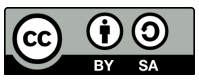


Frama-C User Manual

For Frama-C 34.0 beta (Selenium)

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





Work licensed under Creative Commons BY-SA licence
https://creativecommons.org/licenses/by-sa/4.0/

© 2009-2026 CEA-List

Contents

Foreword 6

1 **Introduction** 7

1.1 About this document . . . 7

1.2 Outline . . . 7

2 **Overview** 9

2.1 What is Frama-C? . . . 9

2.2 Frama-C as a Code Analysis Tool . . . 9

2.2.1 Frama-C as a Lightweight Semantic-Extractor Tool . . . 10

2.2.2 Frama-C for Formal Verification of Critical Software . . . 10

2.3 Frama-C as a Tool for C programs . . . 10

2.4 Frama-C as an Extensible Platform . . . 10

2.5 Frama-C as a Collaborative Platform . . . 11

2.6 Frama-C as a Development Platform . . . 11

2.7 Frama-C as an Educational Platform . . . 12

3 **Getting Started** 13

3.1 Installation . . . 13

3.2 One Framework, Several Executables . . . 13

3.3 Frama-C Command Line and General Options . . . 14

3.3.1 Getting Help . . . 14

3.3.2 Frama-C Configuration . . . 14

3.3.3 Options Outline . . . 15

3.3.4 Autocompletion for Options . . . 16

3.3.5 Splitting a Frama-C Execution into Several Steps . . . 16

3.3.6 Verbosity and Debugging Levels . . . 17

3.3.7 Copying Output to Files . . . 17

3.3.8 Terminal Capabilities . . . 18

3.3.9 Getting time . . . 18

3.3.10 Inputs and Outputs of Source Code . . . 18

3.3.11 Group Command Line Options in a File . . . 18

3.4 Environment Variables . . . 19
3.4.1 Sandbox mode . . . 19
3.4.2 FRAMAC_SESSION . . . 19
3.4.3 User directories . . . 20
3.5 Exit Status . . . 20

4 **Setting Up Plug-ins** 21
4.1 The Plug-in Taxonomy . . . 21
4.2 Installing External Plug-ins . . . 21
4.3 Loading Plug-ins . . . 21
4.3.1 Loading Single OCaml Files as Plug-ins . . . 22

5 **Preparing the Sources** 23
5.1 Overview of Source Processing in Frama-C . . . 23
5.2 Parsing Using a Mopsa-DB File . . . 23
5.3 Preprocessing Sources in a GCC-Compatible Environment . . . 24
5.3.1 Preprocessing Using a Compilation Database . . . 24
5.4 Preprocessing Sources, Full Details . . . 24
5.5 Merging the Source Code files . . . 26
5.6 Normalizing the Source Code . . . 26
5.7 Normalizing Contracts . . . 29
5.8 Incremental parsing . . . 30
5.9 Predefined macros . . . 30
5.10 Compiler and language extensions . . . 31
5.11 Standard library (libc) . . . 32
5.12 Testing the Source Code Preparation . . . 32

6 **ACSL Extensions** 33
6.1 Extension Syntaxes . . . 33
6.2 Handling Indirect Calls with calls . . . 33
6.3 Importing External Module Definitions . . . 33

7 **Platform-wide Analysis Options** 35
7.1 Entry Point . . . 35
7.2 Feedback Options . . . 35
7.3 Customizing Analyzers . . . 36
7.4 Performance Parameters . . . 39

8 **Property Statuses** 40
8.1 A Short Detour through Annotations . . . 40
8.2 Properties, and the Statuses Thereof . . . 40
8.3 Consolidating Property Statuses . . . 41

9 **General Kernel Services** 43
9.1 Projects . . . 43
9.1.1 Creating Projects . . . 43
9.1.2 Using Projects . . . 43
9.1.3 Saving and Loading Projects . . . 43
9.2 Dependencies between Analyses . . . 44

10 **Graphical User Interface** 46
10.1 Command-line options . . . 47

11 **Reports** 48
11.1 Reporting on Property Statuses . . . 48
11.2 Exporting to CSV . . . 49
11.3 Classification . . . 49
11.3.1 Action . . . 49
11.3.2 Rules . . . 50
11.3.3 Regular Expressions . . . 50
11.3.4 Reformulation . . . 51
11.3.5 JSON Output Format . . . 51
11.3.6 Classification Options . . . 51
11.4 SARIF Output via the Markdown Report Plug-in . . . 52
11.4.1 Prerequisites . . . 52
11.4.2 Generating a SARIF Report . . . 52

12 **Variadic Translation** 53
12.1 Translating variadic function calls . . . 53
12.2 Automatic generation of specifications for libc functions . . . 54
12.3 Usage . . . 54
12.3.1 Concrete Example . . . 54
12.3.2 Main options . . . 55
12.3.3 Similar diagnostics by other tools . . . 55
12.3.4 Common causes of warnings in formatted input/output functions . . . 56
12.3.5 Pretty-printing translated code . . . 56

13 **Analysis Scripts** 57
13.1 Requirements . . . 57
13.2 Usage . . . 57
13.2.1 General Framework . . . 57
13.2.2 Necessary Build Information . . . 58
13.2.3 Possible Workflows in the Absence of Build Information . . . 58
13.2.4 Using a JSON Compilation Database (JCDB) . . . 59
13.3 Using the generated Makefile, via `fcmake` . . . 59
13.3.1 Important Variables . . . 61
13.3.2 Predefined targets . . . 61
13.3.3 Adding new analyses and stages . . . 62
13.4 Script Descriptions . . . 62
13.5 Practical Examples: Open Source Case Studies . . . 64
13.6 Technical Notes . . . 64

14 **Compliance** 65
14.1 Unsupported C99 and C11 Features . . . 65
14.2 Frama-C Options Related to C Undefined Behaviors . . . 66
14.3 RTE categories and C Undefined Behaviors . . . 67
14.4 C Undefined Behaviors *not* handled by Frama-C . . . 68
14.5 Common Weakness Enumerations (CWEs) Reported and not Reported by Frama-C 68

15 **Reporting Errors** 75

A **Changes** 78

Bibliography 84

List of Figures 86

Index 87

Foreword

This is the user manual of Frama-C[1]. The content of this document corresponds to the version 34.0 beta (Selenium), released on November 13, 2026, of Frama-C.

Acknowledgements

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

ANR This project has received funding from the French ANR projects CAT (ANR-05-RNTL-00301) and U3CAT (08-SEGI-021-01) .

This project has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement N° 317753 (STANCE).
It has also received funding from the Horizon 2020 research and innovation programme, under grant agreements N° 731453 (VESSEDIA), N° 824231 (DECODER), N° 830892 (SPARTA), and N° 883242 (ENSURESEC) .

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

Introduction 1

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 (Eva, WP, E-ACSL, ...). Each of these analyses has its own specific documentation [7, 10, 16]. Furthermore, research papers [9, 12] 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 [15].

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 orange 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 preprocessor.

Chapter 7 gives some general options for parameterizing analyzers.

Chapter 8 touches on the topic of code properties, and their validation by the platform.

Chapter 9 introduces the general services offered by the platform.

Chapter 10 gives a detailed description of the graphical user interface of Frama-C.

Chapter 11 describes the `Report` plug-in, used for textual consolidation and export of warnings, errors and properties.

Chapter 12 presents the Variadic Translation component, used to help plug-ins handle code containing variadic functions, such as `printf` and `scanf`.

Chapter 13 details several scripts used to help setup and run analyses on large code bases.

Chapter 14 contains information related to compliance to standards and coding guidelines (ISO C, CERT, CWEs, etc).

Chapter 15 explains how to report errors *via* Frama-C's public Gitlab repository.

Overview 2

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 concurrency safety analysis (Mthread [17]), an enforcer of secure information flow (SecureFlow [1, 2]), or a set of tools for various test coverage criteria (LTest [13]), among many others.

2.2 Frama-C as a Code 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 code 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.

Frama-C also provides some *dynamic analysis*, via plug-ins such as E-ACSL [16], which performs *runtime verification*. It is often used as a complement to static analysis: when some properties cannot be proven statically, E-ACSL instruments the code so that, during execution, such properties are verified

and *enforced*: violations lead to alerts or immediate termination, preventing silent program corruption or malicious infiltration, and helping pinpoint the exact cause of a problem, instead of an indirect consequence much later.

For both static and dynamic analyses, Frama-C focuses on the *source code*. Plug-ins can help translate higher-level properties and specifications, or even provide front-ends to other languages; but, in the end, the strength of the platform is centered around the code and its 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*) [3]. 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[1]. C comments may contain ACSL annotations [3] used as specifications to be interpreted by Frama-C. The subset of ACSL currently interpreted in Frama-C is described in [4].

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 Eclipse IDE): 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

[1] Some parts of the C11 standard, as well as some common GCC extensions, are also supported.

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 verification (WP), runtime verification (E-ACSL), *etc.* Each of these analyses has its own specific documentation [10, 7, 16].

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 Frama-Clang plug-in [6], which provides an experimental front-end for C++ code; or MetAcsl [14], which allows specifying higher-level meta-properties; among others.

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 [3]. 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 [15].

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.

Getting Started 3

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, including 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 exact set of dependencies varies from release to release. They are listed as constraints in the `opam` file of the source distribution.

For more installation instructions, consider reading the `INSTALL.md` file of the source distribution. The main components necessary for compiling and running Frama-C are described below.

A C preprocessor is required for *using* Frama-C on C files. If you do not have any C preprocessor, you can only run Frama-C on already preprocessed `.i` files.

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

A Unix-like compilation environment with at least the tool `GNU make`[2] ≥ 4.0, as well as various POSIX commands, libraries and header files, is necessary for compiling Frama-C and its plug-ins.

The OCaml compiler [3] is required both for compiling Frama-C from source *and* for compiling additional plug-ins. Compatible OCaml versions are listed as constraints in the `opam` file.

Other components, such as OcamlGraph and Zarith, are listed in the `opam` file.

3.2 One Framework, Several Executables

The main executables installed by Frama-C are:

[1] http://opam.ocaml.org

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

[3] http://ocaml.org

- `frama-c`: batch version (command line);
- `frama-c-gui`: interactive version (graphical interface).

Batch version The `frama-c` binary can be used to perform most Frama-C analyses: from parsing the sources to running complex analyses, on-demand or as part of a continuous integration pipeline. Many analyses can display their output in simple text form, but some structured results (HTML, CSV or JSON) are also available. For instance, a SARIF (see Section 11.4) output based on JSON is can be produced by the Markdown Report plug-in. While the batch version is very powerful, it does not offer the visualization features of the interactive version.

Interactive version 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 10 for details). For instance, you can hover over an expression in the code and obtain immediate syntactic and semantic information about it; or use context menus for easy code navigation. However, the initial analysis setup (especially parsing) can be complex for some projects, and the batch version is better suited for providing error messages at this stage.

Both versions are complementary: the batch version is recommended for initial setup and analysis, while the interactive version is recommended for results visualization and interactive proofs.

Besides these two executables, Frama-C also provides a few other command line binaries:

- `frama-c-script`: contains several utilities related to source preparation, results visualization and analysis automation. Run it without arguments to obtain more details.

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`.

Finally, the option `-explain` can be used to obtain information about some specific options of the kernel or of any installed plug-ins: it prints a help message for each other option given on the command line.

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.

For a more thorough display of configuration-related data, in JSON format, use option `-print-config-json`. Note that the data output by this option is likely to change in future releases.

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
```

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`. 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* [10] 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.

The kernel option `-autocomplete` provides a text output listing all Frama-C options, so that autocompletion scripts may use it to provide completion. The Frama-C team welcomes improved and additional autocompletion scripts.

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`, [10]) with an unrolling level of 4 (option `-ulevel`, Section 5.6). 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 9.1) 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 follows:

```
$ 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 7.2 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 5.6). 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

3.3.11 Group Command Line Options in a File

The option `-commands-file <file>` allows one to group command line options in a file that can then be referenced from the command line or from other files. The syntax is the same as regular options from the command line but with one and exactly one option per line.

For instance a file `eva.txt` could regroup command line options specific to the Eva plugin:

```
-eva
-eva-precision 5
-eva-slevel 2000
```

It could then be re-used in different command lines:

```
$ frama-c file1.c -commands-file eva.txt
$ frama-c file2.c -commands-file eva.txt
```

Regular options and options from a commands file can be mixed and matched. For instance the following example is taken from section 3.3.5 but the call to Eva is replaced with a call to `-commands-file`:

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

Conversely, all those options can be put in a single file `config.txt` and have only one call to `-commands-file` in the Frama-C command:

```
-semantic-const-fold main
file.c
-then-on propagated
-commands-file eva.txt
-set-project-as-default
```

Note that we have one and only one option per line, and that we can use `-commands-file` inside of another commands file.

```
$ frama-c -commands-file config.txt
```

For convenience, it is possible to add comments by starting the line with a `#` in the configuration file and break long options on several lines by using `\`:

```
# Comments are ignored by Frama-C when parsing the configuration file
# and long options can be split by adding \ at the end of the line
-eva-domains equality, \
          gauges
# is equivalent to
-eva-domains equality,gauges
```

3.4 Environment Variables

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

3.4.1 Sandbox mode

Frama-C can operate in *sandbox mode* when required by a security policy. In this mode, any user-supplied custom commands are blocked from execution. Notably, modifying the preprocessor's command-line is not allowed.

To activate sandbox mode, set the `FRAMAC_SANDBOX` environment variable before running Frama-C. When enabled, Frama-C will verify that potentially unsafe parameters—such as `-cpp-command` or `-load-module`—are not used. These and other options considered unsafe in sandbox mode are marked as such in the Frama-C help output.

This mode is experimental, and some commands may still bypass the current checks.

3.4.2 `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`.

3.4.3 User directories

Frama-C provides facilities for sharing information between sessions using XDG-like directories for cache, config and state data. The default location of these directories is system-dependent.

- macOS
  - cache: `~/Library/Caches/frama-c`
  - config: `~/Application Support/frama-c/config`
  - state: `~/Application Support/frama-c/state`
- Unix (non macOS)
  - cache: `~/.cache/frama-c`
  - config: `~/.config/frama-c`
  - state: `~/.local/state/frama-c`
- Windows
  - cache: `%TEMP%/frama-c`
  - config: `%LOCALAPPDATA%/frama-c/config`
  - state: `%LOCALAPPDATA%/frama-c/state`

There are three ways to customize these locations for Frama-C:

- Frama-C options: `-cache` / `-config` / `-state`
- Frama-C variables: `FRAMAC_CACHE` / `FRAMAC_CONFIG` / `FRAMAC_STATE`
- XDG variables: `XDG_CACHE` / `XDG_CONFIG` / `XDG_STATE_HOME` (in each case, a `frama-c` subdirectory is created inside it)

If several are used at the same time for the same directory, the order of priority is:
Frama-C command-line options > Frama-C variables > XDG variables.

Each Frama-C plug-in may have its own user directories if required; they are put in a subdirectory `<plug-in shortname>`. Furthermore, they may provide options and variables to override it (where `<plug-in>` is the plug-in's shortname):

- `-<plug-in>-cache` / `-<plug-in>-config` / `-<plug-in>-state`
- `FRAMAC_<PLUG-IN>_CACHE` / `FRAMAC_<PLUG-IN>_CONFIG` / `FRAMAC_<PLUG-IN>_STATE`

Again, command-line options override environment variables. Furthermore, plug-in-specific settings have the priority over kernel settings.

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 15);

4,5,6 Frama-C exits on an internal error. Please report a "bug report" on the Bug Tracking System (see Chapter 15);

125 Frama-C exits abnormally on an unknown error. Please report a "bug report" on the Bug Tracking System (see Chapter 15).

Setting Up Plug-ins 4

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 [15]). Nor does it deal with usage of plug-ins, which is the purpose of individual plug-in documentation (see e.g. [10, 7, 5]).

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: internal plug-ins are automatically installed with the Frama-C kernel, while external plug-ins must be installed separately.

4.2 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, while `frama-c -print-lib-path` must return the directory where the Frama-C compiled library is installed.

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

4.3 Loading Plug-ins

At launch, Frama-C loads all plug-ins in the directories indicated by `frama-c -print-plugin-path`. These directories contain `META` files which are used by Dune to automatically load these plug-ins.

Like other OCaml libraries, Frama-C plug-ins can be located via the environment variable `OCAMLPATH`. It does not need to be set by default, but if you install Frama-C in a non-standard directory (e.g. `<PREFIX>`), you may need to add directory `<PREFIX>/lib` to `OCAMLPATH`.

To prevent Frama-C from automatically loading any plug-ins, you can use option `-no-autoload-plugins`. Then the plugin to load can be selected using `-load-plugin <plugin-name>` (e.g. `aorai`). Since Frama-C plugins are also OCaml libraries it is possible to use `-load-library <library-name>` (e.g. `frama-c-aorai`). Both options accept comma-separated lists of names.

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.

4.3.1 Loading Single OCaml Files as Plug-ins

Frama-C used to have an option `-load-script` that allowed loading a single OCaml file as a mini-plug-in. Since Frama-C 26 (Iron), the use of Dune requires a different approach: you need to create a directory containing the following files:

- a `dune-project` file;
- a `dune` file;
- the `script.ml` file that you want to load with Frama-C.

With these files, you will be able to run `dune build` to compile the script, and then `frama-c -load-module script.cmxs` to load it.

Here is an example `dune-project` file:

```
(lang dune 3.13)
```

Note: you can match the language version (here, 3.7) to the one of your installed `dune` package. Later versions often enable additional warnings.

Here is an example `dune` file:

```
(executable
  (name "script") ; must match the name of the .ml file
  (modes plugin)
  (libraries frama-c.kernel) ; add more if needed
  (flags -open Frama_c_kernel :standard)
  (promote (until-clean)) ; keeps script.cmxs in the base directory
)
```

If your script depends on Frama-C plug-ins, you need to add them to `libraries`, e.g. `frama-c-<plugin>.core`.

Preparing the Sources 5

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, there are a few possibilities:

- If the project is built using `cc`/`gcc` (with a Makefile, for instance), you can use Mopsa-build[1] and `-mopsa-db` (detailed in Section 5.2) to automate the parsing process;
- If you have the list of sources and preprocessing flags (included directories and macros, i.e. `-I`/`-D` for GCC/Clang), you can use option `-cpp-extra-args="-I... -D..."` (detailed in Section 5.3);
- For more complex cases or if you do not know the above commands/options, Section 5.4 details the entire preprocessing and parsing steps performed by Frama-C.

5.2 Parsing Using a Mopsa-DB File

The Mopsa analyzer includes the `mopsa-build` tool, a build wrapper that produces a build database, similar to tools such as `bear` (Build EAR, mentioned in Section 5.3.1).

By running `mopsa-build <build commands>`, it produces a `mopsa.db` file. This binary database can be converted to JSON using `mopsa-db -json > mopsa-db.json`. This file can then be passed on to Frama-C using option `-mopsa-db`. By default, this option will print the list of *targets* identified during the build. A target is either a library or an executable, that you might want to analyze with Frama-C.

Adding option `-mopsa-target` will instruct Frama-C to parse the source files used to build the given target. For instance, if a target `app` has been built using `main.o` and `util.o`, which were in turn built using `main.c` and `util.c`, run:

```
$ frama-c -mopsa-db mopsa-db.json -mopsa-target app
```

It amounts to running `frama-c main.c util.c` (with relevant preprocessing flags, if any).

Note that `-mopsa-db` can accept a directory name, in which case it will search for a file named `mopsa-db.json` inside it. Also, if absent when another Mopsa-DB-related option is set, it behaves as if the user had added `-mopsa-db $PWD`, i.e. set it to the current directory.

Other Mopsa-DB-related options are available, but they are only occasionally useful:

[1] Mopsa is an open-source static analyzer, available at https://gitlab.com/mopsa/mopsa-analyzer. `mopsa-build` is a tool shipped with Mopsa which wraps compiler calls and produces a build database that Frama-C can use.

`-mopsa-exclude-sources <sources...>` tells Frama-C not to parse such sources, even if they are present in the Mopsa-DB file as dependencies of a selected target.

`-mopsa-list-deps <targets>` prints the list of sources that were used to produce the given targets, along with their preprocessing flags, then exits.

`-permissive` allows Frama-C to proceed parsing even when there are missing sources, instead of aborting. Used when the Mopsa-DB contains spurious entries.

Note that, even when using a Mopsa-DB, you can add other source files (e.g. additional stubs) to the command line, to be parsed along with those from the Mopsa-DB.

5.3 Preprocessing Sources in a GCC-Compatible Environment

This section assumes you cannot or do not want to use `-mopsa-db` directly.

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 preprocessed C files. Frama-C preprocesses `.c` and `.h` files with the following basic command:

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

Plus some architecture-dependent flags. The *exact* preprocessing command line can be obtained via options `-kernel-msg-key pp` and `-print-cpp-commands` (the latter exits Frama-C after printing).

Option `-cpp-extra-args` can be used to add arguments to the default preprocessing 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`.

5.3.1 Preprocessing Using a Compilation Database

If you have a JSON compilation database file[2], you can use it to retrieve the above preprocessing flags for each file in the database, via option `-compilation-db <path>`, where `<path>` is the path to the JSON file or to a directory containing a file named `compile_commands.json`.

JSON compilation databases can be produced by CMake (with option `-DCMAKE_EXPORT_COMPILE_COMMANDS=1`) or build wrappers such as Build EAR[3].

Note that, unlike a Mopsa-DB, a `compile_commands.json` file does not include linking information and cannot track dependencies from binaries to sources. This means you need to provide the list of source files to be parsed.

With `compilation-db` 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.

5.4 Preprocessing Sources, Full Details

This section contains details about how preprocessing and parsing are done in Frama-C, which should help deal with non-standard usage scenarios.

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

[3] https://github.com/rizsotto/Bear

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.

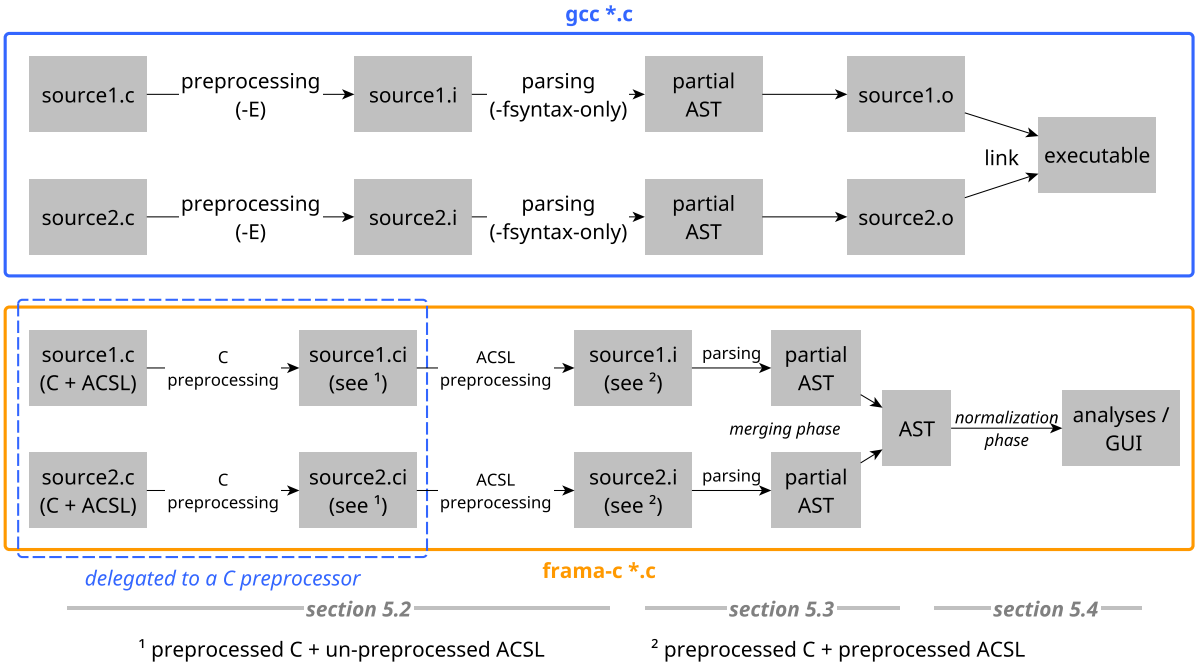


Figure 5.1: Overview of source preparation steps: as performed by GCC (top) and as performed by Frama-C (bottom).

Note that some plug-ins and kernel modules, such as Variadic (described in chapter 12), perform further AST transformations before most analyses are run.

By default, Frama-C considers that Gcc is installed and uses it as preprocessor. If option `-cpp-extra-args` mentioned in the previous section is not enough for your needs (e.g. you use a non-GCC-compatible compiler), you can replace the preprocessing command entirely with option `-cpp-command`. Placeholders (see below) can be used for advanced commands. If no placeholders are used, the preprocessor is invoked in the following way.

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

In this command, `<output file>` is a temporary filename 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 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`.

Preprocessing ACSL Annotations

By default, ACSL annotations are preprocessed (option `-pp-annot` is set by default). If you do *not* want annotations to be preprocessed, 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 in Section 5.11) use such annotations, therefore it might be necessary to disable inclusion of such headers.

Also note that ACSL annotations are preprocessed 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 preprocessing. Instead, `-pp-annot` relies on the ability of Gcc to output all macro definitions (including those given with `-D`) in the preprocessed 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 preprocessed as explained in the previous paragraph. This allows to have macros in ACSL annotations while using a non-GNU-like preprocessor.

5.5 Merging the Source Code files

After preprocessing, 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 apply option `-kernel-warn-key annot-error=warning`[4] to continue parsing after emitting a warning when an ACSL annotation fails to type-check.

5.6 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).

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

[4] See section 7.2 for more information on warning statuses.

`-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`.

`-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 `default`. This includes:

- `default`: chose between `gcc` or `msvc` depending on the machdep (see `-machdep`).
- `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`: same behavior than `gcc-enums` but starting from `char` instead (Gcc's `-fshortenums` option)
- `msvc`: same behavior than `int`

`-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.

`-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-functions <value>` defines which unused function prototypes are kept in the AST. By default (value `specified`), only those with ACSL specifications are preserved; the user can force those to be removed as well (with value `none`), or keep prototypes that are both unused and unspecified (value `all`). A special value `all_debug` also preserves predefined prototypes (e.g. compiler builtins). This option extends and replaces previous options `-keep-unused-specified-functions` and `-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_64` 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
```

Apart from these default platforms, it is possible to give as argument to `-machdep` option the path of a YAML file containing the information needed by Frama-C. The Plug-in Development Guide [15] describes this format in more detail, as well as the use of `frama-c-script make-machdep` to automatically generate it. The YAML schema contains a list of builtin macros with their values. However, any such `MACRO` can be undefined by passing `-UMACRO` in the preprocessor arguments (using `-cpp-extra-args` or its per-file counterpart as in Sect. 5.3, or a compilation database as in Sect. 5.3.1). In addition, since compilers tend to define builtin macros with a varying number of underscores as prefix and suffix, `-UMACRO` will also undefine `_MACRO`, `__MACRO`, `__MACRO__`, etc.

`-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.

`-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 unfolded individually, by inserting a `loop unfold` annotation just before the loop statement. Do not confuse this option with plug-in-specific options that may also be called "unrolling" [10][5]. Below is a typical example of use:

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

The transformation introduces an additional `loop unfold` annotation indicating that the unrolling process has been done:

```
... // loop unrolled 10 times
/*@ loop unfold 10;
    loop unfold "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 `loop unfold "done"` and force unrolling, the option `-ulevel-force` has to be set.

Passing a negative argument to `-ulevel` will disable unrolling, even in case of `loop unfold` annotations.

[5] Historically, *syntactic* loop unrolling was denoted by `loop pragma UNROLL` annotations. Later on, EVA introduced *semantic* unrolling with `loop unroll` annotations. Loop pragams was removed for Frama-C 29 (Copper) and it was decided to introduce `loop unfold` annotations for *syntactic* unrolling and to reserve `loop unroll` annotations for *semantic* loop unrolling with EVA.

Mode	ACSL		Frama_C		Safe	
	Proto	Body	Proto	Body	Proto	Body
exits	—	—	\false	\false	—	\false
assigns	\everything	\everything	Auto[a]	Auto	\everything	\nothing
allocates	\nothing	\nothing	ACSL[b]	ACSL	\everything	\nothing
terminates	\true	\true	ACSL	ACSL	\false	\true
requires	—	—	ACSL	ACSL	\false	—

[a] Automatically generated using the function parameters.

[b] Refer to ACSL mode to see what is generated.

Table 5.1: Specification generation's behaviors according to available modes

5.7 Normalizing Contracts

Normalization gives a program which is semantically equivalent to the original one, but there is an exception for function contracts. By default, function contracts are left as is, but Frama-C and plug-ins can require the presence of some specification to improve their performances. When they do, they can ask Frama-C's kernel to generate specifications.

For example, let us say we have a function `f` that is only declared and has no ACSL `assigns` clause, but these `assigns` are required by Eva. Eva asks the kernel to generate `assigns` before resuming the analysis. Indeed, as mentioned in the ACSL manual [3], 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 generation is done either by combining existing specifications or according to a *mode* which can be selected via Frama-C's options. A plug-in developer has also the possibility to create their own mode using Frama-C's API. The Plug-in Development Guide [15] describes all steps required to create and use a custom mode. All available modes are detailed in table 5.1. Additionally, a status is emitted for each generated clause. A summary can be found in table 5.2.

By default, Frama-C selects the mode `Frama_C` and the following options allow selecting mode(s) for the specification generation:

`-generated-spec-mode <mode>` selects which `mode` is used to generate missing specifications. `mode` can be either "frama-c", "acsl", "safe" or any custom mode registered in a plug-in.

`-generated-spec-custom <clause_1:mode_1,clause_2:mode_2,...>` has a similar behavior as `-generated-spec-mode`, except that it allows the user choosing a `mode` individually for each clause. It also allows the user ignoring Frama-C and/or plug-ins needs by skipping the generation for a clause, with the mode "skip". This mode is not available with `-generated-spec-mode` and trying to use it emits a warning and replaces it by `Frama_C`. For example:

```
$ frama-c -generated-spec-custom exits:skip,requires:safe
```

Here every clause is generated using the default mode `Frama_C` except for exits which are not generated and `requires` which are generated using the mode `Safe`.

As mentioned, the generation can also be done by combining existing clauses from other behaviors following this algorithm:

1. If the function contract has some clauses `C` in unguarded (without `assumes` clauses) behaviors, we combine them in the default behavior.
2. Else, if it has some clauses `C` in complete behaviors, we combine them in the default behavior.

Mode	ACSL	Frama_C	Safe
exits	Dont_know	Dont_know	Dont_know
assigns	—	Dont_know	Dont_know
allocates	True	Dont_know	Dont_know
terminates	True	Dont_know	Dont_know
requires	—	—	—

Table 5.2: Status emitted during the specification generation according to the selected mode

3. Else, if it has some clauses `C` anywhere, we combine them in the default behavior.
4. Else we use the selected mode to generate these clauses (see mode's table).

Note: There is an exception with the mode `ACSL` for which we never try to combine existing clauses from other behavior, and jump directly to the step of our algorithm to use the selected mode.

When a clause is generated for a function, Frama-C's kernel can emit a warning if this function is a prototype and if it is not part of Frama-C's builtins. The message emitted depends on the generated clauses, their origins (if we combined some clauses from existing ones, or generated them from nothing, or both) as well as if the function contract was empty before the generation or incomplete.

5.8 Incremental parsing

(experimental)

Parsing large code bases takes a non-negligible amount of time. More importantly, non-modular analyses need to be recomputed in their entirety even for minuscule AST changes.

In order to help deal with these issues, and to help support for future incremental analyses (which recompute results only for the modified AST parts and their dependencies), you can use option `-ast-diff`. It enables computing the difference between a previous AST (loaded via option `-load` (Section 9.1.3) and the AST computed from the current sources. This difference is stored by the Frama-C kernel and can be used by plug-ins which support it. Note however that since `-load` implies that files given on the command line are ignored, `-load` and `-ast-diff` with the list of C files to consider must be properly sequenced through the use of `-then` or one of its variants (Section 3.3.5), as in:

```
$ frama-c -load previous.sav -then -ast-diff file1.c file2.c [...]
```

Note: currently, parsing and type-checking are systematically performed even with this option, and `-ast-diff` compares the normalized ASTs. In the future, though, these steps should be refactored to allow saving further time.

5.9 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 preprocessing 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.

`__FC_FORCE_POSIX_C_SOURCE`: By default, Frama-C's positions `_POSIX_C_SOURCE` according to the relevant machdep field `posix_c_source`. Defining `__FC_FORCE_POSIX_C_SOURCE` overrides this value. If the macro is defined to an empty value, `_POSIX_C_SOURCE` will be undefined.

5.10 Compiler and language extensions

Frama-C's default behavior is to be fairly strict concerning language features. By default, most C99 and C11 features are accepted[6], but most non-ISO C compiler extensions are not accepted, similarly to when compiling a program with `gcc -std=c11 -pedantic -pedantic-errors`.

However, depending on the machine architecture (see option `-machdep`, in Section 5.6), 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; see option
 -machdep or run 'frama-c -machdep help' for the list of available machdeps
```

This means that using a GCC or MSVC machdep (e.g., `-machdep gcc_x86_64`) will allow the language extension to be accepted by Frama-C.

Frama-C's standard library (detailed in the next section) also includes a GCC-specific header to improve analyses: `__fc_gcc_builtin_macros.h` defines several GCC libc-related builtins as their libc equivalents (e.g. `__builtin_pow` as `pow`) so that Frama-C can reuse its standard library specifications. You need to include it manually, however, using an option such as:

```
$ frama-c <options and sources> \
  -cpp-extra-args="-include __fc_gcc_builtin_macros.h"
```

[6] For more details about C99/C11 feature support, see Section 14.1.

5.11 Standard library (libc)

Frama-C bundles a C standard library in order to help parse and analyze code that relies on the functions specified in the ISO C standard. Furthermore, In order to simplify parsing of code using common open-source libraries, Frama-C's standard library also includes some POSIX and non-POSIX headers which are not part of ISO C. Such libraries are provided on a best-effort basis, and they are part of the *trusted computing base*: to ensure its correctness, the specifications must be ultimately proofread by the user.

This library, while incomplete, is constantly being improved and extended. It is 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 a few library functions.

By default, Frama-C's preprocessing will include the headers of its own standard library, instead of those installed in the user's machine. This avoids issues with non-portable, compiler-specific features. Option `-frama-c-stdlib` (set by default) adds `-I$(frama-c -print-share-path)/libc` to the preprocessor command, as well as some GCC-specific options, including (but not limited to) `-dD`, `-nostdinc`, `-undef`, and some `-Wno-*` options. If any of these if not accepted by your preprocessor, `-no-cpp-frama-c-compliant` can be given to Frama-C (see section 5.4).

The use of Frama-C's standard library contains a few caveats:

- definitions which are not present in Frama-C's standard library may cause parsing errors, e.g. missing type definitions, or missing fields in structs;
- if a header is included which is not available in Frama-C's library, preprocessing will fail by default (due to the `-nostdinc` option); if the user manually includes the system's library, e.g. by adding `-cpp-extra-args='-I/usr/include'`, the preprocessor will end up mixing headers from Frama-C's library with those from the system, often leading to incompatible definitions and unexpected parsing errors. In this case, the best approach is usually to include *only* the missing definitions, for instance by copying them to a separate header file, included manually or via the GCC-compatible option `-include <header.h>` (see *GCC's Preprocessor Options* for more details). Alternatively, consider filing an issue (see Chapter 15) to ask for the inclusion of such headers and/or definitions in Frama-C's standard library.

Note that, while Frama-C's library intends to offer maximum portability, some definitions such as numeric constants require actual values to be defined. For pragmatic reasons, such definitions are most often based on the values defined in GNU/Linux's libc, and may differ from those in your system. As stated before, if you want to ensure the code analyzed by Frama-C is strictly equivalent to the one from the target system, you must either proofread the definitions, or provide your own library files.

5.12 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 preprocessing 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
```

ACSL Extensions 6

6.1 Extension Syntaxes

When a plug-in registers an extension, it can be used in ACSL annotations with 2 different syntaxes. As an example, with an extension `bar` registered by the plug-in `foo`, we can use the short syntax `bar _` or the complete syntax `\foo::bar _`.

The complete syntax is useful to print better warning/error messages, and to better understand which plug-in introduced the extension. Additionally, all extensions coming from an unloaded plug-in can be ignored this way. For example, if Eva is not loaded, `\eva::unroll _` annotations will be ignored with a warning, whereas `unroll _` cannot be identified as being supported by Eva, which means that it can only be treated as a user error.

6.2 Handling Indirect Calls with `calls`

In order to help plug-ins support indirect calls (i.e. calls through a function pointer), an ACSL extension is provided. It is introduced by keyword `calls` and can be placed before a statement with an indirect call to give the list of functions that may be the target of the call. As an example,

```
/*@ calls f1, f2, ... , fn */
*f(args);
```

indicates that the pointer `f` can point to any one of `f1`, `f2`, ..., `fn`.

It is in particular used by the WP plug-in (see [7] for more information).

6.3 Importing External Module Definitions

Support for ACSL modules has been introduced in Frama-C+dev. Module definitions can be nested. Inside a module `A`, a sub-module `B` will actually defines the module `A::B`, and so on.

Notice than for long identifiers like `A::B::C` to be valid, no space is allowed around the `:` characters, and `A`, `B`, `C` must be regular ACSL identifiers, i.e. they shall only consist of upper case or lower case letters, digits, underscores, and must start with a letter.

Inside module `M` declaration, where `M` it the long identifier of the module being declared, a logic declaration `a` will actually define the symbol `M::a`. You shall always use the complete name of an identifier to avoid ambiguities in your specifications. However, in order to ease reading, it is also possible to use shortened names instead.

The rules for shortening long identifiers generalize to any depth of nested modules. We only illustrate them in a simple case. Consider for instance a logic declaration `a` in module `A::B`, depending on the context, it is possible to shorten its name as follows:

- Everywhere, you can use `A::B::a`;
- Inside module `A`, you can use `B::a`;
- Inside module `A::B`, you can use `a`;
- After annotation **import** `A::B`, you can use `B::a`;
- After annotation **import** `A::B as C`, you can use `C::a`;

You may also use local **import** annotations inside module definitions, in which case the introduced aliases will be only valid until the end of the module scope.

Depending on dedicated plug-in support, you may also import modules definitions from external specifications, generally from an external proof assistant like Coq or Why3. The ACSL extended syntax for importing external specifications is as follows:

```
import <Loader>: <ModuleName> [ \as <Name> ];
```

This is a generalization of the regular ACSL **import** clause just mentioned above. The `<Loader>` name identifies the kind of external specifications to be loaded. Loaders are defined by dedicated plug-in support only, and you shall consult the documentation of each plug-in to known which loaders are available. A loader syntax can be either just a name, used when the extension was registered, or `<\plugin::name>`. The second syntax is useful to avoid ambiguities if several plug-ins register a module importer extension with the same name.

The `<ModuleName>` identifies both the name of the imported module and the external specification to be imported, with a `<Loader>` dependent meaning.

The alias name `<Name>`, if provided, has the same meaning than when importing regular module names (just described above) in the current scope.

When importing *external* specifications, depending on the `<Loader>` used, it is possible to have logic identifiers with an extended lexical format:

- `M::(op)` where `M` is a regular module identifier, and `op` any combination of letters, digits, operators, brackets, braces, underscores and quotes. For instance, `map::Map::([<-])` is a syntactically valid identifier, and `number::Complex::(<=)(a,b)` is a syntactically valid expression.
- `M::X` where `M` is a regular module identifier and `X` any combination of letters, digits, underscores and quotes. For instance, `Foo::bar'jazz` is a syntactically valid identifier.

External module importers are defined by plug-ins *via* the extension API, consult the plug-in manuals and the plug-in developer manual for more details.

Platform-wide Analysis Options 7

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. [10, 5, 7]) for specific options.

7.1 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.

7.2 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

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

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.

feedback-once a feedback message is emitted only once; the status of the category is then reset to `inactive`.

warning a proper warning is emitted.

warning-once a proper warning is emitted only once; the status of the category is then reset to `inactive`.

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

error-once an error is emitted only once; the status of the category is then reset to `inactive`.

abort an error message is emitted and Frama-C will immediately abort its execution.

The `=<status>` part might be omitted, which is equivalent to asking for `warning` status. The new status will be propagated to subcategories, with one exception: statuses "abort", "error" and "error-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 $path_1 : name_1, \ldots, path_n : name_n$ in argument and replaces each $path_i$ with $name_i$ when displaying file locations in messages.

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

7.3 Customizing Analyzers

The descriptions of the analysis options follow. For the first two, the description comes from the Eva manual [10]. Note that these options are very likely to be modified in future versions of Frama-C.

`-absolute-valid-range m-M` specifies that absolute addresses between `m` and `M`, inclusive, are valid (for reading or writing). All other absolute addresses are considered invalid. This option allows specifying several disjoint intervals of valid absolute addresses. m and M may 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;
```

Currently, the option is disabled by default. The rationale for this is the fact that it creates a lot of redundancies with pointer access alarms and that most of the time even when a pointer reaches such an invalid value, it is never read again, or if it is, the memory access will trigger an alarm.

`-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-unaligned-pointer` may be used to check that the code does not create pointers whose alignment is not suitable for the type of the object that it points to. This option is set by default. In the following example, analyzers report by default an error on the second line. Disabling the option removes this verification.

```
char c[5];
int* p = (int*) &c[1];
```

`-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`.

7.4 Performance Parameters

The following options allow the user to adjust the amount of memory used by Frama-C:

`-memory-footprint <1-10>` governs the memory usage of Frama-C. With smaller values, analyses consume much less memory, but are also slightly slower. The argument must be between 1 and 10; default is 6.

`-cache-size <1-10>` controls the amount of memory allocated to some internal caches, mainly used by the Eva analysis. Each increase of 1 doubles the size of the caches, the default value being 2. Small values are most suitable for the analysis of small programs (less than 1000 lines), which would not benefit from larger caches. Higher values (around 7-8) can speed up the analysis of large code bases, but increase their memory usage.

Property Statuses 8

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 [8].

8.1 A Short Detour through Annotations

Frama-C supports writing code annotations with the ACSL language [3]. 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 [11]).

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*" [3, 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.

8.2 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 7). 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 [7] 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.

8.3 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 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:

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

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.

[1] Some plug-ins might consider *all possible traces*, which constitute a safe over-approximation of the intended 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 10 for details), or in batch mode by the `report` plug-in.

General Kernel Services 9

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

9.1 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'.

9.1.1 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.

9.1.2 Using Projects

The list of existing projects of a given session is visible in the graphical interface through the `Project` menu and through the `Projects` sidebar. Among other actions on projects (duplicating, renaming, removing, saving, *etc*), they allow the user to switch between different projects during the same session.

In the command line, the only way to handle a multi-project session is through 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 consumption. In particular, the category `@all_but_current` removes all existing projects other than the current one.

9.1.3 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 9.1 *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 `-explain`, `-quiet` and `-unicode` are not saved on disk.

9.2 Dependencies between Analyses

Usually analyses do have parameters (see Chapter 7). 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 5.6). To compute its result, the value analysis assumes the memory range 0:0x1000 is addressable (option `-absolute-valid-range`, see Section 7.3). 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.

Graphical User Interface 10

Running `frama-c-gui` displays the Frama-C Graphical User Interface (GUI): Ivette (*Integrated Verification Environment Toolkit for Trusted Execution*).

Ivette is based on the Electron[1] framework and is thus not always directly distributed with Frama-C, especially if you installed it using your distribution's package manager.

For an Opam-based installation of Frama-C, running `frama-c-gui` for the first time will trigger a script that downloads, compiles, installs and runs Ivette. This script requires some tools such as Node.js and Yarn. Subsequent executions of `frama-c-gui` will directly run the installed executable.

Note that Ivette is also distributed as a standalone executable in the Frama-C website, as AppImage on Linux and Apple Disk Image (.dmg) on macOS.

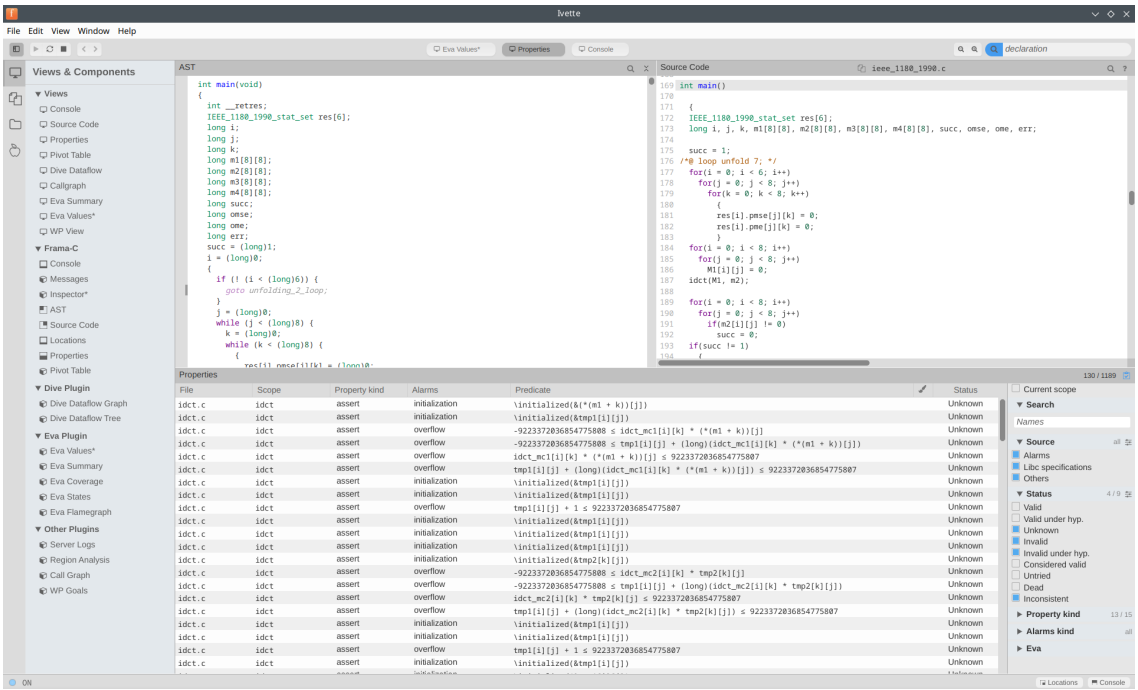


Figure 10.1: Screenshot of Ivette, illustrating several components

Ivette's documentation is accessible from its interface, via the *Help* - *Documentation* menu, and from each component's question mark (?) button. Clicking on them opens a dialog with the relevant documentation.

[1] https://www.electronjs.org/

10.1 Command-line options

`frama-c-gui` accepts almost all command-line options that `frama-c` does, so you can usually replace `frama-c <options>` with `frama-c-gui <options>` to run an analysis and see its results.

There are currently a few options that are intercepted by the Electron process itself instead of passed on to the underlying `frama-c` process:

- `--no-sandbox`: this option avoids issues with Electron sandboxing (with e.g. Ubuntu 24.04). *Note that this may have security implications.*
- `--settings <file.json|CLEAN>`: allows using a specific settings file. The special value `CLEAN` launches Ivette with default settings.
- `--working <dir>`: uses `<dir>` as working directory.

Reports 11

An execution of Frama-C outputs many messages, warnings and various property statuses in textual form. The Graphical User Interface (see Chapter 10) 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.

Frama-C has recently earned the ability to output data in the SARIF[1] format. This is performed by the Markdown Report plug-in, described in section 11.4.

11.1 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.

[1] Static Analysis Results Interchange Format, https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=sarif

11.2 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
```

11.3 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 11.3.6 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.

11.3.1 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.

11.3.2 Rules

Each classification rule is a JSON record following the structure of Figure 11.1. 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 11.1: 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[2] 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.

11.3.3 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.

[2] Warning categories are described in section 7.2.

11.3.4 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>`.

11.3.5 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 11.2.

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

Figure 11.2: Classified Event JSON Format

11.3.6 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'`)

11.4 SARIF Output via the Markdown Report Plug-in

SARIF is a JSON-based standard output format for static analysis results. It is currently supported by Frama-C in its version 2.1.0. Some IDEs, such as Visual Code, contain plug-ins which allow importing SARIF reports. A few Microsoft tools and libraries related to SARIF are available at https://sarifweb.azurewebsites.net/. Microsoft also publishes command-line tools for SARIF, made available via NPM packages and .Net Core applications.

11.4.1 Prerequisites

SARIF output is produced by the Markdown Report (MdR) plug-in. The plug-in is distributed with Frama-C, but it depends on optional features, namely the `opam` package `ppx_deriving_yojson`, so it may not be available in every Frama-C installation.

When installing Frama-C via opam, include the optional dependency `ppx_deriving_yojson` to ensure MdR will be available. Note that MdR has other features and output formats, which are not described here.

11.4.2 Generating a SARIF Report

To enable generation of a SARIF report, use option `-mdr-gen sarif`. It will produce a JSON file (by default, `report.sarif`) with an entry for each ACSL property.

You can change the output file name with option `-mdr-out <f>`.

Note that there are no filtering options in the SARIF output; it is up to the tools importing the file to decide which information to sort, filter, and display.

For more details about Markdown Report, use option `-mdr-help`.

Variadic Translation 12

This chapter briefly presents the transformations performed by kernel option `-variadic-translation` (enabled by default), which translates 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 Translation component helps dealing with some of these challenges, reducing or eliminating the need for plug-ins to have to deal with these special cases.

12.1 Translating variadic function calls

ACSL does not allow the direct specification of variadic functions: variadic arguments have no name and no statically known type. Option `-variadic-translation` 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-translation` 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.

12.2 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`). `-variadic-translation` 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.

12.3 Usage

12.3.1 Concrete Example

Here is a concrete example of a program using a variadic function from the C standard library:

```
#include <stdio.h>
void main() {
  int n = 5;
  printf("%d, %*s", 42, n, "hello");
}
```

And here is the output produced by `-variadic-translation`:

```
#include "stddef.h"
#include "stdio.h"
```

```
/*@ requires valid_read_string(param2);
    requires valid_read_string(format);
    assigns \result, __fc_stdout->__fc_FILE_data;
    assigns \result
      \from (indirect: __fc_stdout->__fc_FILE_id),
            (indirect: __fc_stdout->__fc_FILE_data),
            (indirect: *(format + (0 ..))), (indirect: *(param2 + (0 ..))),
            (indirect: param1), (indirect: param0);
    assigns __fc_stdout->__fc_FILE_data
      \from (indirect: __fc_stdout->__fc_FILE_id),
           __fc_stdout->__fc_FILE_data, (indirect: *(format + (0 ..))),
           *(param2 + (0 ..)), param1, param0;
 */
int printf_va_1(char const * restrict format, int param0, int param1,
                char *param2);

void main(void)
{
  int n = 5;
  printf("%d, %*s",42,n,(char *)"hello"); /* printf_va_1 */
  return;
}
```

12.3.2 Main options

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

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

`-no-variadic-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.

12.3.3 Similar diagnostics by other tools

Some of the issues detected by `-variadic-translation`, 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.

12.3.4 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-translation`, 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 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.

12.3.5 Pretty-printing translated code

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

Analysis Scripts 13

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.

13.1 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.

13.2 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 `lib` directory (`frama-c -print-lib-path`), in subdirectory `analysis-scripts`.

13.2.1 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 configures the initial analysis (see Section 13.2.2 below);
3. The user edits and runs the generated Makefile, adjusting the analysis as needed and re-running `fcmake`[1].

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

Section 13.3 details usage of the Makefile and presents an illustrative diagram.

13.2.2 Necessary Build Information

The necessary build information for a new analysis consists in the following data:

machdep : architectural information about the system where the code will run: integer type sizes, compiler, OS, etc. See section 5.6 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 does not require a `main` function, but plug-ins such as `Eva` do.*)

The standard way of setting up a new analysis is via these steps:

1. create a `.frama-c` directory in the root of the code repository (all Frama-C-related files will be stored inside it).
2. copy the *analysis template* in the `.frama-c` directory. This template is a Makefile located in `FRAMAC_LIB/analysis-scripts/template.mk`[2]. It is usually renamed as `GNUmakefile`.
3. open this Makefile and fill in the required information:
   - the architecture in the `MACHDEP` variable.
   - the preprocessing flags in the `CPPFLAGS` variable.
   - the list of sources in the `main.parse` target.
   - the function to analyze, if different from `main`, by adding `-main <function>` to the `FCFLAGS` variable.

A project without this information is incomplete; an alternative workflow is then necessary. The next section presents some possibilities to retrieve such information. Note that with tools such as `mopsa-build` (mentioned in Section 5.2), all but the first item can be obtained automatically from the name of an executable file produced by a build tool such as Make.

13.2.3 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.

[1] `fcmake` is described in Section 13.3.

[2] `FRAMAC_LIB` is the result of the command `frama-c -print-lib-path`

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 13.4), 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).

13.2.4 Using a JSON Compilation Database (JCDB)

Note: If you can use `mopsa-build` to obtain a `mopsa-db.json` file, you can skip this section and use Frama-C's option `-mopsa-db`. See Section 5.2 for more details.

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 5.3). This leads to a different workflow:

1. For CMake:
   - Run `cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=1 <targets>`.

   For Makefile:
   - Install Build EAR (https://github.com/rizsotto/Bear);
   - Run `bear -- make <targets>` (instead of `make <targets>`).

   In both cases, you will obtain 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. You can then follow the numbered steps in section 13.2.2.

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 inspection of the result of `frama-c-script list-files` is often necessary.

13.3 Using the generated Makefile, via `fcmake`

The generated Makefile can be used to run one or several analyses. Its basic usage involving the `fcmake` alias (equivalent to `make -C .frama-c`) is the following:

- `fcmake <target>.parse`: parse the sources
- `fcmake <target>.eva`: run Eva
- `fcmake <target>.eva.gui`: open the results in the GUI

This section details how to produce the Makefile, how to define the `fcmake` alias, and lists other useful targets and settings.

Storing Frama-C files and results in `.frama-c`

By default, the generated `GNUmakefile` is created in the (hidden) directory `.frama-c`, which should contain all files specific to Frama-C. This arrangement provides several benefits:

- Frama-C-related files do not pollute the original code; everything is stored in a separate directory, easily identifiable by its name;
- Existing makefiles are not overridden by Frama-C's;
- Having a standardized structure helps with CI integration.

However, special attention is needed due to a few consequences of this structure:

- The `make` process will be run from a subdirectory of the current one; therefore, source and include paths must be either absolute or prefixed with `..`;
- In some cases, it may be necessary to add flags such as `-I ..`, so that the preprocessor will find the required files.

Defining and using `fcmake`

We recommend defining the following *alias* in your shell:

```
alias fcmake='make -C .frama-c'
```

Running `fcmake` will have the same effect as running `make` inside the `.frama-c` directory.

The commands in this section assume usage of the `fcmake` alias defined above.

Frama-C makefile targets and variables

The diagram in Fig. 13.1 summarizes the usage of the generated Makefile. Its 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 that, since the generated makefile is inside `.frama-c`, relative paths to source files will always begin with `../`, except for sources located within `.frama-c`, e.g. `fc_stubs.c`.

Target names can contain hyphens and underscores, but neither slashes nor 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.

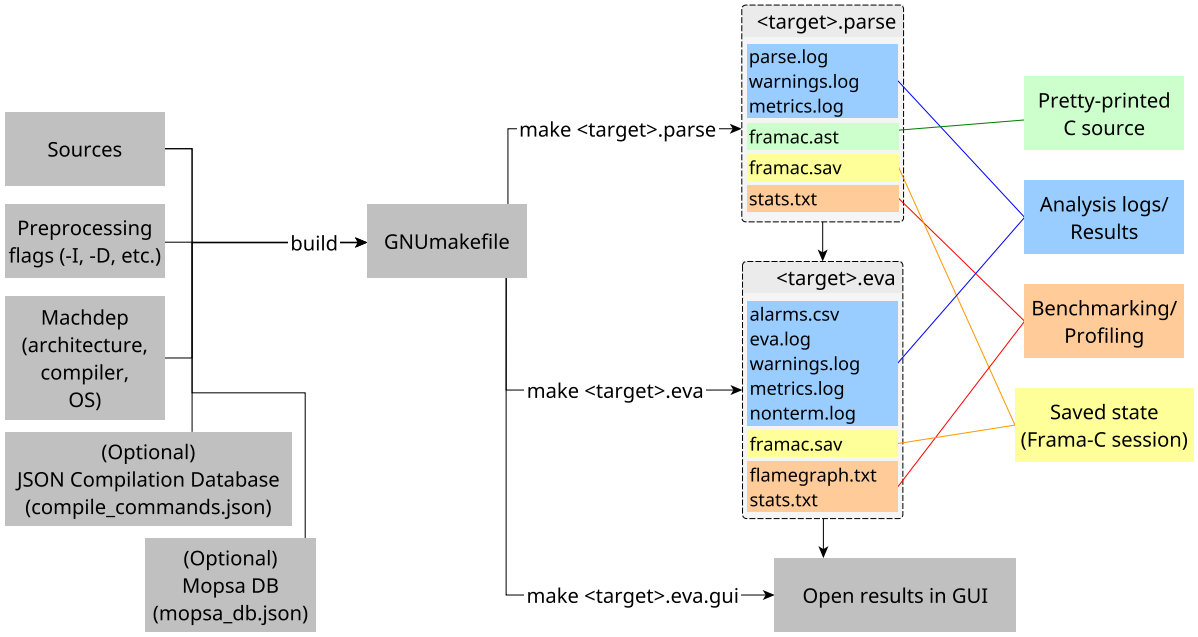


Figure 13.1: Overview of the *analysis-scripts* workflow: the inputs on the left produce a GNUmakefile which is then used for parsing, analyzing and visualizing results.

13.3.1 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
```

`-I` flags referencing relative paths must take into account the fact that Frama-C will be run from the `.frama-c` directory, and therefore must include an initial "`../`".

13.3.2 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. 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.

13.3.3 Adding new analyses and stages

Besides the predefined Eva-oriented steps, you can easily add other stages and analyses, which may or may not depend on Eva.

For instance, to add a SARIF report using the Markdown Report plug-in, you can simply add, before the template epilogue, the following lines, where `target` is the name of your target:

```
target.sarif: target.parse
	$(FRAMAC) -load $^/framac.sav -mdr-gen sarif -mdr-out $@
```

This rule will create a file `target.sarif` inside the `.frama-c` directory. The rule will depend on the parsing of `target.parse` and use the saved session at `target.parse/framac.sav`.

If you want the report to run after the analysis with Eva, instead, simply replace `.parse` with `.eva`.

Then, running `fcmake target.sarif` will create or update the report, recomputing dependencies when needed.

Adding a new stage, with a saved session that can be reused later for other stages and analyses, requires just a few more lines, as in the following example:

```
target.wp: target.parse
	mkdir -p $@
	$(FRAMAC) -load $^/framac.sav -wp -save $@/framac.sav
```

In the example above, we define a new stage, `target.wp`, which depends on the parsing stage, runs the WP plug-in, and saves the result in a session file. This session file can then be loaded by another stage, or in the GUI. For instance, the `.gui` predefined target works out of the box in this case: running `fcmake target.wp.gui` will load the saved session in the GUI.

13.4 Script Descriptions

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

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 described in section 13.2.3.

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 : 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. Provided on a best-effort basis, but does work for all scripts.

flamegraph : opens a *flamegraph*[3] 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.

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 command, based on an external tool (C-Reduce[4], or its Python rewrite, `cvise`):

creduce : A script to help running the C-Reduce 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. Basic usage is as follows:

```
frama-c-script creduce file.i [frama-c options]
```

The script requires a single file name (better if preprocessed, but also works with `.c` files) followed by Frama-C options that cause the crash. It creates a `script_for_creduce.sh` file in the current directory, as well as a `creducing` subdirectory containing the reduced code under progress. The script contains extensive comments about its usage, and it is also described in a Frama-C blog post[5].

To use the `creduce` command, you need to have the `creduce` or `cvise` tools installed in your path, or in environment variable `CREDUCE`.

---

[3] See https://github.com/brendangregg/FlameGraph for details about flamegraphs.

[4] See https://embed.cs.utah.edu/creduce for more details.

[5] *Debugging Frama-C analyses: better privacy with C-Reduce*, at https://www.frama-c.com/scripts/usability/2020/04/02/creduce.html.

13.5 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 `.frama-c/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.

13.6 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 `fcmake` with `-B` (to force recomputation of dependencies), or running `fcmake clean` before re-running `fcmake`.

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.

Compliance 14

Frama-C provides sound analyses for several kinds of code defects. Given the large amount of covered defects and command-line options which toggle their reporting, we provide in this chapter information about standards compliance, coding guidelines and related documents, such as the ISO C standard, CWEs, CERT C, etc.

This chapter is not exhaustive; in particular, some defects are implicitly checked without any controlling options; others may be affected by a combination of several options which is hard to precisely express.

Please contact the Frama-C team if you require a thorough evaluation of the standards cited here, or of different ones. This reference is provided on a best-effort basis.

14.1 Unsupported C99 and C11 Features

The table below lists features of the C99 standard (ISO/IEC 9899:1999) which are currently unsupported by Frama-C. Please contact the Frama-C team if you would like them be to supported.

Table 14.1: Unsupported C99 features

Feature	***Details***
Complex/Imaginary types	`_Complex`/`_Imaginary` types are not natively supported in OCaml. Lack of user demand led to the absence of their implementation in Frama-C.
Nested VLAs	Nested variable-length arrays (VLAs), that is, `int t[a][a]` where `a` is not a compile-time constant, are currently unsupported. VLAs with a single variable dimension are supported.
Floating-point environment	Some prototypes and stubs are available in the `fenv.h` and `fenv.c` files provided in Frama-C's standard library (Section 5.11), but they are incomplete.

The table below lists features of the C11 standard (ISO/IEC 9899:2011) which are currently unsupported by Frama-C. Please contact the Frama-C team if you would like them be to supported.

Table 14.2: Unsupported C11 features

Feature	***Details***
`_Atomic`	Currently unsupported (ignored) by the Frama-C kernel.

Table 14.2: Unsupported C11 features

Feature	***Details***
Unicode	`u/U` character constants, `u8/u/U` string literals, and `\u` escape sequences are currently unsupported by the lexer. **Note**: in a few cases, such as ACSL specifications, some `\u` escape sequences are allowed (e.g. for ACSL symbols such as $\in$, $\mathbb{Z}$, etc).
Complex/Imaginary types	As in C99 (see Table 14.1). Note that C11 made complex types optional (they were required in C99).
Nested VLAs	As in C99 (see Table 14.1). Note that C11 made variable-length types optional (they were required in C99).

14.2 Frama-C Options Related to C Undefined Behaviors

This section lists several Frama-C options affecting (either enabling or disabling) the detection of *unspecified* and *undefined* behaviors listed in Annexes J.1 and J.2 of the C11 standard (ISO/IEC 9899:2011).

Note: the ISO C standard does not provide an identifier for each behavior; therefore, we use the numbers listed in *SEI CERT C Coding Standard's Back Matter*, sections *CC. Undefined Behavior* and *DD. Unspecified Behavior*, whenever possible. These tables can be found at `https://wiki.sei.cmu.edu/confluence/display/c`. Note that SEI CERT does not list implementation-defined behaviors; in such cases, we simply refer to the related section in the ISO C standard.

Table 14.3: Impact of some Frama-C options on a few unspecified and undefined behaviors.

Command-line Option	***Affected behaviors***
`-eva-initialization-padding-globals`	Controls UnsB related to *DD.10* for the initialization status of padding in global variables.
`-eva-warn-pointer-subtraction`	Toggles warnings related to UB *CC.48*.
`-eva-warn-undefined-pointer-comparison`	Toggles warnings related to pointer comparisons (related to UB *CC.53*).
`-initialized-padding-locals`	Toggles UnsB related to *DD.10* for local variables.
`-unspecified-access`	Enables reporting of some instances of UB *CC.35*.
`-warn-invalid-bool`	Toggles reporting of UB *CC.12* when applied to values of type `_Bool`.
`-warn-invalid-pointer`	Toggles reporting of UBs *CC.46* and *CC.62*.
`-warn-unaligned-pointer`	Toggles reporting of UBs *CC.24*.
`-warn-left-shift-negative`	Toggles reporting of UB *CC.52*.
`-warn-pointer-downcast`	Toggles reporting of UB *CC.23*.
`-warn-right-shift-negative`	Toggles reporting of the IDB mentioned in C11 §6.5.7.5.
`-warn-signed-downcast`	Toggles reporting of UB *CC.36* for signed types, when converting from a wider to a narrower type.
`-warn-signed-overflow`	Toggles reporting of UB *CC.36* for operations on signed types (except when converting from a wider to a narrower type).
`-warn-unsigned-downcast`	Toggles reporting of UB *CC.36* for unsigned types, when converting from a wider to a narrower type.
`-warn-unsigned-overflow`	Toggles reporting of a situation similar to UB *CC.36*, for operations on *unsigned* types, even though they are allowed by C11.

14.3 RTE categories and C Undefined Behaviors

This section presents the correspondence between runtime execution (RTE) alarms emitted by the Eva plugin and C undefined behaviors.

Table 14.4: Correspondence between Frama-C's runtime error categories and C's undefined behaviors.

RTE Category	***Related UBs***	***Notes***
bool_value	*CC.12*	All values other than $\{0, 1\}$ are trap representations for the `_Bool` type.
dangling_pointer	*CC.9*, *CC.10*, *CC.177*	
differing_blocks	*CC.48*	
division_by_zero	*CC.45*	
float_to_int	*CC.17*	
function_pointer	*CC.26*, *CC.41*	
freeable	*CC.179*	This is not an RTE category, but an alarm related to an ACSL precondition.
index_bound	*CC.49*	
initialization	*CC.11*, *CC.21*, *CC.180*	
mem_access	*CC.33*, *CC.43*, *CC.47*, *CC.62*, *CC.64*, *CC.176* (among others)	
overflow	*CC.23*, *CC.36*, *CC.50*	This category comprises the following RTEs: `signed_overflow`, `unsigned_overflow`, `signed_downcast` and `unsigned_downcast` (mostly related to *CC.36*), and `pointer_downcast` (related to *CC.23*). For *CC.50*, see *Note about `ptrdiff_t`*.
overlap	*CC.54*	*CC.100* is handled by Frama-C's libc ACSL specifications.
pointer_alignment	*CC.24*	
pointer_value	*CC.46*, *CC.62*	
ptr_comparison	*CC.53*	
separation		Related to ACSL memory separation hypotheses.
shift	*CC.51*, *CC.52*	
special_float		Non-finite floating-point values are not UB, but can be optionally considered as undesirable.

Note that some undefined behaviors, such as *CC.100*, *CC.191*, *CC.192* and others, are not handled by specific categories of RTEs, but instead by ACSL specifications in Frama-C's libc. These specifications are used by some analyzers and result in warnings and errors when violated.

Note about `ptrdiff_t` *CC.50* deals with pointer subtractions, related to type `ptrdiff_t`. Frama-C does not perform specific handling for this type, but in all standard machdeps defined in Frama-C, its definition is such that pointer subtraction will lead to a signed overflow if the difference cannot be represented in `ptrdiff_t`, thus preventing the undefined behavior. However, if option `-no-warn-signed-overflow` is used, or in a custom machdep, this may not hold.

14.4 C Undefined Behaviors *not* handled by Frama-C

This section lists some of the C undefined behaviors which are currently *not* directly covered by the open-source version of Frama-C.

The list includes UBs which are delegated to other tools, such as the preprocessor or the compiler. Frama-C does not preprocess the sources, relying on external preprocessors such as GCC's or Clang; therefore, related UBs are out of the scope of Frama-C and listed below, even though in practice they are verified by the underlying preprocessor in all but the most exotic architectures.

For a few UBs of syntactic nature, which are always detected during compilation, Frama-C delegates the task to the compiler. This implies that, when running Frama-C directly on the code, it may not complain about them; but the code will generate an error during compilation anyway, *without* requiring specific compilation flags.

This list is not exhaustive; in particular, some UBs not listed here are partially handled by Frama-C. Please contact the Frama-C team if you would like more information about whether a specific set of UBs is handled by the platform.

Table 14.5: C undefined behaviors *not* handled by Frama-C.

UB	***Notes***
CC.2	Syntactic; out of the scope of Frama-C's analyzers.
CC.3	Out of scope (lexing/preprocessing-related).
CC.5	Requires concurrency analysis; the Mthread plugin can provide some guarantees.
CC.14	Only applies to implementations not following IEEE-754. Frama-C's machdeps assume they do.
CC.18	Only applies to implementations not following IEEE-754. Frama-C's machdeps assume they do.
CC.27	Out of scope (lexing/preprocessing-related).
CC.28	Out of scope (lexing/preprocessing-related).
CC.30	Out of scope (lexing/preprocessing-related).
CC.31	Out of scope (lexing/preprocessing-related).
CC.34	Out of scope (lexing/preprocessing-related).
CC.58	Syntactic; delegated to the compiler.
CC.59	Syntactic; delegated to the compiler.
CC.90	Out of scope (lexing/preprocessing-related).
CC.91	Out of scope (lexing/preprocessing-related).
CC.92	Out of scope (lexing/preprocessing-related).
CC.93	Out of scope (lexing/preprocessing-related).
CC.94	Out of scope (lexing/preprocessing-related).
CC.95	Out of scope (lexing/preprocessing-related).
CC.96	Out of scope (lexing/preprocessing-related).

14.5 Common Weakness Enumerations (CWEs) Reported and not Reported by Frama-C

This section lists some CWEs handled by Frama-C, as well as some which are currently out of the scope of Frama-C's default plugins. The latter may be handled by specialized analyses, such as plugins not currently distributed with the open-source release of Frama-C, or in future evolutions of the Frama-C platform.

Note that, due to the large amount of existing CWEs, and the fact that mapping them to undefined

behaviors is not always straightforward, there are likely several other CWEs which are already handled by Frama-C, at least in some contexts. Please contact the Frama-C team if you would like more information about whether a specific set of CWEs is handled by some analysis in the platform.

Remark: this section is partially based on work related to NIST's Software Assurance Metrics and Tool Evaluation (SAMATE[1]), namely its Static Analysis Tool Expositions (SATE), which allowed us to identify some of the CWEs which Frama-C already can or cannot handle. The CWEs listed here are mostly those present in Juliet's 1.3 C/C++ test suite, available at NIST SAMATE's website. Several caveats apply; for instance, some Juliet tests marked as *bad* (that is, exhibiting a weakness) only do so for specific architectural configurations, e.g. 32-bit pointers. If Frama-C is run with a different `machdep`, the test may not exhibit any undefined behavior, as the weakeness is not actually present for such architectures. A rigorous assessment of such situations is necessary for strong claims of soundness.

In Table 14.6, the *Status* column summarizes the current handling of the CWE by Frama-C, as one of the following:

Handled means Frama-C already handles the CWE; in some cases, its detection may be controlled by command-line options;

Partially Handled means the CWE is handled only in some specific cases; Frama-C can thus help *detect* occurrences of this CWE, but not *prove their absence*;

Annotations means the CWE could be handled by Frama-C, but some development is required; in practice, this often means user annotations will be required, e.g. for specifying the source of tainted data; this often implies adding a new abstract domain to Eva, or modifying the analyzer to propagate the data provided by the annotations;

Syntactic means the CWE is of a syntactic nature, while Frama-C specializes in semantic analyses; it is currently not handled by Frama-C and could probably be so, if a user required it; but it is not the primary concern of the platform developers.

Too Vague means the CWE is either too vague or too broad to be formally matched to a given set of properties that Frama-C can verify.

Table 14.6: CWEs handled and not currently handled by Frama-C's open-source plugins. The *Notes* column may contain references to the command-line options table (14.3). The descriptions in the *Status* column are detailed above.

CWE	***Status***	***Notes***
CWE-15: External Control of System or Configuration Setting	Annotations	Requires annotating configuration settings and untrusted sources
CWE-23: Relative Path Traversal	Annotations	Requires annotating path-related functions and untrusted sources
CWE-36: Absolute Path Traversal	Annotations	Requires annotating path-related functions and untrusted sources
CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	Annotations	Requires annotating which functions may inject OS commands
CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')	Annotations	Requires annotating which functions are LDAP-related
CWE-114: Process Control	Annotations	Requires annotating sensitive functions and untrusted sources
CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer	Handled	-

[1] https://samate.nist.gov/Main_Page.html

Table 14.6: CWEs handled and not currently handled by Frama-C's open-source plugins. The *Notes* column may contain references to the command-line options table (14.3). The descriptions in the *Status* column are detailed above.

CWE	***Status***	***Notes***
CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	Handled	-
CWE-121: Stack-based Buffer Overflow	Handled	-
CWE-122: Heap-based Buffer Overflow	Handled	-
CWE-123: Write-what-where Condition	Handled	-
CWE-124: Buffer Underwrite	Handled	-
CWE-125: Out-of-bounds Read	Handled	-
CWE-126: Buffer Overread	Handled	-
CWE-127: Buffer Underread	Handled	-
CWE-128: Wrap-around Error	Handled	Toggled by options `-warn-signed-overflow`, `-warn-unsigned-overflow`, `-warn-signed-downcast` and `-warn-unsigned-downcast`.
CWE-129: Improper Validation of Array Index	Handled	-
CWE-131: Incorrect Calculation of Buffer Size	Handled	Reported not directly at the allocation site, but during usage; the GUI allows navigating back to the source.
CWE-134: Use of Externally-Controlled Format String	Annotations	Requires annotating which format strings come from external sources
CWE-176: Improper Handling of Unicode Encoding	Annotations	Requires annotating Unicode-related functions and variables
CWE-188: Reliance on Data Memory Layout	Partially Handled	Frama-C memory model handles some kinds of invalid accesses
CWE-190: Integer Overflow or Wraparound	Handled	See remarks about *CC.17* in Table 14.3
CWE-191: Integer Underflow	Handled	See remarks about *CC.17* in Table 14.3
CWE-194: Unexpected Sign Extension	Handled	See note about *Numerical Conversions*
CWE-195: Signed to Unsigned Conversion Error	Handled	See note about *Numerical Conversions*
CWE-196: Unsigned to Signed Conversion Error	Handled	See note about *Numerical Conversions*
CWE-197: Numeric Truncation Error	Handled	See note about *Numerical Conversions*
CWE-222: Truncation of Security Relevant Information	Not Handled	-
CWE-223: Omission of Security Relevant Information	Not Handled	-
CWE-226: Sensitive Information in Resource Not Removed Before Reuse	Annotations	Requires annotating shared resources and external entities
CWE-242: Use of Inherently Dangerous Function	Annotations	Requires annotating which functions are "inherently dangerous"
CWE-244: Improper Clearing of Heap Memory Before Release ('Heap Inspection')	Not Handled	Semantic property unavailable at the C memory model; syntactic heuristics can be devised
CWE-252: Unchecked Return Value	Syntactic	-
CWE-253: Incorrect Check of Function Return Value	Syntactic + Annotations	Similar to CWE-252, but also requires annotations defining what "correct check" means
CWE-256: Unprotected Storage of Credentials	Annotations	Requires annotating credential-related functions and variables
CWE-259: Use of Hard-coded Password	Annotations	Requires annotating password-related functions and variables
CWE-272: Least Privilege Violation	Annotations	Requires annotating sensitive functions
CWE-273: Improper Check for Dropped Privileges	Annotations	Requires annotating sensitive functions and forbidden control paths

Table 14.6: CWEs handled and not currently handled by Frama-C's open-source plugins. The *Notes* column may contain references to the command-line options table (14.3). The descriptions in the *Status* column are detailed above.

CWE	***Status***	***Notes***
CWE-284: Improper Access Control	Annotations	Requires annotating privileges, permissions, control paths, etc.
CWE-319: Cleartext Transmission of Sensitive Information	Annotations	Requires annotating sensitive data and transmission-related functions
CWE-321: Hard Coded Cryptographic Key	Annotations	Requires annotating which data correspond to cryptographic keys
CWE-325: Missing Cryptographic Step	Annotations	Requires annotating sequences of valid cryptographic function calls
CWE-327: Use of a Broken or Risky Cryptographic Algorithm	Annotations	Requires annotating which algorithms are "broken or risky"
CWE-328: Reversible One-Way Hash	Annotations	Requires annotating hash-related functions and variables
CWE-338: Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)	Annotations	Requires annotating PRNG-related functions and variables
CWE-364: Signal Handler Race Condition	Not Handled	-
CWE-366: Race Condition Within Thread	Not Handled	Some situations can be handled by the Mthread plugin
CWE-367: TOC TOU	Not Handled	-
CWE-369: Divide by Zero	Handled	-
CWE-377: Insecure Temporary File	Annotations	Requires annotating functions and data flows related to temporary files
CWE-390: Error Without Action	Not Handled	-
CWE-391: Unchecked Error Condition	Not Handled	-
CWE-400: Uncontrolled Resource Consumption	Annotations	Requires annotating resource-related functions and variables
CWE-401: Missing Release of Memory after Effective Lifetime	Not Handled	-
CWE-404: Improper Resource Shutdown or Release	Annotations	Requires annotating resources and functions manipulating them
CWE-415: Double Free	Handled	-
CWE-416: Use After Free	Handled	-
CWE-426: Untrusted Search Path	Not Handled	Not UB-related; requires annotations
CWE-427: Uncontrolled Search Path Element	Annotations	Requires annotating path-related functions and untrusted sources
CWE-440: Expected Behavior Violation	Too Vague	-
CWE-457: Use of Uninitialized Variable	Handled	See remarks about *DD.10*
CWE-459: Incomplete Cleanup	Annotations	Requires annotating resources and cleanup functions
CWE-464: Addition of Data Structure Sentinel	Not Handled	-
CWE-467: Use of sizeof on a Pointer Type	Syntactic	-
CWE-468: Incorrect Pointer Scaling	Syntactic	-
CWE-469: Use of Pointer Subtraction to Determine Size	Handled	-
CWE-475: Undefined Behavior for Input to API	Too Vague	Frama-C already handles some cases related to the C standard library, but in general, may require annotations
CWE-476: NULL Pointer Dereference	Handled	-
CWE-478: Missing Default Case in Switch	Syntactic	-
CWE-479: Signal Handler Use of Non Reentrant Function	Not Handled	-
CWE-480: Use of Incorrect Operator	Too Vague	-
CWE-481: Assigning Instead of Comparing	Syntactic	-

Table 14.6: CWEs handled and not currently handled by Frama-C's open-source plugins. The *Notes* column may contain references to the command-line options table (14.3). The descriptions in the *Status* column are detailed above.

CWE	***Status***	***Notes***
CWE-482: Comparing Instead of Assigning	Syntactic	-
CWE-483: Incorrect Block Delimitation	Syntactic	-
CWE-484: Omitted Break Statement in Switch	Syntactic	-
CWE-506: Embedded Malicious Code	Too Vague	Defining "malicious code" automatically is hard; manual annotations defeat the purpose
CWE-510: Trapdoor	Too Vague	Sound analyses such as those proposed by Frama-C are able to exhaustively identify some kinds of trapdoors
CWE-511: Logic/Time Bomb	Not Handled	-
CWE-526: Exposure of Sensitive Information Through Environmental Variables	Annotations	Requires annotating sensitive information and environment-related functions and variables
CWE-546: Suspicious Comment	Syntactic	-
CWE-561: Dead Code	Partially Handled	Metrics and Eva provide information about syntactic and semantic coverage
CWE-562: Return of Stack Variable Address	Handled	Related to Eva's warning category `locals-escaping`
CWE-563: Unused Variable	Syntactic	Mostly syntactic in nature; compilers often warn about it
CWE-570: Expression Always False	Syntactic	Mostly syntactic in nature; compilers often warn about it
CWE-571: Expression Always True	Syntactic	Mostly syntactic in nature; compilers often warn about it
CWE-587: Assignment of Fixed Address to Pointer	Handled	Detected via `-warn-invalid-pointer` at the assignment, otherwise indirectly at the point of usage; option `-absolute-valid-range` changes its behavior
CWE-588: Attempt to Access Child of a Non-structure Pointer	Partially Handled	Frama-C emits warnings for certain types of incompatible casts
CWE-590: Free Memory Not on Heap	Handled	-
CWE-591: Sensitive Data Storage in Improperly Locked Memory	Not Handled	-
CWE-605: Multiple Binds to the Same Port	Annotations	Requires annotating socket-related functions and variables
CWE-606: Unchecked Loop Condition	Annotations	Requires annotating which data is user-supplied
CWE-615: Inclusion of Sensitive Information in Source Code Comments	Not Handled	-
CWE-617: Reachable Assertion	Handled	Frama-C's libc `assert` specification contains an ACSL assertion
CWE-620: Unverified Password Change	Not Handled	-
CWE-665: Improper Initialization	Handled	-
CWE-666: Operation on Resource in Wrong Phase of Lifetime	Annotations	Requires annotating resources and their lifecycles
CWE-667: Improper Locking	Annotations	Requires annotating locks and functions manipulating them
CWE-672: Operation on Resource After Expiration or Release	Annotations	Requires annotating resources and their lifecycles
CWE-674: Uncontrolled Recursion	Too Vague	The identification of an "uncontrolled" recursion is complex
CWE-675: Duplicate Operations on Resource	Annotations	Requires annotating resources and operations on them
CWE-676: Use of Potentially Dangerous Function	Annotations	Requires annotating which functions are "potentially dangerous"

Table 14.6: CWEs handled and not currently handled by Frama-C's open-source plugins. The *Notes* column may contain references to the command-line options table (14.3). The descriptions in the *Status* column are detailed above.

CWE	***Status***	***Notes***
CWE-680: Integer Overflow to Buffer Overflow	Handled	-
CWE-681: Incorrect Conversion Between Numeric Types	Partially Handled	See note about *Numerical Conversions*
CWE-685: Function Call With Incorrect Number of Arguments	Partially Handled	`-variadic-translation` handles most cases related to variadic function calls
CWE-688: Function Call With Incorrect Variable or Reference as Argument	Partially Handled	Some cases are related to variadic functions (e.g. `printf`) and detected by `-variadic-translation`
CWE-690: Unchecked Return Value to NULL Pointer Dereference	Handled	For functions related to dynamically allocated memory, toggled via option `-eva-alloc-returns-null`
CWE-758: Undefined Behavior	Partially Handled	The C language has too many undefined behaviors, but Frama-C does handle several of them
CWE-761: Free Pointer Not at Start of Buffer	Handled	-
CWE-762: Mismatched Memory Management Routines	Annotations	Requires annotating memory management functions and objects
CWE-773: Missing Reference to Active File Descriptor or Handle	Annotations	Requires annotating resources and operations on them
CWE-775: Missing Release of File Descriptor or Handle	Annotations	Requires annotating resources and their lifecycles
CWE-780: Use of RSA Algorithm Without OAEP	Not Handled	-
CWE-785: Path Manipulation Function Without Max Sized Buffer	Annotations	Requires all relevant filepath-related functions to have correct annotations
CWE-786: Access of Memory Location Before Start of Buffer	Handled	-
CWE-787: Out-of-bounds Write	Handled	-
CWE-788: Access of Memory Location After End of Buffer	Handled	-
CWE-789: Memory Allocation with Excessive Size Value	Annotations	Requires annotating memory limits
CWE-823: Use of Out-of-range Pointer Offset	Handled	-
CWE-824: Access of Uninitialized Pointer	Handled	-
CWE-825: Expired Pointer Dereference	Handled	-
CWE-832: Unlock of Resource That is Not Locked	Annotations	Requires annotating resources and operations on them
CWE-835: Infinite Loop	Partially Handled	Loops which, semantically, are *always* infinite are reported by the Nonterm plugin
CWE-843: Access of Resource Using Incompatible Type ('Type Confusion')	Not Handled	-
CWE-912: Hidden Functionality	Too Vague	Frama-C's sound analysis can show the absence of backdoors, but only if they can be semantically specified (e.g. via annotations)

Note about *Numerical Conversions* A few CWEs (CWE-194, CWE-195, CWE-196 and CWE-197) related to numerical conversion issues do not map directly to undefined behaviors (except, in some cases, to *CC.36*); some of them are related to unspecified and implementation-defined behaviors. Frama-C has a set of command-line options to enable warnings related to these conversions, when they can overflow or generate unexpected values: `-warn-signed-overflow`, `-warn-unsigned-overflow`, `-warn-signed-downcast`, and `-warn-unsigned-downcast`.

See Section 7.3 for more details and some examples related to these options.

Also note that conversion from a floating-point value to an integer may overflow; in this case, an alarm `float_to_int` is generated, independently of the previous options.

Reporting Errors 15

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 15.1.

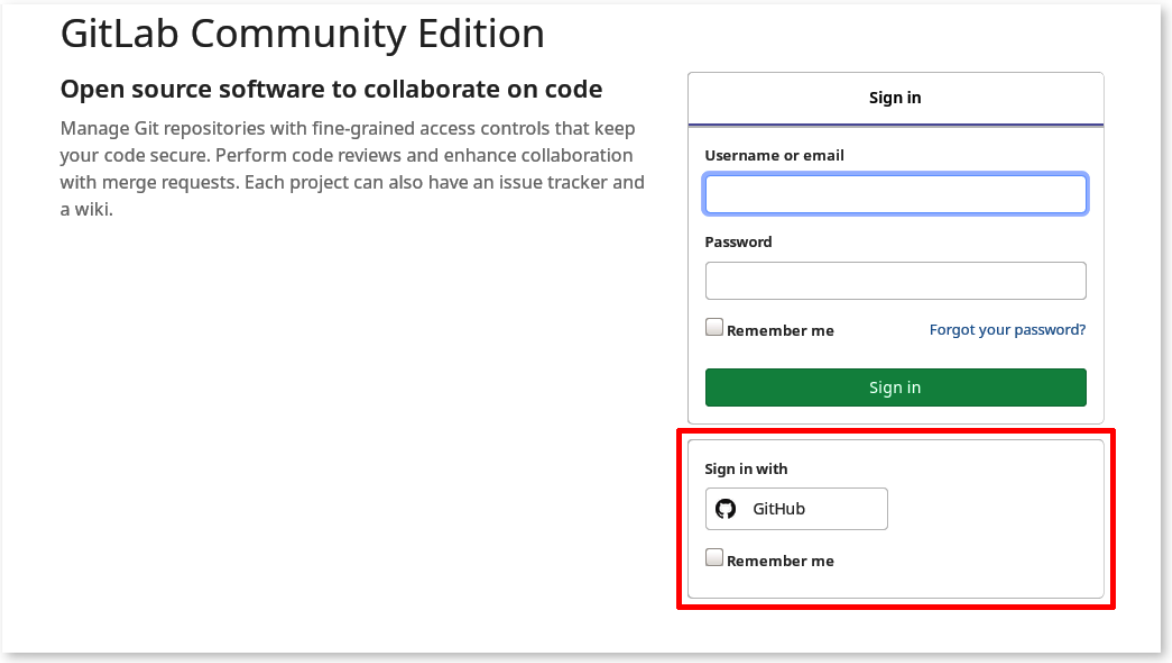


Figure 15.1: The Frama-C Gitlab login page. Note that no direct account creation is possible; you need to sign in via Github unless the Frama-C team provided you an account.

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 15.2.

Please fill the template as precisely as possible, *in English*[1], which helps the Frama-C team more

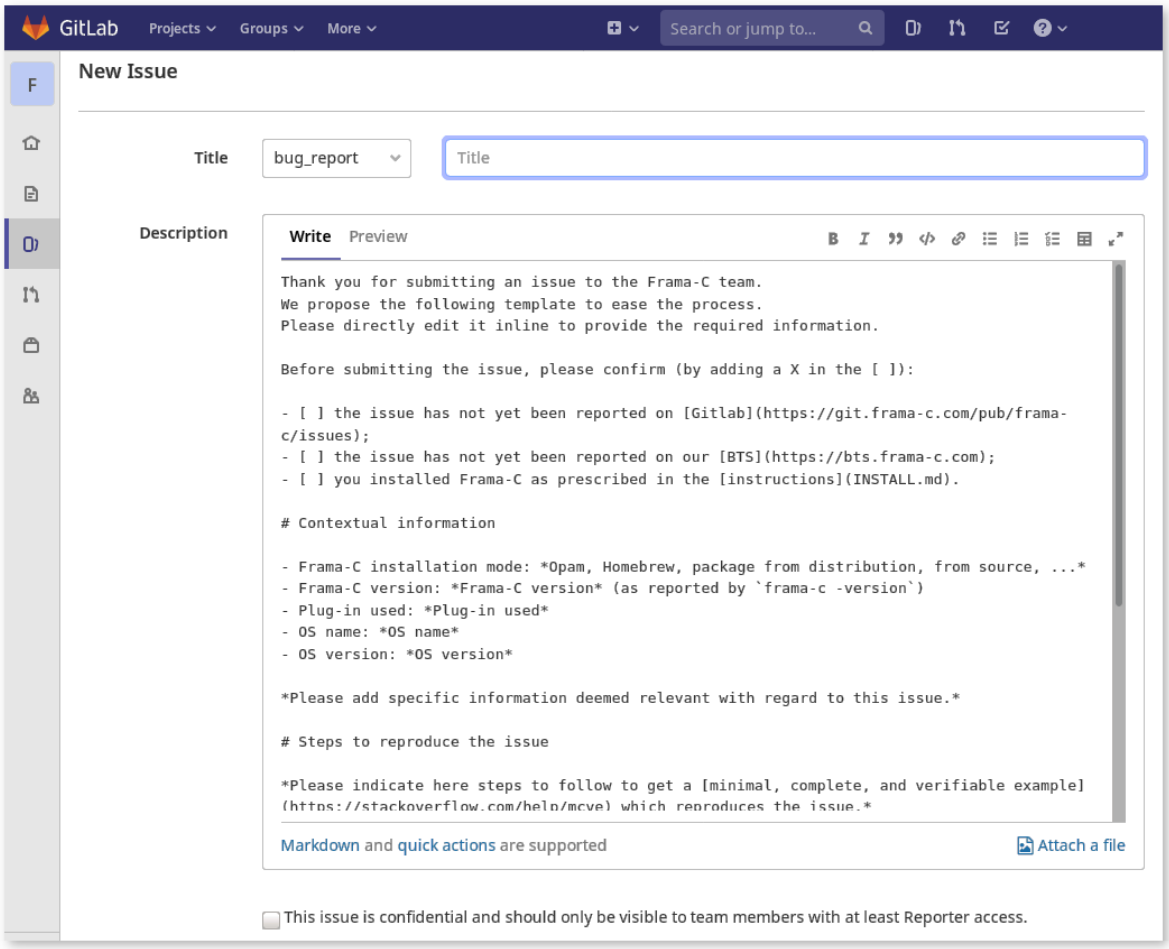


Figure 15.2: The Gitlab new issue page, with the `bug_report` template. The checkbox at the bottom enables marking the issue as private, so that only Frama-C developers can see it.

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.

Changes A

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

32.0 (Germanium)

- **Environment Variables:** added section about sandbox mode.
- **Preparing the Sources:** added section about Mopsa-DB options (`-mopsa-db`, `-mopsa-target`, `-mopsa-list-deps`, `-mopsa-exclude-sources`) and restructured the chapter.
- **Preparing the Sources:** renamed option `-json-compilation-database` to `-compilation-db`.
- **Variadic Translation:** renamed chapter due to Variadic no longer being a plug-in, but a kernel component.
- **Customizing Analyzers:** added option `-warn-unaligned-pointer`.

31.0 (Gallium)

- **Platform-wide Analysis Options:** add section about performance-related parameters.
- **Frama-C Command Line and General Options:** add section about the `-commands-file` option.

30.0 (Zinc)

- **Environment variables:** detail variables and options dedicated to user directories.
- **Normalizing the Source Code:** extend and replace options `-keep-unused-specified-functions` and `-remove-unused-specified-functions` with `-keep-unused-functions`.
- **Preparing the Sources:** remove previously obsoleted options `-continue-annot-error` (replaced by `-kernel-warn-key annot-error`), `-implicit-function-declaration` (replaced by `-kernel-warn-key typing:implicit-function-declaration`), and `-warn-decimal-float` (replaced by `-kernel-warn-key parser:decimal-float`).
- **ACSL extensions:** introduced support for ACSL modules and external module loading via dedicated plug-ins.

29.0 (Copper)

- **Normalizing the Source Code:** document the possibility of undefining builtin macros from the chosen `-machdep`.
- **Preparing the Sources:** loop pragmas are removed and replaced by dedicated ACSL extension. Use `loop unfold` in place of `loop pragma UNROLL`.
- **Preparing the Sources:** add subsection on standard library about portability considerations.

27.0 (Cobalt)

- **Normalizing the Source Code:** new usage of `-machdep`.
- **Normalizing the Source Code:** deprecated option `-c11` (enabled by default).

26.0 (Iron)

- Removed Journalisation
- **Preparing the Sources:** added option `-ast-diff`.
- **Setting Up Plug-ins:** removed options `-add-path` and `-load-script`, and added option `-load-library`.

24.0 (Chromium)

- **Standard library (libc):** Section added.

23.0 (Vanadium)

- **Platform-wide Analysis Options:** swap argument order of `-add-symbolic-path` (now uses `path:name`).

22.0 (Titanium)

- **Getting Started:** added option `-print-config-json`.
- **Getting Started:** added option `-autocomplete`.
- **Getting Started:** updated installation instructions.
- **Preparing the Sources:** added option `-print-cpp-commands`.
- **Reports:** add section about SARIF output (via Markdown Report).

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:** changed 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 5.10).
– **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 5.9).
- **Normalizing the Source Code:** document new option `-custom-annot-char` (Section 5.6)
- **Normalizing the Source Code:** document handling of new file suffix `.ci` (Section 5.4)
- **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`.
- **Preprocessing 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] Mounir Assaf. *From Qualitative to Quantitative Program Analysis: Permissive Enforcement of Secure Information Flow*. PhD thesis, Université Rennes 1, May 2015.

[2] Gergö Barany and Julien Signoles. Hybrid Information Flow Analysis for Real-World C Code. In *International Conference on Tests and Proofs (TAP'17)*, July 2017.

[3] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language. Version 1.16*, November 2020.

[4] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language. Version 1.16 – as implemented in Frama-C 22.0*, November 2020.

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

[6] David Cok. *Frama-C's Frama-Clang plug-in*, 2021. https://www.frama-c.com/download/frama-clang-manual.pdf.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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 [9].

[13] Michaël Marcozzi, Sébastien Bardin, Mickaël Delahaye, Nikolai Kosmatov, and Virgile Prevosto. Taming coverage criteria heterogeneity with ltest. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, pages 500–507, 2017.

[14] Virgile Robles, Nikolai Kosmatov, Virgile Prévosto, Louis Rilling, and Pascale Le Gall. MetAcsl: Specification and Verification of High-Level Properties. In *TACAS 2019*, volume 11427 of *LNCS*, Prague, Czech Republic, April 2019.

[15] 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.

[16] Julien Signoles, Basile Desloges, and Kostyantyn Vorobyov. *Frama-C's E-ACSL Plug-in*. https://frama-c.com/fc-plugins/e-acsl.html.

[17] Boris Yakobowski and Richard Bonichon. *Frama-C's Mthread plug-in*, 2012. http://frama-c.com/download/frama-c-mthread-manual.pdf.

List of Figures

5.1 Overview of source preparation steps: as performed by GCC (top) and as performed by Frama-C (bottom). 25

10.1 Screenshot of Ivette, illustrating several components 46

11.1 Classification Rule JSON Format 50
11.2 Classified Event JSON Format 51

13.1 Overview of the *analysis-scripts* workflow: the inputs on the left produce a GNUmakefile which is then used for parsing, analyzing and visualizing results. 61

15.1 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. 75
15.2 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. 76

Index

-<plug-in>-debug, ***35***
-<plug-in>-help, ***35***
-<plug-in>-log, ***17***
-<plug-in>-msg-key, ***35***
-<plug-in>-verbose, ***35***
-<plug-in>-warn-key, ***36***
__FC_*, ***31***
__FC_FORCE_POSIX_C_SOURCE, ***31***
__FC_INDETERMINABLE_FLOATS, ***31***
__FC_MACHDEP_*XXX*, ***30***
__FC_NO_MONOTONIC_CLOCK, ***31***
__FRAMAC__, ***30***

-absolute-valid-range, ***36***, 44
-add-symbolic-path, ***36***
-aggressive-merging, ***26***
-allow-duplication, ***26***
-annot, ***26***
-asm-contracts, ***26***
-asm-contracts-auto-validate, ***27***
-ast-diff, ***30***
-autocomplete, ***16***

Batch version, 14
-big-ints-hex, ***18***

-c, ***18***, ***19***
C compiler, 13
C preprocessor, 13
C99 ISO standard, 10
-cache-size <1-10>, ***39***
-collapse-call-cast, ***27***
-commands-file, ***18***
-compilation-db, ***24***
-constfold, ***27***
-cpp-command, ***19***, 25, ***25***, 28
-cpp-extra-args, 15, ***24***
-cpp-extra-args-per-file, ***24***
-cpp-frama-c-compliant, 25, ***25***

-debug, ***17***, 44

-enums, ***27***

-eva, 16
-eva-use-spec, 15
-explain, ***14***, 44

-float-hex, ***18***
-float-normal, ***18***
-float-relative, ***18***
frama-c, 14
frama-c-gui, 14, 46, 47
frama-c-script, 14
-frama-c-stdlib, ***32***
FRAMAC_SESSION, ***19***

-generated-spec-custom
<clause_1:mode_1,clause_2:mode_2,...>,
29
-generated-spec-mode, 29
-generated-spec-mode <mode>, ***29***

-h, ***14***
-help, ***14***, 44
--help, ***14***

-initialized-padding-locals, ***27***
-inline-calls, ***27***, 28
Installation, 13
Interactive version, 14

-keep-comments, ***18***
-keep-switch, ***27***
-keep-unused-functions, ***27***
-keep-unused-types, ***27***
-kernel-debug, ***17***, ***35***
-kernel-h, ***14***
-kernel-help, ***14***, ***35***
-kernel-log, ***17***
-kernel-msg-key, ***35***
-kernel-verbose, ***17***, ***35***
-kernel-warn-key, ***36***

-lib-entry, ***35***
-load, ***43***, 44
-load-library, ***21***
-load-module, ***19***

-load-plugin, ***21***

-machdep, ***27***, 30, ***31***
-main, ***35***
-memory-footprint, ***39***
-mopsa-db, ***23***
-mopsa-exclude-sources, ***24***
-mopsa-list-deps, ***24***
-mopsa-target, ***23***

-no-autoload-plugins, ***21***
-no-cpp-frama-c-compliant, 32
-no-unicode, 15
-no-variadic-strict, ***55***
-no-variadic-translation, ***55***

OCaml compiler, 13
OCAMLPATH, 21
-ocode, ***18***
opam, 13
Options, 14

-permissive, ***24***, ***36***
Plug-in
    External, 21, ***21***
    Internal, ***21***
-plugins, ***14***
-pp-annot, ***26***
-print, ***18***, 26
-print-config, ***14***
-print-config-json, ***15***
-print-cpp-commands, ***24***
-print-lib-path, ***14***, 21
-print-libc, ***18***
-print-plugin-path, ***14***, 21
-print-share-path, ***14***, 21
-print-version, ***14***
Project, ***43***

-quiet, ***17***, 44

-remove-inlined, ***28***
-remove-projects, ***43***
-remove-unused-types, ***27***
report, ***48***

-safe-arrays, ***37***
-save, ***43***, 44
-semantic-const-fold, 16
-session, ***19***
-set-project-as-default, ***16***
-simplify-cfg, ***28***
-simplify-trivial-loops, ***28***
-slevel-function, 15

-then, ***16***
-then-last, 16, ***16***, 43
-then-on, ***16***, 43
-then-replace, ***16***, 43
-time, ***18***
-tty, ***18***
-typecheck, ***32***

-ulevel, 16, 26, ***28***, 44
-ulevel-force, ***28***
-unicode, ***18***, 44
-unsafe-arrays, ***36***
-unspecified-access, ***37***

-variadic-translation, 53, ***53***, 54–56
-verbose, ***17***, 44
-version, ***14***

-warn-invalid-bool, ***39***
-warn-invalid-pointer, ***37***
-warn-left-shift-negative, ***38***
-warn-pointer-downcast, ***38***
-warn-right-shift-negative, ***38***
-warn-signed-downcast, ***38***
-warn-signed-overflow, ***38***
-warn-special-float, ***38***
-warn-unaligned-pointer, ***38***
-warn-unsigned-downcast, ***38***
-warn-unsigned-overflow, ***38***
warning status, 36
-wp-no-print, 15
-wp-print, 15