyze must be provided on the command line, or chosen interactively in the GUI. Files with the suffix `.i` are assumed to be already pre-processed C files. Frama-C pre-processes the other files with the following command.

```
$ gcc -C -E -I .
```

Option `-cpp-extra-args` can be used to add arguments to the default pre-processing command, typically via the inclusion of defines (`-D` switches) and header directories (`-I` switches), as in `-cpp-extra-args="-DDEBUG -DARCH=ia32 -I./headers"`. You can also add arguments on a per-file basis, using option `-cpp-extra-args-per-file`.

If you need to, you can also *replace* the pre-processing command entirely with option `-cpp-command`. Placeholders (see below) can be used for advanced commands. If no placeholders are used, the pre-processor is invoked in the following way.

```
<cmd> <args> -o <output file> <input file>
```

In this command, `<output file>` is chosen by Frama-C while `<input file>` is one of the filenames provided by the user.

For commands which do not follow this pattern, it is also possible to use the following placeholders:

`%input`, `%i` or `%1`	Input file
`%output`, `%o` or `%2`	Output file
`%args`	Additional arguments (see `-cpp-extra-args` below)

Here are some examples for using this option.

```
$ frama-c -cpp-command 'gcc -C -E -I. -x c' file1.src file2.i
$ frama-c -cpp-command 'gcc -C -E -I. %args -o %o %i' file1.c file2.i
$ frama-c -cpp-command 'cp %i %o' file1.c file2.i
$ frama-c -cpp-command 'cat %i > %o' file1.c file2.i
$ frama-c -cpp-command 'CL.exe /C /E %args %i > %o' file1.c file2.i
```

When using `-cpp-command`, you may add `-cpp-frama-c-compliant` to indicate that the custom preprocessor accepts the same set of options as GNU `cpp`. `-cpp-frama-c-compliant` also implies some extra constraints, such as accepting architecture-specific flags, e.g. `-m32`.

If you have a JSON compilation database file[1], you can use it to retrieve preprocessing macros such as `-D` and `-I` for each file in the database, via option `-json-compilation-database <path>`, where `<path>` is the path to the JSON file or to a directory containing a file named `compile_commands.json`. With this option set, Frama-C will parse the compilation database and include associated preprocessing flags, as if they had been manually added via `-cpp-extra-args-per-file`. Note: if both `-cpp-extra-args-per-file` and the JSON compilation database specify options for a given file, the former are used and the latter are ignored.

[1] http://clang.llvm.org/docs/JSONCompilationDatabase.html

In all of the above cases, ACSL annotations are pre-processed by default (option `-pp-annot` is set by default). Unless a custom pre-processor is specified (via `-cpp-frama-c-compliant`), Frama-C considers that GCC is installed and uses it as pre-processor. If you do *not* want annotations to be pre-processed, you need to pass option `-no-pp-annot` to Frama-C. Note that some headers in the standard C library provided with Frama-C (described below) use such annotations, therefore it might be necessary to disable inclusion of such headers.

Also note that ACSL annotations are pre-processed separately from the C code in a second pass, and that arguments given as `-cpp-extra-args` are *not* given to the second pass of pre-processing. Instead, `-pp-annot` relies on the ability of GCC to output all macro definitions (including those given with `-D`) in the pre-processed file. In particular, `-cpp-extra-args` must be used if you are including header files who behave differently depending on the number of times they are included.

In addition, files having the suffix `.ci` will be considered as needing preprocessing for ACSL annotations only. Those files may contain `#define` directives and annotations are pre-processed as explained in the previous paragraph. This allows to have macros in ACSL annotations while using a non-GNU-like pre-processor.

An experimental, incomplete, C standard library is bundled with Frama-C and installed in the sub-directory `libc` of the directory printed by `frama-c -print-share-path`. It contains standard C headers, some ACSL specifications and definitions for some library functions. By default, these headers are used instead of the standard library ones: option `-frama-c-stdlib` (set by default) adds `-I$FRAMAC_SHARE/libc` to the preprocessor command, and also `-nostdinc` if `-cpp-frama-c-compliant` is set. Note that this library is provided on a best-effort basis, but it is part of the *trusted computing base*: the specifications must be proofread for correctness.

5.3 Merging the Source Code files

After pre-processing, Frama-C parses, type-checks and links the source code. It also performs these operations for the ACSL annotations optionally present in the program. Together, these steps form the *merging* phase of the creation of an analysis project.

Frama-C aborts whenever any error occurs during one of these steps. However users can use the option `-kernel-warn-key annot-error=active`[2] in order to continue after emitting a warning when an ACSL annotation fails to type-check.

5.4 Normalizing the Source Code

After merging the project files, Frama-C performs a number of local code transformations in the *normalization* phase. These transformations aim at making further work easier for the analyzers. Analyses usually take place on the normalized version of the source code. The normalized version may be printed by using the option `-print` (see Section 3.3.10).

Normalization gives a program which is semantically equivalent to the original one, except for one point. Namely, when the specification of a function `f` that is only declared and has no ACSL `assigns` clause is required by some analysis, Frama-C generates an `assigns` clause based on the prototype of `f` (the form of this clause is left unspecified). Indeed, as mentioned

[2] See section 6.2 for more information on warning statuses.

in the ACSL manual [2], assuming that `f` can write to any location in the memory would amount to stop any semantical analysis at the first call to `f`, since nothing would be known on the memory state afterwards. *The user is invited to check that the generated clause makes sense*, and to provide an explicit `assigns` clause if this is not the case.

The following options allow to customize the normalization process.

`-aggressive-merging` forces some function definitions to be merged into a single function if they are equal modulo renaming. Note that this option may merge two functions even if their original source code is different but their normalized version coincides. This option is mostly useful to share function definitions that stem from headers included by several source files.

`-allow-duplication` allows the duplication of small blocks of code during normalization of loops and tests. This is set by default and the option is mainly found in its opposite form, `-no-allow-duplication` which forces Frama-C to use labels and gotos instead. Note that bigger blocks and blocks with a non-trivial control flow are never duplicated. Option `-ulevel` (see below) is not affected by this option and always duplicates the loop body.

`-annot` forces Frama-C to interpret ACSL annotations. This option is set by default, and is only found in its opposite form `-no-annot`, which prevents interpretation of ACSL annotations.

`-asm-contracts` tells Frama-C to generate `assigns` clauses for inline `asm` statements using extended GNU notation with output and input operands. This option is set by default, and opposite form `-no-asm-contracts` prevents the generation of such clauses. If the assembly block already has a statement contract which is not guarded by a `for b:` clause `assigns` clauses are generated only for behaviors which do not already have one.

`-asm-contracts-auto-validate` tells Frama-C to automatically mark as valid `assigns` clauses generated from `asm` statements. However, if an `asm` statement contains `memory` in its clobber list, the corresponding clause will *not* be considered valid through this option.

`-collapse-call-cast` allows, in some cases, the value returned by a function call to be implicitly cast to the type of the value it is assigned to (if such a conversion is authorized by the C standard). Otherwise, a temporary variable separates the call and the cast. The default is to have implicit casts for function calls, so the opposite form `-no-collapse-call-cast` is more useful.

`-constfold` performs a syntactic folding of constant expressions. For instance, the expression `1+2` is replaced by `3`.

`-continue-annot-error` *Deprecated option.* Just emits a warning and discards the annotation when it fails to type-check, instead of generating an error (errors in C are still fatal). This behavior is now managed by warning category (Section 6.2) `annot-error`, whose default status is "abort". Behavior of `-continue-annot-error` can thus be obtained with `-kernel-warn-key annot-error=active`, or even `-kernel-warn-key annot-error=inactive` to silently ignore any ill-formed annotations.

`-enums <repr name>` specifies which representation should be used for a given enumerated type. Namely, the C standard allows the use of any integral type in which all the corresponding tags can be represented. Default is `gcc-enums`. A list of supported options can be obtained by typing:

```
$ frama-c -enums help
```

This includes:

- `int`: treat everything as `int` (including enumerated types with `packed` attribute).
- `gcc-enums`: use an unsigned integer type when no tag has a negative value, and choose the smallest rank possible starting from `int` (default GCC's behavior)
- `gcc-short-enums`: use an unsigned integer type when no tag has a negative value, and choose the smallest rank possible starting from `char` (GCC's `-fshortenums` option)

`-initialized-padding-locals` forces to initialize padding bits of locals to 0. If false, padding bits are left uninitialized. This option is set by default.

`-inline-calls <f1,...,fn>` inlines calls to functions. Use `@inline` to select all functions with attribute `inline`. For recursive functions, only the first level is inlined (e.g., the function will contain an inlined version of itself with a recursive call inside it). Calls via function pointers are ignored.

`-remove-inlined <f1,...,fn>` removes from the AST functions `f1,...,fn`, which must have been given to `-inline-calls`. Note: this option does not check if the given functions were fully inlined.

`-keep-switch` preserves `switch` statements in the source code. Without this option, they are transformed into `if` statements. An experimental plug-in may require this option to be unset to avoid having to handle the `switch` construct. Other plug-ins may prefer this option to be used because it better preserves the structure of the original program.

`-keep-unused-specified-functions` does not remove from the AST uncalled function prototypes that have ACSL contracts. This option is set by default. So you mostly use the opposite form, namely `-remove-unused-specified-functions`.

`-keep-unused-types` does not remove unused types and enum/struct/union declarations. By default, such types are removed, that is, its opposite option `-remove-unused-types` is set.

`-machdep <machine architecture name>` defines the target platform. The default value is a `x86_32` bits platform. Analyzers may take into account the *endianness* of the target, the size and alignment of elementary data types, and other architecture/compilation parameters. The `-machdep` option provides a way to define all these parameters consistently in a single step. Note that multiarch preprocessors such as GCC's may require special flags to ensure compatibility with the chosen machdep, e.g. `-m32` for a `x86_64` GCC compiling 32-bit code. In most cases this is handled automatically, but otherwise option `-cpp-command` can be used.

The list of supported platforms can be obtained by typing:

```
$ frama-c -machdep help
```

The process for adding a new platform is described in the Plug-in Development Guide [12].

`-simplify-cfg` allows Frama-C to remove break, continue and switch statements. This option is automatically set by some plug-ins that cannot handle these kinds of statements. This option is off by default.

`-simplify-trivial-loops` simplifies trivial loops such as `do ... while(0)`. This option is set by default.

`-ulevel <n>` unrolls all loops n times. This is a purely syntactic operation. Loops can be unrolled individually, by inserting the `UNROLL` pragma just before the loop statement. Do not confuse this option with plug-in-specific options that may also be called "unrolling" [7]. Below is a typical example of use.

```
/*@ loop pragma UNROLL 10; */
for (i = 0; i < 9; i++) ...
```

The transformation introduces an `UNROLL` pragma indicating that the unrolling process has been done:

```
... // loop unrolled 10 times
/*@ loop pragma UNROLL 10;
    loop pragma UNROLL "done", 10; */
  ... // remaining loop
```

That allows to disable unrolling transformation on such a loop when reusing Frama-C with a code obtained by a previous use of Frama-C tool. To ignore this disabling `UNROLL` pragma and force unrolling, the option `-ulevel-force` has to be set.

Passing a negative argument to `-ulevel` will disable unrolling, even in case of `UNROLL` pragma.

`-c11` allows the use of some C11 constructs. Currently supported are typedefs redefinition.

5.5 Predefined macros

Frama-C, like several compilers and code analysis tools, predefines and uses a certain number of C macros. They are summarized below.

`__FRAMAC__`: defined to 1 during pre-processing by Frama-C, as if the user had added `-D__FRAMAC__` to the command line. Useful for conditional compilation and detection of an execution by Frama-C.

`__FC_MACHDEP_`*`XXX`*, where *`XXX`* is one of `X86_16`, `X86_32`, `X86_64`, `PPC_32` or `MSVC_X86_64`: according to the option `-machdep` chosen by the user, the corresponding macro is predefined by Frama-C. Those macros correspond to the values of `-machdep` that are built-in in the Frama-C kernel. In addition, custom machdeps can be added by plug-ins or scripts. If such a machdep named `custom-name` is selected, Frama-C will predefine the macro `__FC_MACHDEP_CUSTOM_NAME`, that is the upper-case version of the name of the machdep. Conversely, if an `__FC_MACHDEP_`*`XXX`* macro is defined by the user in `-cpp-command` or `-cpp-extra-args`, then Frama-C will consider it as a custom machdep and will *not* add any machdep-related macros. It is then the responsibility of the user to ensure that `-machdep` and `__FC_MACHDEP_`*`XXX`* are coherent.

`__FC_*`: macros prefixed by `__FC_` are reserved by Frama-C and should not be defined by the user (except for custom machdeps, as mentioned above, and for customization macros, as described below). These include machdep-related macros and definitions related to Frama-C's standard library.

Furthermore, some macros are undefined in the standard library, and can be defined (e.g. through `-cpp-extra-args`) to customize the Frama-C standard library. They are described below.

`__FC_NO_MONOTONIC_CLOCK`: By default, Frama-C defines a `MONOTONIC_CLOCK` in `time.h`. If this macro is defined, this clock is not available.

`__FC_INDETERMINABLE_FLOATS`: By default, Frama-C's libc uses an IEEE-754 compatible environment. If this macro is defined, rounding mode and float evaluation (macros `FLT_ROUNDS` and `FLT_EVAL_METHOD`) will be considered as indeterminable.

5.6 Compiler and language extensions

Frama-C's default behavior is to be fairly strict concerning language features. By default, most non-C99 compiler extensions are not accepted, similarly to when compiling a program with `gcc -std=c99 -pedantic -pedantic-errors`.

However, depending on the machine architecture (see option `-machdep`, in Section 5.4), Frama-C accepts some compiler extensions, namely for GCC and MSVC machdeps. For instance, trying to parse a program containing an empty initializer, such as `int c[10] = {};` will result in the following error message:

```
[kernel] user error: empty initializers only allowed for GCC/MSVC
```

This means that using a GCC or MSVC machdep (e.g., `-machdep gcc_x86_32`) will allow the language extension to be accepted by Frama-C.

Alternatively, some C11 extensions are supported via option `-c11`. For instance, the following program is invalid in C99 but valid in C11 (typedef redefinition): `typedef int a; typedef int a;`

The error message given by Frama-C when trying to parse this program will indicate the option needed to allow the file to be parsed:

```
[kernel] user error: redefinition of type 'a' in the same scope is only
         allowed in C11 (option -c11).
```

5.7 Warnings during normalization

Note: the options below are deprecated, replaced by the more general and flexible mechanism of warning categories*, described in Section 6.2.*

Some options can be used to influence the warnings that are emitted by Frama-C during the normalization phase.

`-warn-decimal-float <freq>` (*deprecated*) warns when floating-point constants in the program cannot be exactly represented; freq must be one of `none`, `once` or `all`. Defaults to `once`. *Superseded by warning category* ***`parser:decimal-float`****.*

`-implicit-function-declaration <action>` (*deprecated*) defines the action to perform (`ignore`, `warn` or `error`) whenever a call to a function that has not been previously declared is found. This is invalid in C90 or in C99, but could be valid K&R code. Defaults to `warn`. *Superseded by warning category* ***typing:implicit-function-declaration****.* **Note:** parsing is not guaranteed to succeed, regardless of the emission of the warning. Upon encountering a call to an undeclared function, Frama-C attempts to continue its parsing phase by inferring a prototype corresponding to the type of the arguments at the call (modulo default argument promotions). If the real declaration does not match the inferred prototype, parsing will later end with an error.

5.8 Testing the Source Code Preparation

If the steps up to normalization succeed, the project is then ready for analysis by any Frama-C plug-in. It is possible to test that the source code preparation itself succeeds, by running Frama-C without any option.

```
$ frama-c <input files>
```

If you need to use other options for pre-processing or normalizing the source code, you can use the option `-typecheck` for the same purpose. For instance:

```
frama-c -cpp-command 'gcc -C -E -I. -x c' -typecheck file1.src file2.i
```

Chapter 6

Platform-wide Analysis Options

The options described in this chapter provide each analysis with common hypotheses that influence directly their behavior. For this reason, the user must understand them and the interpretation the relevant plug-ins have of them. Please refer to individual plug-in documentations (e.g. [7, 3, 4]) for specific options.

6.1 Entry Point

The following options define the entry point of the program and related initial conditions.

`-main <function_name>` specifies that all analyzers should treat function `function_name` as the entry point of the program.

`-lib-entry` indicates that analyzers should not assume globals to have their initial values at the beginning of the analysis. This option, together with the specification of an entry point `f`, can be used to analyze the function `f` outside of a calling context, even if it is not the actual entry point of the analyzed code.

6.2 Feedback Options

All Frama-C plug-ins define the following set of common options.

`-<plug-in shortname>-help` (or `-<plug-in shortname>-h`) prints out the list of options of the given plug-in.

`-<plug-in shortname>-verbose <n>` sets the level of verbosity to some positive integer `n`. A value of 0 means no information messages. Default is 1.

`-<plug-in shortname>-debug <n>` sets the debug level to a positive integer `n`. The higher this number, the more debug messages are printed. Debug messages do not have to be understandable by the end user. This option's default is 0 (no debugging messages).

`-<plug-in shortname>-msg-key <keys>` sets the categories of messages that must be output for the plugin. `keys` is a comma-separated list of names. The list of available categories can be obtained with `-<plug-in shortname>-msg-key help`. To enable all

categories, use the wildcard `'*'`[1]. Categories can have subcategories, defined by a colon in their names. For instance, `a:b:c` is a subcategory `c` of `a:b`, itself a subcategory of `a`. Enabling a category will also enable all its subcategories. An enabled category `cat` can be disabled by using `-cat` in the list of `keys`. Several occurrences of the option may appear on the command line and will be processed in order.

`-<plug-in shortname>-warn-key <keys>` allows setting the status of a category of warnings. The argument `keys` is a comma-separated list of `key` of the form `<category>=<status>`, where category is a warning category (possibly a sub-category as for messages categories above), and status is one of:

inactive no message is emitted for the category

feedback a feedback message is emitted

active a proper warning is emitted

once a proper warning is emitted, and the status of the category is reset to `inactive`, *i.e.* at most one message for the category will be emitted.

error a warning is emitted. Frama-C execution continues, but its exit status will not be 0 at the end of the run.

abort a warning is emitted and Frama-C will immediately abort its execution with an error.

feedback-once a feedback message is emitted, and the status is reset to `inactive`

err-once combines the actions of `error` and `once` statuses.

The `=<status>` part might be omitted, which is equivalent to asking for `active` status. The new status will be propagated to subcategories, with one exception: statuses "abort", "error" and "err-once" will only be propagated to subcategories whose current status is not "inactive".

Finally, as for debug categories, passing `help` (without status) in the list of `keys` will list the available categories, together with their current status. Passing `*` in the list of keys will change the status of all warning categories, and affect warnings that do not have a category. Hence, `-kernel-warn-key *=abort` will stop Frama-C's execution at the first warning triggered by the kernel.

The two following options modify the behavior of output messages:

`-add-symbolic-path` takes a list of the form $name_1 : path_1, \ldots, name_n : path_n$ in argument and replaces each $path_i$ by $name_i$ when displaying file locations in messages.

`-permissive` performs less verification on validity of command-line options.

6.3 Customizing Analyzers

The descriptions of the analysis options follow. For the first two, the description comes from the Eva manual [7]. Note that these options are very likely to be modified in future versions of Frama-C.

[1] Be sure to enclose it in single quotes or your shell might expand it, leading to unexpected results.

`-absolute-valid-range m-M` specifies that the only valid absolute addresses (for reading or writing) are those comprised between `m` and `M` inclusive. This option currently allows to specify only a single interval, although it could be improved to allow several intervals in a future version. m and M can be written either in decimal or hexadecimal notation.

`-unsafe-arrays` can be used when the source code manipulates n-dimensional arrays, or arrays within structures, in a non-standard way. With this option, accessing indexes that are out of bounds will instead access the remainder of the struct. For example, the code below will overwrite the fields `a` and `c` of `v`.

```
struct s {
  int a;
  int b[2];
  int c;
};

void main(struct s v) {
  v.b[-1] = 1;
  v.b[2] = 4;
}
```

The opposite option, called `-safe-arrays`, is set by default. With `-safe-arrays`, the two accesses to `v` are considered invalid. (Accessing `v.b[-2]` or `v.b[3]` remains incorrect, regardless of the value of the option.)

`-warn-invalid-pointer` may be used to check that the code does not perform illegal pointer arithmetics, creating pointers that do not point inside an object or one past an object. This option is disabled by default, allowing the creation of such invalid pointers without alarm — but the dereferencing of an invalid pointer *always* generates an alarm.

For instance, no error is detected by default in the following example, as the dereferencing is correct. However, if option `-warn-invalid-pointer` is enabled, an error is detected at line 4.

```
    int x;
    int *p = &x;
    p++; // valid
    p++; // undefined behavior
    *(p-2) = 1;
```

`-unspecified-access` may be used to check when the evaluation of an expression depends on the order in which its sub-expressions are evaluated. For instance, this occurs with the following piece of code.

```
int i, j, *p;
i = 1;
p = &i;
j = i++ + (*p)++;
```

In this code, it is unclear in which order the elements of the right-hand side of the last assignment are evaluated. Indeed, the variable j can get any value as i and p are aliased. The `-unspecified-access` option warns against such ambiguous situations. More precisely, `-unspecified-access` detects potential concurrent write accesses (or a write access and a read access) over the same location that are not separated by a sequence point. Note however that this option *does not warn* against such accesses if they occur in an inner function call, such as in the following example:

```
int x;
int f() { return x++; }
int g() { return f() + x++; }
```

Here, the `x` might be incremented by `g` before or after the call to `f`, but since the two write accesses occur in different functions, `-unspecified-access` does not detect that.

`-warn-pointer-downcast` may be used to check that the code does not downcast a pointer to an integer type. This option is set by default. In the following example, analyzers report by default an error on the third line. Disabling the option removes this verification.

```
    int x;
    uintptr_t addr = &x;
    int a = &x;
```

`-warn-signed-downcast` may be used to check that the analyzed code does not downcast an integer to a signed integer type. This option is *not* set by default. Without it, the analyzers do not perform such a verification. For instance consider the following function.

```
short truncate(int n) {
  return (short) n;
}
```

If `-warn-signed-downcast` is set, analyzers report an error on `(short) n` which downcasts a signed integer to a signed short. Without it, no error is reported.

`-warn-unsigned-downcast` is the same as `-warn-signed-downcast` for downcasts to unsigned integers. This option is also *not* set by default.

`-warn-signed-overflow` may be used to check that the analyzed code does not overflow on integer operations. If the opposite option `-no-warn-signed-overflow` is specified, the analyzers assume that operations over signed integers may overflow by following two's complement representation. This option is set by default. For instance, consider the function `abs` that computes the absolute value of its `int` argument.

```
int abs(int x) {
  if (x < 0) x = -x;
  return x;
}
```

By default, analyzers detect an error on `-x` since this operation overflows when `MININT` is the argument of the function. But, with the `-no-warn-signed-overflow` option, no error is detected.

`-warn-unsigned-overflow` is the same as `-warn-signed-overflow` for operations over unsigned integers. This option is *not* set by default.

`-warn-left-shift-negative` can be used to check that the code does not perform signed left shifts on negative values, i.e., `x << n` with `x` having signed type and negative value. This is set by default, and can be disabled with option `-no-warn-left-shift-negative`.

`-warn-right-shift-negative`, as its left-shift counterpart, can be used to check for negative *right* shifts, i.e., `x >> n` with `x` having signed type and negative value. This is *not* set by default. `-no-warn-right-shift-negative` can be used to disable the option if previously enabled.

`-warn-special-float <type>` may be used to allow or forbid special floating-point values, generating alarms when they are produced. `<type>` can be one of the following values:

`non-finite` : warn on infinite floats or NaN

`nan` : warn on NaN only

`none` : no warnings

`-warn-invalid-bool` may be used to check that the code does not use invalid _Bool values by reading trap representations from lvalues of _Bool types. A trap representation does not represent a valid value of the _Bool type, which can only be 0 or 1. This option is set by default, and can be disabled with `-no-warn-invalid-bool`.

Chapter 7

Property Statuses

This chapter touches on the topic of program properties, and their validation by either standalone or cooperating Frama-C plug-ins. The theoretical foundation of this chapter is described in a research paper [5].

7.1 A Short Detour through Annotations

Frama-C supports writing code annotations with the ACSL language [2]. The purpose of annotations is to formally specify the properties of C code: Frama-C plug-ins can rely on them to demonstrate that an implementation respects its specification.

Annotations can originate from a number of different sources:

the user who writes his own annotations: an engineer writing code specifications is the prevalent scenario here;

some plug-ins may generate code annotations. These annotations can, for instance, indicate that a variable needs to be within a safe range to guarantee no runtime errors are triggered (cf the RTE plug-in [9]).

the kernel of Frama-C, that attempts to generate as precise an annotation as it can, when none is present.

Of particular interest is the case of unannotated function prototypes[a]: the ACSL specification states that a construct of that kind "potentially modifies *everything*" [2, Sec. 2.3.5]. For the sake of precision and conciseness, the Frama-C kernel breaks this specification, and generates a function contract with clauses that relate its formal parameters to its results[b]. This behavior might be incorrect – for instance because it does not consider functions that can modify globals. While convenient in a wide range of cases, this can be averted by writing a custom function contract for the contentious prototypes.

[a] A function prototype is a function declaration that provides argument types and return type, but lacks a body.

[b] Results here include the return value, and the formal modifiable parameters.

The rest of this chapter will examine how plug-ins can deal with code annotations, and in particular what kind of information can be attached to them.

7.2 Properties, and the Statuses Thereof

A property is a logical statement bound to a precise code location. A property might originate from:

- an ACSL code annotation – e.g. `assert p[i] * p[i] <= INT_MAX`. Recall from the previous section that annotations can either be written by the user, or generated by the Frama-C plug-ins or kernel;
- a plugin-dependent meta-information – such as the memory model assumptions.

Consider a program point i, and call T the set of traces that run through i. More precisely, we only consider the traces that are coming from the program entry point[1] (see option `-main` in chapter 6). A logical property P is valid at i if it is valid on all $t \in T$. Conversely, any trace u that does not validate P, stops at i: properties are *blocking*.

As an example, a property might consist in a statement $p[j] \times p[j] \leq 2147483647$ at a program point i. A trace where $p[j] = 46341$ at i will invalidate this property, and will stop short of reaching any instruction succeeding i.

An important part of the interactions between Frama-C components (the plug-ins/the kernel) rely on their capacity to *emit* a judgment on the validity of a property P at program point i. In Frama-C nomenclature, this judgment is called a *local property status*. The first part of a local status ranges over the following values:

- `True` when the property is true for all traces;
- `False` when there exists a trace that falsifies the property;
- `Maybe` when the emitter e cannot decide the status of P.

As a second part of a local property status, an emitter can add a list of *dependencies*, which is the set of properties whose validity may be necessary to establish the judgment. For instance, when the WP plug-in [4] provides a demonstration of a Hoare triple $\{A\}\ c\ \{B\}$, it starts by setting the status of B to "True", and then adds to this status a dependency on property A. In more formal terms, it corresponds to the judgment $\vdash A \Rightarrow B$: "for a trace to be valid in B, it may be necessary for A to hold". This information on the conditional validity of B is provided *as a guide* for validation engineers, and should not be mistaken for the formal proof of B, which only holds when *all* program properties are verified – hence the *local* status.

7.3 Consolidating Property Statuses

Recall our previous example, where the WP plug-in sets the local status of a property B to "True", with a dependency on a property A. This might help another plug-in decide that the validity of a third property C, that hinges upon B, now depends on A. When at last A is proven by, say, the value analysis plug-in, the cooperative proofs of A, B, and C are marked

[1] Some plug-ins might consider *all possible traces*, which constitute a safe over-approximation of the intended property.

as completed. In formal terms, Frama-C has combined the judgments: $\vdash A \Rightarrow B$, $\vdash B \Rightarrow C$, and $\vdash A$ into proofs of $\vdash B$ and $\vdash C$, by using the equivalent of a *modus ponens* inference:

$$\frac{\overline{\vdash A} \quad \overline{\vdash A \Rightarrow B}}{\vdash B}$$

Notice how, without the final $\vdash A$ judgment, both proofs would be incomplete.

This short example illustrates how incremental the construction of program property proofs can be. By *consolidating* property statuses into an easily readable display, Frama-C aims at informing its users of the progress of this process, allowing them to track unresolved dependencies, and selectively validate subsets of the program's properties.

As a result, a consolidated property status can either be a *simple* status:

- `never_tried`: when no status is available for the property.
- `unknown`: whenever the status is `Maybe`.
- `surely_valid`: when the status is `True`, and dependencies have the consolidated status `surely_valid` or `considered_valid`.
- `surely_invalid`: when the status is `False`, and all dependencies have the consolidated status `surely_valid`.
- `inconsistent`: when there exist two conflicting consolidated statuses for the same property, for instance with values `surely_valid` and `surely_invalid`. This case may also arise when an invalid cyclic proof is detected. This is symptomatic of an incoherent axiomatization.

or an *incomplete* status:

- `considered_valid`: when there is no possible way to prove the property (e.g., the post-condition of an external function). We assume this property will be validated by external means.
- `valid_under_hyp`: when the local status is `True` but at least one of the dependencies has consolidated status `unknown`. This is typical of proofs in progress.
- `invalid_under_hyp`: when the local status is `False`, but at least one of the dependencies has status `unknown`. This is a telltale sign of a dead code property, or of an erroneous annotation.

and finally:

- `unknown_but_dead`: when the status is locally `Maybe`, but in a dead or incoherent branch.
- `valid_but_dead`: when the status is locally `True`, but in a dead or incoherent branch.
- `invalid_but_dead`: when the status is locally `False`, but in a dead or incoherent branch.

The dependencies are meant *as a guide* to safety engineers. They are neither correct, nor complete, and should not be relied on for formal assessment purposes. In particular, as long as partial proofs exist (there are `unknown` or `never_tried`), there is no certainty with regards to any other status (including `surely_valid` properties).

These consolidated statuses are displayed in the GUI (see section 9 for details), or in batch mode by the `report` plug-in.

Chapter 8

General Kernel Services

This chapter presents some important services offered by the Frama-C platform.

8.1 Projects

A Frama-C project groups together one source code with the states (parameters, results, *etc*) of the Frama-C kernel and analyzers.

In one Frama-C session, several projects may exist at the same time, while there is always one and only one so-called *current* project in which analyses are performed. Thus projects help to structure a code analysis session into well-defined entities. For instance, it is possible to perform an analysis on the same code with different parameters and to compare the obtained results. It is also possible to extract a program p' from an initial program p and to compare the results of an analysis run separately on p and p'.

8.1.1 Creating Projects

A new project is created in the following cases:

- at initialization time, a default project is created; or
- *via* an explicit user action in the GUI; or
- a source code transforming analysis has been made. The analyzer then creates a new project based on the original project and containing the modified source code. A typical example is code slicing which tries to simplify a program by preserving a specified behaviour.

8.1.2 Using Projects

The list of existing projects of a given session is visible in the graphical mode through the `Project` menu (see Section 9.2). Among other actions on projects (duplicating, renaming, removing, saving, *etc*), this menu allows the user to switch between different projects during the same session.

In batch mode, the only way to handle a multi-project session is through the command line options `-then-on`, `-then-last` or `-then-replace` (see Section 3.3.5). It is also possible to

remove existing projects through the option `-remove-projects`. It might be useful to prevent prohibitive memory consumptions. In particular, the category `@all_but_current` removes all the existing projects, but the current one.

8.1.3 Saving and Loading Projects

A session can be saved to disk and reloaded by using the options `-save <file>` and `-load <file>` respectively. Saving is performed when Frama-C exits without error. In case of a fatal error or an unexpected error, saving is done as well, but the generated file is modified into `file.crash` since it may have been corrupted. In other error cases, no saving is done. The same operations are available through the GUI.

When saving, *all* existing projects are dumped into an unique non-human-readable file.

When loading, the following actions are done in sequence:

1. all the existing projects of the current session are deleted;
2. all the projects stored in the file are loaded;
3. the saved current project is restored;
4. Frama-C is replayed with the parameters of the saved current project, except for those parameters explicitly set in the current session.

Consider for instance the following command.

```
$ frama-c -load foo.sav -eva
```

It loads all projects saved in the file `foo.sav`. Then, it runs the value analysis in the new current project if and only if it was not already computed at save time.

Recommendation 8.1 *Saving the result of a time-consuming analysis before trying to use it in different settings is usually a good idea.*

Beware that all the existing projects are deleted, even if an error occurs when reading the file. We strongly recommend you to save the existing projects before loading another project file.

Special Cases Options `-help`, `-verbose`, `-debug` (and their corresponding plugin-specific counterpart) as well as `-quiet` and `-unicode` are not saved on disk.

8.2 Dependencies between Analyses

Usually analyses do have parameters (see Chapter 6). Whenever the values of these parameters change, the results of the analyses may also change. In order to avoid displaying results that are inconsistent with the current value of parameters, Frama-C automatically discards results of an analysis when one of the analysis parameters changes.

Consider the two following commands.

```
$ frama-c -save foo.sav -ulevel 5 -absolute-valid-range 0-0x1000 -eva foo.c
$ frama-c -load foo.sav
```

Frama-C runs the value analysis plug-in on the file `foo.c` where loops are unrolled 5 times (option `-ulevel`, see Section 5.4). To compute its result, the value analysis assumes the memory range 0:0x1000 is addressable (option `-absolute-valid-range`, see Section 6.3). Just after, Frama-C saves the results on file `foo.sav` and exits.

At loading time, Frama-C knows that it is not necessary to redo the value analysis since the parameters have not been changed.

Consider now the two following commands.

```
$ frama-c -save foo.sav -ulevel 5 -absolute-valid-range 0-0x1000 -eva foo.c
$ frama-c -load foo.sav -absolute-valid-range 0-0x2000
```

The first command produces the very same result than above. However, in the second (load) command, Frama-C knows that one parameter has changed. Thus it discards the saved results of the value analysis and recomputes it on the same source code by using the parameters `-ulevel 5 -absolute-valid-range 0-0x2000` (and the default value of each other parameter).

In the same fashion, results from an analysis A_1 may well depend on results from another analysis A_2. Whenever the results from A_2 change, Frama-C automatically discards results from A_1. For instance, slicing results depend on value analysis results; thus the slicing results are discarded whenever the value analysis ones are.

8.3 Journalisation

Journalisation logs each operation that modifies some parameters or results into a file called a *journal*. Observational operations like viewing the set of possibles values of a variable in the GUI are not logged.

By default, the name of the journal is `SESSION_DIR/frama_c_journal.ml` where `SESSION_DIR` is the Frama-C session directory (see Section 3.4.4). It can be modified by using the option `-journal-name`.

A journal is a valid Frama-C dynamic plug-in. Thus it can be loaded by using the option `-load-script` (see Section 4.4). The journal replays the very same results as the ones computed in the original session.

Journals are commonly used for the three different purposes described thereafter.

- Easily replaying a given set of analysis operations in order to reach a certain state. Once the final state is reached, further analyses can be performed normally. Beware that journals may be source dependent and thus may not necessarily be reused on different source codes to perform the same analyses.
- Acting as a macro language for plug-in developers. They can perform actions on the GUI to generate a journal and then adapt it to perform a more general but similar task.
- Debugging. In the GUI, a journal is always generated, even when an error occurs. The output journal usually contains information about this error. Thus it provides an easy way to reproduce the very same error. Consequently, it is advised to attach the journal when reporting an error in the Frama-C BTS (see Chapter 13).

By default, a journal is generated upon exit of the session only whenever Frama-C crashes in graphical mode. In all other cases, no journal is generated. This behavior may be customized

by using the option `-journal-enable` (resp. `-journal-disable`) that generates (resp. does not generate) a journal upon exiting the session.

Special Cases Modifications of options `-help`, `-verbose`, `-debug` (and their corresponding counterpart) as well as `-quiet` and `-unicode` are not written in the journal.

Chapter 9

Graphical User Interface

Running `frama-c-gui` or `frama-c-gui.byte` displays the Frama-C Graphical User Interface (GUI).

9.1 Frama-C Main Window

Upon launching Frama-C in graphical mode on some C files, the following main window is displayed (figure 9.1):

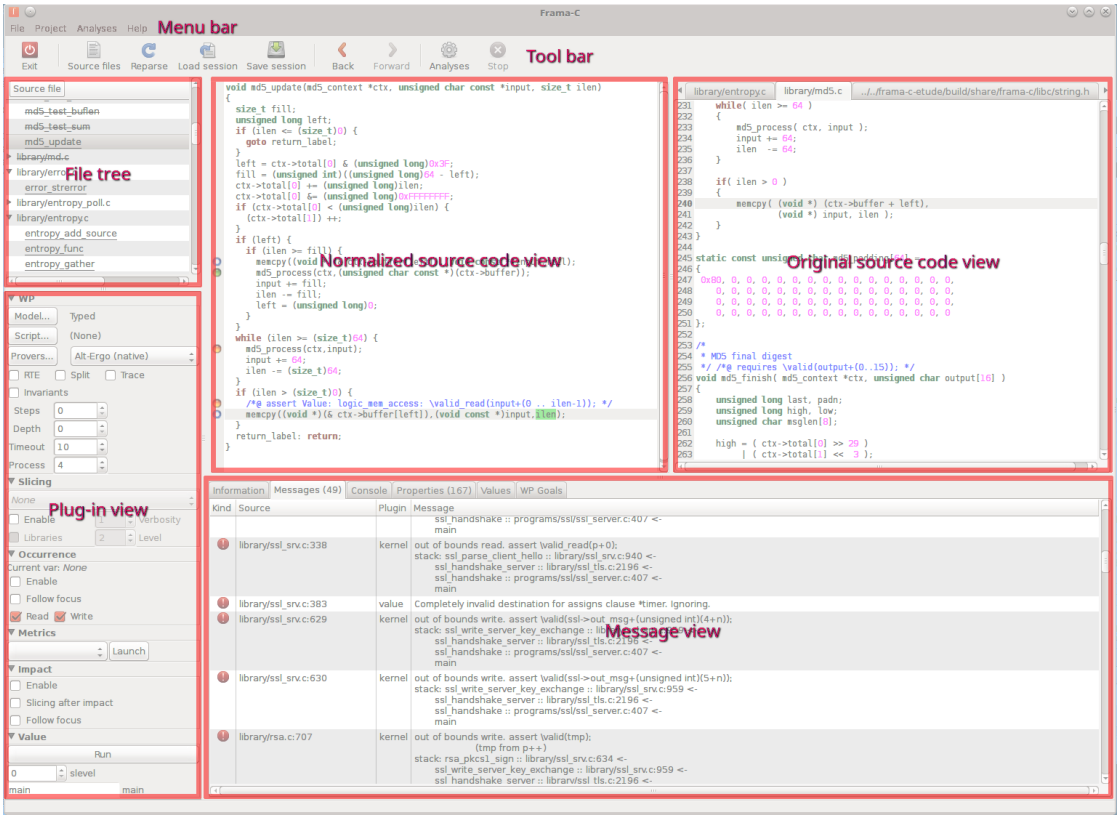

Figure 9.1: Initial View

From top to bottom, the window is made of several separate sub-parts.

The menu bar organizes the highest-level functions of the tool into structured categories. Plug-ins may also add their own entries in the "Analyses" menu.

The toolbar gives access to the main functions of the tool. They are usually present in one menu of the menu bar. Plug-ins may also add their own entries here.

The file tree provides a tree-like structure of the source files involved in the current analysis. This tree lists all the global variables and functions each file contains. Within a file, entries are sorted alphabetically, without taking capitalization into account. Functions are underlined, to separate them from variables. Plug-ins may also display specific information for each file and/or function. Finally, the "Source file" button offers some options to filter the elements of the file tree:

- The "Hide variables" and "Hide functions" options offer the possibility to hide the non-desired entries from the tree.
- The "Flat mode" option flattens the tree, by removing the filename level. Instead, functions and globals are displayed together, as if they were in a big namespace. This makes it easier to find a function whose only the name is known.

The normalized and original source code views display the source code of the current selected element of the file tree and its normalized code (see Section 5.4). Left-clicking on an object (statement, left-value, *etc*) in the normalized source code view displays information about it in the "Information" page of the Messages View and displays the corresponding object of the original source view, while right-clicking on them opens a contextual menu. Items of this menu depend on the kind of the selected object and on plug-in availability.

Only the normalized source view is interactive: the original one is not.

The plug-ins view shows specific plug-in interfaces. The interface of each plug-in can be collapsed.

The messages view contains by default six different pages, namely:

Information: provides brief details on the currently selected object, or informative messages from the plugins.

Messages: shows important messages generated by the Frama-C kernel and plug-ins, in particular all alarms. Please refer to the specific documentation of each plug-in in order to get the exact form of alarms. Alarms that have a location in the original source can be double-clicked; this location will then be shown in the original and normalized source code viewers.[1]

Console: displays messages to users in a textual way. This is the very same output than the one shown in batch mode.

Properties: displays the local and consolidated statuses of properties.

Values: displays information relative to the Value Analysis plug-in.

WP Goals: displays information relative to the WP plug-in.

[1] Notice however that the location in the normalized source may not perfectly correspond, as more than one normalized statement can correspond to a source location.

9.2 Menu Bar

The menu bar is organised as follows:

The file menu proposes items for managing the current session.

`Source files`: changes the analyzed files of the current project.

`Reparse`: reloads the source files of the current project from the disk, reparses them, and restarts the analyses that have been configured.

`Save session`: saves all the current projects into a file. If the user has not yet specified such a file, a dialog box is opened for selecting one.

`Save session as`: saves all current projects into a file chosen from a dialog box.

`Load Session`: opens a previously saved session.

This fully resets the current session (see Section 8.1.3).

`Exit Frama-C`: exits Frama-C without saving.

The project menu displays the existing projects, allowing you to set the current one. You can also perform miscellaneous operations over projects (creating from scratch, duplicating, renaming, removing, saving, *etc*).

The analyses menu provides items for configuring and running plug-ins.

- `Configure and run analyses`: opens the dialog box shown in Figure 9.2, that allows setting Frama-C parameters and re-running analyses.

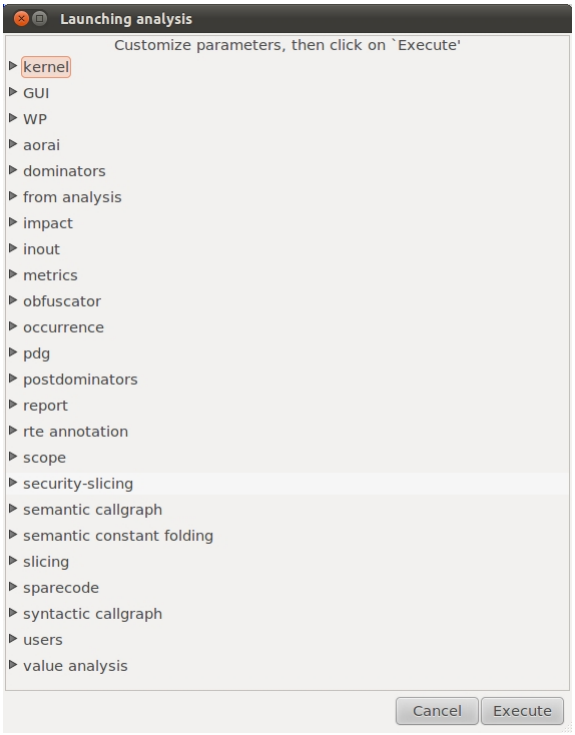

Figure 9.2: The Analysis Configuration Window

- `Compile and run an ocaml script`: allows you to run an OCaml file as a plug-in (in a way similar to the option `-load-script`, see Section 4.4).
- `Load and run an ocaml module`: allows you to run a pre-compiled OCaml file as a plug-in (in a way similar to the option `-load-module`, see Section 4.4).
- Other items are plug-in specific.

The debug menu is only visible in debugging mode and provides access to tools for helping to debug Frama-C and their plug-ins.

The help menu provides help items.

9.3 Tool Bar

The tool bar offers a more accessible access to some frequently used functions of the menu bar. Currently, the available buttons are, from left to right:

- The `Exit` button, that exits Frama-C.
- Four buttons `Source files`, `Reparse`, `Load Session` and `Save session`, equivalent to the corresponding entries in the `File` menu.
- Two navigation buttons, `Back` and `Forward`. They can be used to move within the history of the functions that have been viewed.
- The `Analyses` button, equivalent to the one in the `Analyses` menu.
- A `Stop` button, which halts the running analyses and restores Frama-C to its last known valid configuration.

Chapter 10

Reports

An execution of Frama-C outputs many messages, warnings and various property statuses in textual form. The Graphical User Interface (see Chapter 9) is a very good place to visualize all these results, but there are no synthetic results and integration with development environments can be difficult.

The `report` plug-in, provided by default with the Frama-C platform, is designed for this purpose. It provides the following features, which we detail in turn:

- Printing a summary of property consolidated statuses;
- Exporting a CSV file of property consolidated statuses;
- Filtering and classifying warnings, errors and property consolidated statuses, based on user-defined rules in JSON format;
- Output the above classification in JSON format;
- Make Frama-C exit with a non-null status code on some classified warning or error.

10.1 Reporting on Property Statuses

The following options of the `report` plug-in are available for reporting on consolidated property statuses:

`-report` Displays a summary of property statuses in textual form. The output is structured by functions and behaviors. The details of which plug-ins participate into the consolidation of each property status is also provided. This report is designed to be human-readable, although it can be verbose, subject to changes, and not suitable for integration with other tools.

`-report-print-properties` Also print the properties definition (in ACSL form) within the report.

`-report-(no)-proven` If not set, filter out the proven properties from the report.

`-report-(no)-specialized` If not set, filter out the properties that are auxiliary instances of other properties, like the preconditions of a function at its call sites.

`-report-untried` If set, also include in the report the properties that have not been tried.

10.2 Exporting to CSV

The consolidated property status database can be exported to a `CSV` file, eg. for an easy import into Excel. To use this feature, simply use the following option of the `report` plug-in:

`-report-csv <file>.csv` Output the report in CSV format into the specified file.

Notice than it is *not* necessary to set `-report` option to use this feature. It is highly recommended to use this option in combination with the following other standard Frama-C options:

```
> frama-c ... -then -no-unicode -report-csv <file>.csv
```

The format of the output CSV file is a collection of property consolidated statuses, with one property per line. Each line consists of a collection of TAB (ascii `0x0A`) separated columns. The first line of the report contains the title of the columns, as follows:

```
> head -1 <file>.csv
directory     file   line   function      property kind  status  property
```

10.3 Classification

We denote by *event* any warning, error and (finally consolidated) property status emitted by any plug-in during the execution of Frama-C. We introduce the notion of *event classification* as follows:

- An event can be assigned a *class*, identified by a name;
- The event is associated to a location (*file,line*) when available;
- It can be reformulated with a *title* and an extended *description*;
- An event may trigger an *action* when it is detected.

The classification of events is defined by the user through *classification rules* which are provided to the `report` plug-in via configuration files in JSON format. A typical invocation of Frama-C with classification has the following structure:

```
> frama-c -report-rules <file>.json ...other plugins... -then -report-classify
```

The collection of events starts once the classification rules have been loaded. Finally, a classification report is build. There are various options to tune the classification process and the reporting output. See section 10.3.6 for details.

Remark: it is possible to emit different classification reports successively from the command line. At each `-report-classify`, the pool of collected events is flushed and will not be included in subsequent reports.

10.3.1 Action

Classified events can trigger one of the following actions:

`SKIP` the event is filtered out and not included in the final report;

`INFO` the event is kept for user information;

`REVIEW` the event shall be carefully read by the user, it contains verifications to be performed by hand to guarantee the soundness of the provided results;

`ERROR` all the results shall be considered wrong due to improper use of the tool.

By default, Frama-C warnings shall trigger a `REVIEW` action and errors an `ERROR` one. However, it is possible to modify actions with classification rules.

10.3.2 Rules

Each classification rule is a JSON record following the structure of Figure 10.1. A file of classification rules shall contain one rule or an array of rules. Several files can be loaded. The first rule that applies to an event takes priority over subsequent ones.

An individual rule consists of one mandatory *pattern* field, and other optional fields. All the details are provided in the figure.

```
{
   // Optional Fields

   "classid": "<identifier>" ; // Default is "unclassified"
   "plugin": "<identifier>" ;  // Default is "kernel"
   "category": "<category>" ; // Default is "*" for all categories
   "title": "<free text>" ;    // Default is a short name of the event
   "descr": "<free text>" ;    // Default is the entire text of the event
   "action": "<SKIP|INFO|REVIEW|ERROR>" ; Default is 'REVIEW'

   // Mandatory Pattern Field (unless a category is specified)

   [ "error"         // Applies to error messages
   | "warning"       // Applies to warning messages
   | "unknown"       // Properties with « Unknown » status
   | "untried"       // Properties with « Not Tried » status
   | "invalid"       // Properties with « Invalid » status
   | "unproved"      // Properties with any status other than « Valid »
   ] : "<regexp>" ;
}
```

Figure 10.1: Classification Rule JSON Format

Plug-ins are identified by their short name, typically `"rte"` for the `Rtegen` plug-in, see `frama-c -plugins` for details.

Category filters apply to active warning categories[1] or warning categories promoted to errors. Use option `frama-c -[plugin]-warn-key help` to display the list of available warning categories for a given plugin. Category filters use the same syntax and meaning than warning control options: category `a:b` applies to all messages with category `a:b[:...]`, but *not* to messages with category `a` or `c[:...]`.

When using a category filter, the pattern field can be omitted to match all warning messages of this category.

10.3.3 Regular Expressions

OCaml regular expressions are accepted for `<regexp>` pattern fields.

Regular expressions are used to determine when a rule applies to an event. The rule matches a warning or error of the specified plug-in if the regular expression matches a *prefix* of the event message (excluding location and header). For property rules, the regular expression must match a *prefix* of the canonical property name, which have the following structure:

```
<Function><Behavior><Category><Names>
```

Each part of the canonical property name is optional and separated by a '_' character.

10.3.4 Reformulation

The optional fields `title` and `descr` of a classification rule allow for a *reformulation* of the event. Reformulations are plain text enriched by references to sub-parts of the matching regular expression of the event. Hence, `\*` stands for the entire event message, `\0` is the matched prefix, `\1..\9` refers to the corresponding sub-group of the OCaml regular expression. You can use `\n` to insert a new-line character and `\<c>` to escape character `<c>`.

10.3.5 JSON Output Format

The classification results can be exported to a single file in JSON format. It consists of an array of classified events, each one following the format given in Figure 10.2.

```
{
  "classid": "<ident>" ;
  "action": "<INFO|REVIEW|ERROR>" ;
  "title": "<free text>" ;
  "descr": "<free text>" ;
  [ "file": "<path>" ; "line": "<number>" ; ]
}
```

Figure 10.2: Classified Event JSON Format

[1] Warning categories are described in section 6.2.

10.3.6 Classification Options

`-report-classify` Report classification of all properties, errors and warnings (opposite option is -report-no-classify)

`-report-exit` Exit on error (set by default, opposite option is -report-no-exit)

`-report-output <*.json>` Output -report-classify in JSON format

`-report-output-errors <file>` Output number of errors to `<file>`

`-report-output-reviews <file>` Output number of reviews to `<file>`

`-report-output-unclassified <file>` Output number of unclassified to `<file>`

`-report-absolute-path` Force absolute path in JSON output. Normal behavior is to output relative filepath for files that are relative to the current working directory.

`-report-rules <*.json,...>` Configure the rules to apply for classification, and start monitoring.

`-report-status` Classify also property statuses (set by default, opposite option is -report-no-status)

`-report-stderr` Output detailed textual classification on stderr (opposite option is -report-no-stderr)

`-report-stdout` Force detailed textual classification on stdout (opposite option is -report-no-stdout)

`-report-unclassified-error <action>` Action to be taken on unclassified errors (default is: `'ERROR'`)

`-report-unclassified-invalid <action>` Action to be taken on invalid properties (default is: `'ERROR'`)

`-report-unclassified-unknown <action>` Action to be taken on unknown properties (default is: `'REVIEW'`)

`-report-unclassified-untried <action>` Action to be taken on untried properties (default is: `'SKIP'`)

`-report-unclassified-warning <action>` Action to be taken on unclassified warnings (default is: `'REVIEW'`)

Chapter 11

Variadic Plug-in

This chapter briefly presents the `Variadic` plug-in, which performs the translation of calls to variadic functions into calls to semantically equivalent, but non-variadic functions.

Variadic functions

Variadic functions accept a variable number of arguments, indicated in their prototype by an ellipsis (...) after a set of fixed arguments.

Some functions in the C standard library are variadic, in particular formatted input/output functions, such as `printf`/`scanf`. Due to the dynamic nature of their arguments, such functions present additional challenges to code analysis. The `Variadic` helps dealing with some of these challenges, reducing or eliminating the need for plug-ins to have to deal with these special cases.

11.1 Translating variadic function calls

ACSL does not allow the direct specification of variadic functions: variadic arguments have no name and no statically known type. The `Variadic` plug-in performs a semantically-preserving translation of calls to such functions, replacing them with non-variadic calls.

For instance, consider the following user-defined variadic function `sum`, whose first argument `n` is the number of elements to be added, and the remaining `n` arguments are the values themselves:

```
#include <stdarg.h> // for va_* macros
int sum(unsigned n, ...) {
  int ret = 0;
  va_list list;
  va_start(list, n);
  for(int i = 0; i < n; i++){
    ret += va_arg(list, int);
  }
  va_end(list);
  return ret;
}
```

```
int main(){
  return sum(5, 6, 9, 14, 12, 1);
}
```

Since `Variadic` is enabled by default, running Frama-C on this code will activate the variadic translation. The main differences in the translated code are:

- the prototype of `sum` becomes `int sum(unsigned n, void * const *__va_params);`
- the call to `sum` is converted into:

```
int sum_ret; // temporary storing the return value
{
  int __va_arg0 = 6;
  int __va_arg1 = 9;
  int __va_arg2 = 14;
  int __va_arg3 = 12;
  int __va_arg4 = 1;
  void *__va_args[5] = {& __va_arg0, & __va_arg1,
                        & __va_arg2, & __va_arg3, & __va_arg4};
  sum_ret = sum(5,(void * const *)(__va_args));
}
```

This translation is similar to the relation between functions such as `printf` and `vprintf`, where the former accepts a variable number of arguments, while the latter accepts a single `va_list` argument.

11.2 Automatic generation of specifications for libc functions

The most common use case of variadic functions are the ubiquitous `printf`/`scanf`, but a few other functions in the standard C library are variadic, such as `open` and `fcntl`. The former are entirely dependent on the *format* string, which can have any shape, while the latter are limited to a fixed set of possible argument numbers and types. In both cases, it is possible to specialize the function call and generate an ACSL specification that (1) performs some checks for undefined behaviors (*e.g.* that the argument given to a `%d` format is a `signed int`), and (2) ensures postconditions about the return value and modified arguments (namely for `scanf`). The `Variadic` plug-in generates such specifications whenever possible.

Note that not all calls have their specification automatically generated; in particular, calls to formatted input/output functions using non-static format strings are not handled, such as the following one:

```
printf(n != 1 ? "%d errors" : "%d error", n_errors);
```

In this case, the variadic translation is performed, but the function call is not specialized, and no specification is automatically generated.

11.3 Usage

11.3.1 Main options

By default, `Variadic` is enabled and runs after parsing. A few options are available to modify this behavior:

`-variadic-no-translation` : disables the translation performed by the plug-in; to be used in case it interferes with some other plug-in or analysis.

`-variadic-no-strict` : disables warnings about non-portable implicit casts in the calls of standard variadic functions, *i.e.* casts between distinct integral types which have the same size and signedness.

11.3.2 Similar diagnostics by other tools

Some of the issues detected by `Variadic`, namely some kinds of incompatible arguments in formatted input/output functions, are also detected by compilers such as GCC and Clang, albeit such diagnostics are rarely enabled.

In particular, GCC's option `-Wformat-signedness` (available from GCC 5) reports some issues with signed format specifiers and unsigned arguments, and vice-versa, in cases such as the following:

```
printf("%u", -1);
```

Clang's option `-Wformat-pedantic` (available at least since Clang 4.0, possibly earlier) also enables some extra diagnostics:

```
printf("%p", "string");
```

```
warning: format specifies type 'void *' but the argument has type 'char *'.
```

Note that no single tool is currently able to emit all diagnostics emitted by the other two.

11.3.3 Common causes of warnings in formatted input/output functions

Many C code bases which make use of formatted input/output functions do not specify all of them in a way that is strictly conformant to the C standard. This may result in a rather large number of warnings emitted by `Variadic`, not all of them immediately obvious.

It is however important to remember that C99, §7.19.6.1 states that:

> If a conversion specification is invalid, the behavior is undefined. If any argument is not the correct type for the corresponding conversion specification, the behavior is undefined.

CERT C lists this as Rule FIO47-C.

Some common types of discrepancies are listed below, with an explanation of their causes.

Usage of `%u` or `%x` for values of type `unsigned char` and `unsigned short`. The warning emitted by `Variadic` in this case will mention a *signed* type being cast to *unsigned*. Albeit counterintuitive, this is a consequence of the fact that default argument promotions take place for variadic arguments, and thus `unsigned char` and `unsigned short` arguments are promoted to (signed) `int`, which is incompatible with `%u` and `%x`.

To avoid the warning, use the appropriate length modifiers: `hh` for `char` and `h` for `short`.

Usage of `%o`, `%x` and `%X` to print signed values. The standard specifies that all of these modifiers expect unsigned arguments. A cast to the corresponding unsigned type must therefore be present.

11.3.4 Pretty-printing translated code

The output produced by `Variadic`, in particular when using `va_*` macros (such as `va_list`), is not guaranteed to be parsable, unless option `-print-libc` is enabled.

Chapter 12

Analysis Scripts

This chapter describes some tools and scripts shipped with Frama-C to help users setup and run analyses on large code bases. These scripts can also help dealing with unfamiliar code and automating analyses.

12.1 Requirements

These analysis scripts are chiefly based on the following tools:

Python : most scripts are written using Python 3. Some scripts require features from specific Python versions, and perform version checks when starting.

GNU Make : the main workflow for analysis scripts consists in using GNU Make (4.0 or newer) to compute dependencies and cache intermediate results.

Bash : some scripts are written in Bash.

Besides those, a few tools are occasionally required by the scripts, such as Perl and GNU Parallel.

12.2 Usage

Most scripts are accessible via the `frama-c-script` command, installed along with Frama-C. Running this command without any arguments prints the list of commands, with a brief description of each of them. Some extra scripts are available by directly running them; in both cases, the actual scripts themselves are installed in Frama-C's `share` directory, underneath `analysis-scripts`.

12.2.1 General Framework

Note: while the analysis scripts are intended for usage in a wide variety of scenarios with different plug-ins, they currently focus on analyses with the Eva plug-in only.

The main usage mode of `analysis-scripts` consists in creating a Makefile dedicated to the analysis of a C code base.

This Makefile has three main purposes:

1. To separate the main analysis steps, saving partial results and logging output messages;
2. To avoid recomputing unnecessary data when modifying analysis-specific options;
3. To document analysis options and improve replayability, e.g. when iteratively fine-tuning the analysis in order to improve its results.

The intended usage is as follows:

1. The user identifies a C code base on which they would like to run Frama-C;
2. The user runs a script to interactively fill a template for the analysis, with the list of source files and required parameters (architecture, preprocessing flags, main function);
3. The user edits and runs the generated Makefile, adjusting the analysis as needed and re-running `make`.

Ideally, after modifying the source code or re-parametrizing the analysis, re-running `make` should be enough to obtain a new result.

Section 12.3 details usage of the Makefile and presents an illustrative diagram.

12.2.2 Necessary Build Information

The command `frama-c-script make-template` can be used to generate the Makefile from a template. The user must fill in the following information, required for running an Eva analysis:

machdep : architectural information about system where the code will run: integer type sizes, compiler, OS, etc. See section 5.4 for more details.

preprocessing flags : options given to the C preprocessor, mainly macros (`-D`) and include directories (`-I`).

list of sources : the actual list of source files that make a logical unit (e.g. a test case or a whole program), without duplicate function definitions.

main function : the function where the analysis will start; it is often `main`, but not always. (*Note: Frama-C itself thes not require a `main` function, but plug-ins such as `Eva` do.*)

A project without this information is incomplete; an alternative workflow is then necessary. The next section presents some possibilities to retrieve such information.

12.2.3 Possible Workflows in the Absence of Build Information

It is sometimes the case that the Frama-C user is not the developer of the code under analysis, and does not have full build information about it; or the code contains non-portable features or missing libraries which prevent Frama-C from parsing it. In such cases, these analysis scripts provide two alternative workflows, depending on how one prefers to choose their source files: *one at a time* or *all-in*, described below.

One at a time

In this workflow, the user starts from the entry point of the analysis: typically the `main` function of a program or a test case. Only the file defining that function is included at first. Then, using `make-wrapper` (described in section 12.4), the user iteratively adds new sources as needed, so that all function definitions are included.

- Advantages: higher code coverage; faster preprocessing/parsing; and avoids including possibly unrelated files (e.g. for an alternative OS/architecture).
- Drawbacks: the iterative approach recomputes the analysis several times; also, it may miss files containing global initializers, which are not flagged as missing.

All-in

In this workflow, the user adds *all* source files to the analysis, and if necessary removes some of them, e.g. when there are conflicting definitions, such as when multiple test files define a `main` function.

- Advantages: optimistic approach; may not require any extra iterations, if everything is defined, and only once. Does not miss global initializers, and may find issues in code which is not reachable (e.g. syntax-related warnings).
- Drawbacks: preprocesses and parses several files which may end up never being used; smaller code coverage; if parsing fails, it may be harder to find the cause (especially if due to unnecessary sources).

12.2.4 Using a JSON Compilation Database (JCDB)

Independently of the chosen workflow, some partial information can be retrieved when CMake or Makefile scripts are available for compiling the sources. They allow the production of a JSON Compilation Database (`compile_commands.json`, called JCDB for short; see related option in section 5.2). This leads to a different workflow:

1. Run CMake with the flag `-DCMAKE_EXPORT_COMPILE_COMMANDS=1`, or install Build EAR (https://github.com/rizsotto/Bear) and run `bear make <targets>` instead of `make <targets>`. This will create a `compile_commands.json` file.
2. Run `frama-c-script list-files`. A list of the compiled files, along with files defining a `main` function, will be presented.
3. Run `frama-c-script make-template` to create a template for Frama-C/Eva. Answer "yes" when asked about using the `compile_commands.json` file.

Ideally, the above approach should result in a working template. In practice, however, the compilation database may include extraneous sources (used to compile other files than the target object) and duplicate flags (e.g. when compiling the same source for different binary targets or test cases). Manual intervention may be necessary.

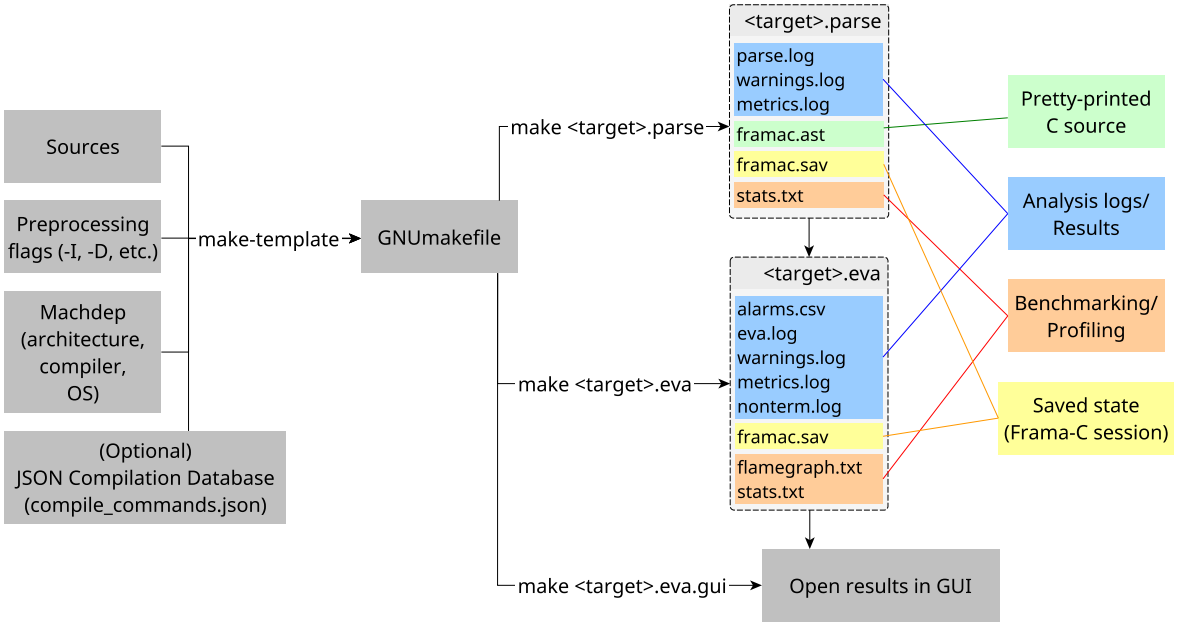

Figure 12.1: Overview of the *analysis-scripts* workflow: the inputs on the left produce a GNUmakefile which is then used for parsing, analyzing and visualizing results.

12.3 Using the generated Makefile

The generated Makefile can be used to run one or several analyses. The diagram in Fig. 12.1 summarizes its usage. Makefile targets and outputs are detailed in this section.

Each analysis is defined in a target, written by the user as follows:

```
<target>.parse:  file1.c file2.c ...
```

That is, the target name (chosen by the user), suffixed with `.parse`, is defined as depending on each of its source files. Changes to any of these sources will trigger a recomputation of the AST.

Note: the target name itself *cannot* contain slashes or dots. See also the *Technical Notes* section about some current limitations.

Then, for each `.parse` target, a corresponding `.eva` target needs to be added to the `TARGETS` variable in the Makefile. This will run Eva on the test case.

Each `.eva` target depends on its corresponding `.parse` target; if the sources change, the analysis must take into account the new AST.

12.3.1 Important Variables

Several Makefile variables are available to customize Frama-C; the main ones are presented below.

TARGETS : as mentioned before, must contain the list of targets, suffixed with `.eva`.

CPPFLAGS : preprocessing options, passed to Frama-C inside option `-cpp-extra-args`, when parsing the sources.

FCFLAGS : extra command-line options given to Frama-C when parsing the sources and when running analyses. Typically, the options given to the Frama-C kernel.

EVAFLAGS : extra command-line options given to Eva when running its analyses.

These variables are defined globally, but they can also be customized for each target; for instance, if a given target `t1` has a main function `test1` and requires a global macro `-DTEST1`, but target `t2`'s main function is `test2` and it requires `-DTEST2`, you can locally modify `FCFLAGS` and `CPPFLAGS` as follows:

```
t1.parse: FCFLAGS += -main test1
t1.parse: CPPFLAGS += -DTEST1
t2.parse: FCFLAGS += -main test2
t2.parse: CPPFLAGS += -DTEST2
```

12.3.2 Predefined targets

The predefined targets below are the *raison d'être* of the generated Makefile; they speed up analyses, provide self-documentation, and enable quick iterations during parametrization of the analysis.

all (default target) : the default target simply calls `<target>.eva`, for each `<target>` added to variable `TARGETS`. Does nothing once the analysis is finished and saved.

<target>.parse : runs Frama-C on the specified target, preprocessing and parsing its source files. Produces a directory `<target>.parse` containing several logs, a pretty-printed version of the parsed code, and a Frama-C session file (`framac.sav`) to be loaded in the GUI or by other analyses. Does nothing if parsing already happened.

<target>.eva : loads the parsed result (from the `.parse` target) and runs the Eva plug-in, with the options given in `EVAFLAGS`. If the analysis succeeds, produces a directory `<target>.eva` with the analysis results and a saved session file. Also creates a timestamped version of `<target>.eva`, to enable future comparisons between different parametrizations. The non-timestamped version corresponds to the latest (successful) analysis. If the analysis fails, tries to save a partial result in `<target>.eva.error` (when possible).

<target>.eva.gui : loads the result of the corresponding `<target>.eva` session and opens it in the GUI. This allows inspecting the results of Eva. This target always opens the GUI, even when no changes have happened.

clean : removes all results produced by the `.parse` and `.eva` targets.

12.4 Script Descriptions

The most useful commands are described below. Run `frama-c-script help` for more details and optional arguments.

make-template : creates the initial Makefile, based on a template. This command creates a file named `GNUmakefile` with some hardcoded sections, some filled in interactively by the user, and comments indicating which parts may need change. Once created, it enables the general workflow mentioned earlier.

make-wrapper <target> <args> : calls `make <target> <args>` with a special wrapper: when running Eva, upon encountering one of a few known error messages, suggests some actions on how to proceed. For instance, if a missing function definition is encountered when analyzing the code with Eva, the wrapper will look for its definition and, if found, suggest that its source code be added to the analysis. This script is meant to be used with the *one at a time* workflow describe in section 12.2.3.

find-fun <fun> : looks for possible declarations and definitions of function `<fun>`. Uses a heuristic that does not depend on Frama-C being able to parse the sources. Useful to find entry points and missing includes.

Other commands, only useful in a few cases, are described below.

configure <machdep> : runs a `configure` script (based on Autoconf) with some settings to emulate a more portable system, removing optional code features that could prevent Frama-C from parsing the sources. Currently still depends partially on the host system, so many features are not disabled.

make-path (for Frama-C developers): to be used when Frama-C is not installed in the PATH; adds a `frama-c-path.mk` file that is used by the Makefile generated by `make-template`.

flamegraph : opens a *flamegraph*[1] to visualize which functions take most of the time during analysis with Eva.

summary : for monitoring the progression of multiple analyses defined in a single Makefile. Presents a summary of the analyses when done. Mostly useful for benchmarking or when dealing with several test cases.

The following commands require a JSON Compilation Database.

list-files : lists all files in the given JCDB. Useful for filling out the Makefile template when running `make-template`.

normalize-jcdb : converts absolute paths inside a `compile_commands.json` file into relative paths (w.r.t. PWD) when possible. Used to allow moving/versioning the directory containing the JCDB file.

Finally, there is the following script, which is *not* available as a command in `frama-c-script`, since its usage scenario is very different. It is available at `$FRAMAC_SHARE/analysis-scripts/creduce.sh`.

creduce.sh : A script to help running the C-Reduce[2] tool to minify C programs causing crashes in Frama-C; useful e.g. when submitting a bug report to Frama-C, without needing to submit potentially confidential data. The script contains extensive comments about its usage. It is also described in a post[3] from the Frama-C blog.

To use the `creduce.sh` script, you need to have the C-Reduce tool installed in your path or in environment variable `CREDUCE`.

[1] See https://github.com/brendangregg/FlameGraph for details about flamegraphs.
[2] See https://embed.cs.utah.edu/creduce for more details.
[3] *Debugging Frama-C analyses: better privacy with C-Reduce*, at https://pub.frama-c.com/scripts/usability/2020/04/02/creduce.html.

12.5 Practical Examples: Open Source Case Studies

The *open-source-case-studies* Git repository (OSCS for short), available at `https://git.frama-c.com/pub/open-source-case-studies`, contains several open-source C code bases parametrized with the help of analysis scripts. Each case study has its own directory, with a `GNUmakefile` defining one or more analysis targets.

Due to the variety of test cases, OSCS provide practical usage examples of the `GNUmakefile` described in this chapter. They are periodically synchronized w.r.t. the public Frama-C repository (daily snapshots), so they may contain features not yet available in the major Frama-C releases. A few case studies may also contain legacy features which are no longer used; but overall, they provide useful examples and allow the user to tweak analysis parameters to test their effects.

12.6 Technical Notes

This section lists known issues and technical details which may help users understand some unintuitive behaviors.

Changes to header files do not trigger a new parsing/analysis. Currently, changes to included files (e.g. headers) are *not* tracked by the generated Makefile and may require running `make` with `-B` (to force recomputation of dependencies), or running `make clean` before re-running `make`.

Why is the generated Makefile called `GNUmakefile`? GNU Make, by default, searches for a file named `GNUmakefile` before searching for a `Makefile`. Thus, running `make` without arguments results in running the Makefile generated by `make-template`. You can rename it to `framac.mk` or something else, and then run it via `make -f framac.mk <targets>`.

Most scripts are heuristics-based and offer no correctness/completeness guarantees. In order to handle files *before* the source preparation step is complete (that is, before Frama-C is able to parse the sources into a complete AST), most commands use scripts based on syntactic heuristics, which were found to work well in practice but are easily disturbed by syntactic changes (e.g. whitespaces).

Most commands are experimental. These analysis scripts are a recent addition and subject to changes. They are provided on a best-effort basis.

Chapter 13

Reporting Errors

If Frama-C crashes or behaves abnormally, you are invited to report an issue *via* the Frama-C Gitlab repository, located at https://git.frama-c.com. The *New Issue* page (https://git.frama-c.com/pub/frama-c/issues/new) allows creating a new report, but you will need an account.

Unless you have an account provided by the Frama-C team, you need to sign in using a Github account, as shown in Figure 13.1.

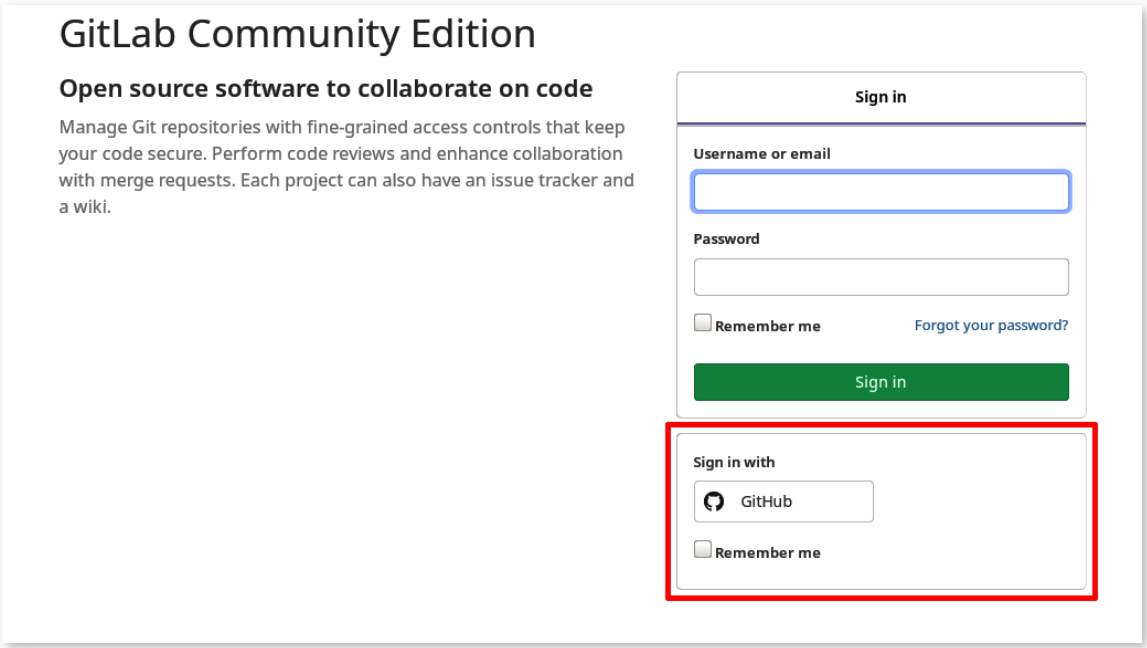

Figure 13.1: The Frama-C Gitlab login page. Note that no direct account creation is possible; you need to sign in via Github unless the Frama-C team provided you an account.

When creating a new issue, choose the `bug_report` template next to *Title*, then enter the title and fill the template.

Bug reports can be marked as public or confidential. Public bug reports can be read by anyone and may be indexed by search engines. Confidential bug reports are only shown to Frama-C developers.

Reporting a new issue opens a webpage similar to the one shown in Figure 13.2.

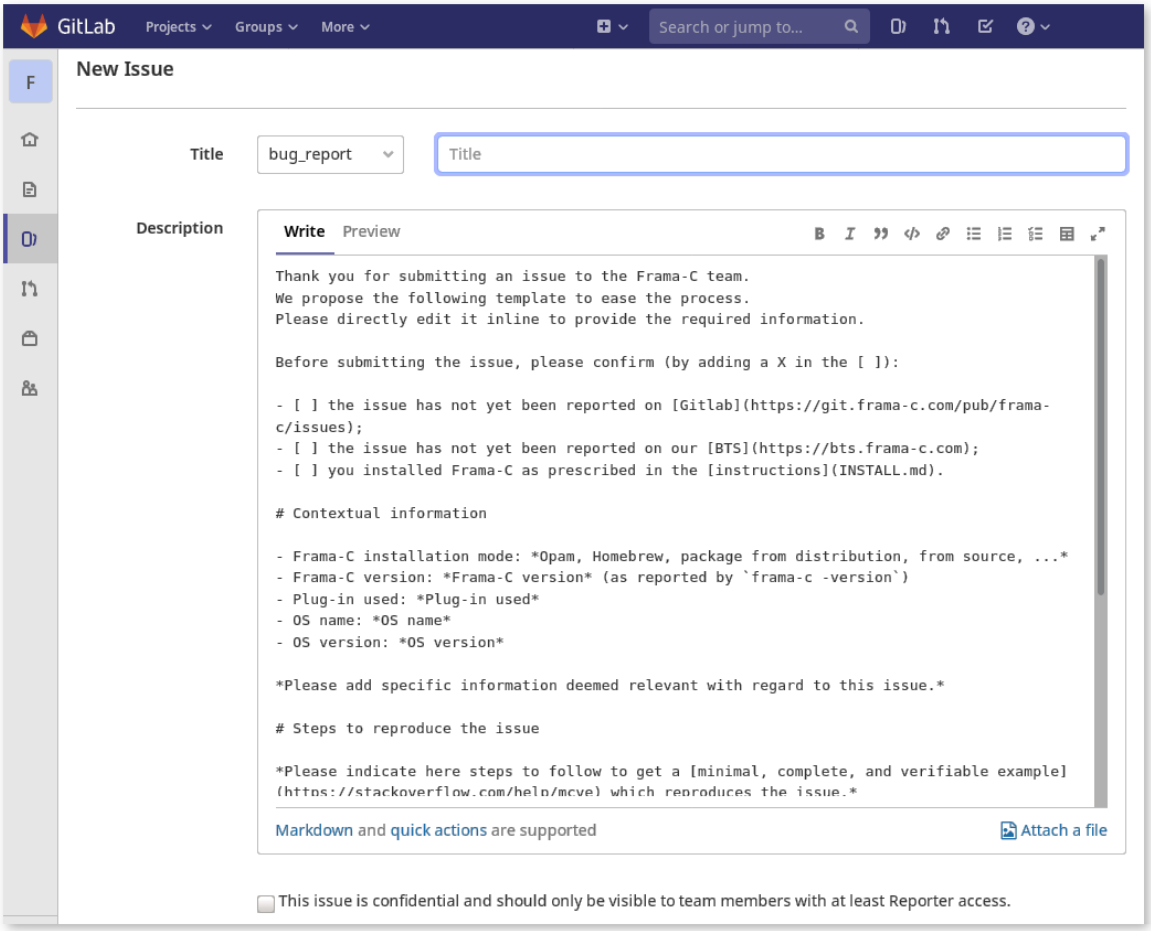

Figure 13.2: The Gitlab new issue page, with the `bug_report` template. The checkbox at the bottom enables marking the issue as private, so that only Frama-C developers can see it.

Please fill the template as precisely as possible, *in English*[1], which helps the Frama-C team more quickly understand, reproduce and respond to the issue. The form uses Markdown syntax and you can attach source files and screenshots to the issue.

Replies and updates concerning your issue are sent by e-mail by Gitlab.

[1] French is also a possible language choice for private entries.

Appendix A

Changes

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

21.0 (Scandium)

- **Preparing the Sources:** added option `-cpp-extra-args-per-file`.
- **Customizing Analyzers:** added options `-warn-invalid-pointer` and `-warn-pointer-downcast`

20.0 (Calcium)

- **Normalizing the Source Code:** added options `-keep-unused-types` and its opposite, `-remove-unused-types`.

18.0 (Argon)

- **Feedback Options:** change options governing status of warning categories
- **Normalizing the Source Code:** added category `@inline` to option `-inline-calls`, and added option `-remove-inlined`.
- **Customizing Analyzers:** added options `-warn-left-shift-negative`, `-warn-right-shift-negative` and `-warn-invalid-bool`.

Chlorine-20180501

- **Normalizing the Source Code:** added option `-inline-calls`.
- **Preparing the Sources:** documentation of macros that can be defined to customize the standard C library.

- **Preparing the Sources:** deprecated option `-implicit-function-declaration` (superseded by warning categories; equivalent to `-kernel-warn-{key,abort,feedback} typing:implicit-function-declaration`).
- **Setting Up Plug-ins:** remove obsolete references to static plug-ins and `-with-all-static` configure option.
- **Feedback Options:** introduction of warning categories and statuses.
- **Customizing Analyzers:** added option `-warn-special-float`.
- **Preparing the Sources:** added option `-json-compilation-database`.
- **Reports:** new chapter documenting reporting facilities.
- **Variadic Plug-in:** new chapter documenting the Variadic plug-in.

Sulfur-20171101

- **Preparing the Sources:** removed option `-force-rl-arg-eval`.
- **Normalizing the Source Code:** added section about compiler and language extensions (Section 5.6).
- **Normalizing the Source Code:** removed option `-custom-annot-char`.

Phosphorus-20170501

- **Getting Started:** Zarith package is now mandatory.
- **Setting Up Plug-ins:** added option `-autoload-plugins`.
- **Preparing the Sources:** renamed option `-cpp-gnu-like` to `-cpp-frama-c-compliant`.
- **Getting Started:** document new bash completion script.
- **Getting Started:** added option `-print-libc`.

Silicon-20161101

- **Getting Started:** OCaml version greater or equal than 4.05.0 is required
- **Normalizing the Source Code:** New option `-c11`

Aluminium-20160501

- **Getting Started:** document new option `-then-replace`.
- **Getting Started:** document new option `-set-project-as-default`.
- **Project:** document new option `-remove-projects`.
- **Getting Started:** document new option `-<plug-in shortname>-log`.
- **Normalizing the Source Code:** document new options `-asm-contracts` and `-asm-contracts-auto-validate`
- **Graphical User Interface:** Option `-collect-messages` is active by default, and cannot be deactivated.

Magnesium-20151001

- **Getting Started:** support is not guaranteed for OCaml versions below 4.x.
- **Getting Started:** the recommended installation method now consists in using the Frama-C OPAM package.
- **Normalizing the Source Code:** option `-pp-annot` is now active by default.
- **Normalizing the Source Code:** added section about macros predefined by Frama-C (Section 5.5).
- **Normalizing the Source Code:** document new option `-custom-annot-char` (Section 5.4)
- **Normalizing the Source Code:** document handling of new file suffix `.ci` (Section 5.2)
- **Preparing the Sources:** option `-warn-undefined-callee` changed to `-implicit-function-declaration warn`.
- **Setting Up Plug-ins:** removed option `-dynlink`.

Sodium-20150201

- **Normalizing the Source Code:** new options `-initialized-padding-locals` and `-simplify-trivial-loops`.
- **Pre-processing the Source Files:** new options `-cpp-gnu-like` and `-frama-c-stdlib`.
- **Customizing Analyzers:** new options `-add-symbolic-path` and `-permissive`.
- **Getting Started:** document options containing several values (*aka* set and map).
- **Getting Started:** improve documentation of options.
- **Getting Started:** document new option `-then-last`.
- **Getting Started:** document new option `-tty`.

Neon-20140*01

- **Getting Started:** fixes list of requirements for compiling Frama-C.
- **Preparing the Sources:** new option `-aggressive-merging`
- **General Kernel Services:** change the default name of the journal.
- **Getting Started:** new options `-config` and `-<plug-in shortname>-config`, as well as new environment variable `FRAMAC_CONFIG`.
- **Getting Started:** new options `-session` and `-<plug-in shortname>-session`, as well as new environment variable `FRAMAC_SESSION`.
- **Getting Started:** document option `-unicode`.
- **General Kernel Services:** clarify when saving is done.

Fluorine-20130*01

- **Getting Started:** update installation requirements.
- **Customizing Analyzers:** document the following new options:
  - `-warn-signed-overflow`,
  - `-warn-unsigned-overflow`,
  - `-warn-signed-downcast`, and
  - `-warn-unsigned-downcast`.
- **Preparing the Sources:** document new option `-enums`

Oxygen-20120901

- **Analysis Option:** better documentation of `-unspecified-access`
- **Preparing the Sources:** better documentation of `-pp-annot`
- **Preparing the Sources:** pragma `UNROLL_LOOP` is deprecated in favor of `UNROLL`
- **Preparing the Sources:** document new normalization options `-warn-decimal-float`, `-warn-undeclared-callee` and `-keep-unused-specified-functions`
- **General Kernel Services:** document special cases of saving and journalisation.
- **Getting Started:** optional Zarith package.
- **Getting Started:** new option `-<plug-in shortname>-share`.

Nitrogen-20111001

- **Overview:** report on Frama-C' usage as an educational tool.
- **Getting Started:** exit status 127 is now 125 (127 and 126 are reserved by POSIX).
- **Getting Started:** update options for controlling display of floating-point numbers
- **Preparing the sources:** document generation of `assigns` clause for function prototypes without body and proper specification
- **Property Statuses:** new chapter to document property statuses.
- **GUI:** document new interface elements.

Carbon-20110201

- **Getting Started:** exit status 5 is now 127; new exit status 5 and 6.
- **GUI:** document new options `-collect-messages`.

Carbon-20101201

- **Getting Started:** document new options `-then` and `-then-on`.
- **Getting Started:** option `-obfuscate` is no more a kernel option since the obfuscator is now a plug-in.

Boron-20100401

- **Preparing the Sources:** document usage of the C standard library delivered with Frama-C
- **Graphical User Interface:** simplified and updated according to the new implementation
- **Getting Started:** document environment variables altogether
- **Getting Started:** document all the ways to getting help
- **Getting Started:** OcamlGraph 1.4 instead 1.3 will be used if previously installed
- **Getting Started:** GtkSourceView 2.x instead of 1.x is now required for building the GUI
- **Getting Started:** documentation of the option `-float-digits`
- **Preparing the Sources:** documentation of the option `-continue-annot-error`
- **Using plug-ins:** new option `-dynlink`
- **Journalisation:** a journal is generated only whenever Frama-C crashes on the GUI

- **Configure:** new option `--with-no-plugin`
- **Configure:** option `--with-all-static` set by default when native dynamic loading is not available

Beryllium-20090902

- First public release

Bibliography

[1] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language. Version 1.8 — Frama-C Oxygen implementation.*, March 2014.

[2] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language. Version 1.8*, March 2014.

[3] Patrick Baudin and Anne Pacalet. Slicing plug-in. http://frama-c.com/slicing.html.

[4] Loïc Correnson, Zaynah Dargaye, and Anne Pacalet. *Frama-C's WP plug-in*, February 2015. http://frama-c.com/download/frama-c-wp-manual.pdf.

[5] Loïc Correnson and Julien Signoles. Combining Analysis for C Program Verification. In *Formal Methods for Industrial Critical Systems (FMICS)*, August 2012.

[6] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C, A software Analysis Perspective. In *Software Engineering and Formal Methods (SEFM)*, October 2012.

[7] Pascal Cuoq, Boris Yakobowski, and Virgile Prevosto. *Frama-C's value analysis plug-in*, February 2015. http://frama-c.com/download/frama-c-eva-manual.pdf.

[8] M. Delahaye, N. Kosmatov, and J. Signoles. Common specification language for static and dynamic analysis of C programs. In *the 28th Annual ACM Symposium on Applied Computing (SAC)*, pages 1230–1235. ACM, March 2013.

[9] Philippe Herrmann and Julien Signoles. *Annotation Generation: Frama-C's RTE plug-in*, April 2013. http://frama-c.com/download/frama-c-rte-manual.pdf.

[10] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, pages 1–37, 2015. Extended version of [6].

[11] Julien Signoles. *Frama-C's E-ACSL Plug-in*, February 2015. http://frama-c.com/eacsl.html.

[12] Julien Signoles, Thibaud Antignac, Loïc Correnson, Matthieu Lemerre, and Virgile Prevosto. *Frama-C Plug-in Development Guide*, February 2015. http://frama-c.com/download/frama-c-plugin-development-guide.pdf.

[13] Nicolas Stouls and Virgile Prevosto. *Frama-C's Aoraï plug-in*, April 2013. http://frama-c.com/download/frama-c-aorai-manual.pdf.

List of Figures

Index