frama C

Software Analyzers

User Manual

Frama-C User Manual

Release Carbon-20110201

Loïc Correnson, Pascal Cuoq, Armand Puccetti and Julien Signoles

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

©2009 CEA LIST

Contents

Foreword **7**

1 Introduction **9**
1.1 About this document . . . 9
1.2 Outline . . . 9

2 Overview **11**
2.1 What is Frama-C? . . . 11
2.2 Frama-C as a Static Analysis Tool . . . 11
2.2.1 Frama-C as a Lightweight Semantic-Extractor Tool . . . 12
2.2.2 Frama-C for Formal Verification of Critical Software . . . 12
2.3 Frama-C as a Tool for C programs . . . 12
2.4 Frama-C as an Extensible Platform . . . 12
2.5 Frama-C as a Collaborative Platform . . . 13
2.6 Frama-C as a Development Platform . . . 13

3 Getting Started **15**
3.1 Installation . . . 15
3.2 One Framework, Four Executables . . . 16
3.3 Frama-C Command Line and General Options . . . 16
3.3.1 Splitting Frama-C Execution in Several Steps . . . 17
3.3.2 Getting Help . . . 17
3.3.3 Frama-C Version . . . 18
3.3.4 Verbosity and Debugging Levels . . . 18
3.3.5 Getting time . . . 18
3.3.6 Inputs and Outputs of source code . . . 18
3.4 Environment Variables . . . 18
3.4.1 Variable `FRAMAC_LIB` . . . 19
3.4.2 Variable `FRAMAC_PLUGIN` . . . 19
3.4.3 Variable `FRAMAC_SHARE` . . . 19
3.5 Exit Status . . . 19

4 Working with Plug-ins **21**
4.1 The Plug-in Taxonomy . . . 21
4.2 Installing Internal Plug-ins . . . 21
4.3 Installing External Plug-ins . . . 22
4.4 Using Plug-ins . . . 22

5 Preparing the Sources **25**
5.1 Pre-processing the Source Files . . . 25
5.2 Merging the Source Code files . . . 26
5.3 Normalizing the Source Code . . . 26
5.4 Testing the Source Code Preparation . . . 27

6 Analysis Options **29**
6.1 Entry Point . . . 29
6.2 Customizing Analyzers . . . 29

7 General Kernel Services **31**
7.1 Projects . . . 31
7.1.1 Creating Projects . . . 31
7.1.2 Using Projects . . . 31
7.1.3 Saving and Loading Projects . . . 32
7.2 Dependencies between Analyses . . . 32
7.3 Journalisation . . . 33

8 Graphical User Interface **35**
8.1 Frama-C Main Window . . . 35
8.2 Menu Bar . . . 36

9 Reporting Errors **39**

A Changes **43**

Bibliography **45**

List of Figures **47**

Index **49**

Foreword

This is the user manual of Frama-C[1]. The content of this document corresponds to the version Carbon-20110201 (February 7, 2011) of Frama-C. However the development of Frama-C is still ongoing: features described here may still evolve in the future.

Acknowledgements

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

[1] http://frama-c.cea.fr

Chapter 1

Introduction

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

This manual gives an overview of Frama-C for newcomers, and serves as a reference for experimented users. It only describes those platform features that are common to all analyzers. Thus it *does not cover* use of the analyzers provided in the Frama-C distribution (Value Analysis, Slicing, ...). Each of these analyses has its own specific documentation [3]. Furthermore, the development of new analyzers is described in the Plug-in Development Guide [4].

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 like ***1*** reference the defining sections for the corresponding entries while other numbers (like 1) are less important references.

Most important paragraphs are displayed inside a gray box 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 how to work with plug-ins.

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

Chapter 6 gives some general options for parametrising analyzers.

Chapter 7 introduces general services offered by the platform.

Chapter 8 gives a detailed description of the graphical user interface of Frama-C.

Chapter 9 explains how to report errors *via* the Frama-C's Bug Tracking System.

Chapter 2

Overview

2.1 What is Frama-C?

Frama-C is a platform dedicated to the static analysis of source code written in C. The Frama-C platform gathers several static analysis techniques into a single collaborative extensible framework. The collaborative approach of Frama-C allows static analyzers to build upon the results already computed by other analyzers in the framework. Thanks to this approach, Frama-C provides very sophisticated tools such as a slicer and dependency analysis.

2.2 Frama-C as a Static Analysis Tool

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

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

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

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

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

of the analyses provided by Frama-C require comparatively little intervention from the user, and the collaborative approach proposed in Frama-C allows the user to get some impressive results.

2.2.1 Frama-C as a Lightweight Semantic-Extractor Tool

Frama-C analyzers may be useful for better understanding a C program by extracting semantic information from its code.

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 allows to verify that source code complies with provided formal specifications.

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

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

2.3 Frama-C as a Tool for C programs

Frama-C analyses C programs.

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

Furthermore, 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 extensible.

It is is organized with a plug-in architecture (comparable to that of the Gimp or Eclipse): each analyzer comes in the form of a *plug-in* and is connected to the platform itself, or *kernel*.

Several ready-to-use analyses are included in the Frama-C distribution. This manual covers the set of features common to all plug-ins. It does not cover use of the plug-ins that come in the Frama-C distribution (Value Analysis, Functional Dependencies, *etc*). Each of these analyses has its own specific documentation [3].

Additional plug-ins can be provided by third-party developers and installed separately from the kernel.

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 Value Analysis plug-in and of the Functional Dependencies plug-in.

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

With Frama-C, it will be possible to take advantage of the complementarity of existing analysis approaches. It will be 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 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 also to be interested in writing his/her 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 latitude 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 static 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 [4].

Chapter 3

Getting Started

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

3.1 Installation

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

It is usually easier to install Frama-C from one of the binary distributions than from the source distribution. The pre-compiled binaries include many of the required libraries and other dependencies, whereas installing from source requires these dependencies already to have been installed.

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

A C pre-processor is required for *using* Frama-C on C files. By default, Frama-C tries to use `gcc -C -E I.` as pre-processing command, but this command can be customized (see Section 5.1). If you do not have any C pre-processor, you can only run Frama-C on already pre-processed `.i` file.

A Unix-like compilation environment is mandatory and shall have at least the tool `GNU make`[1] version 3.81 or higher, as well as various POSIX commands.

The OCaml compiler is required both for compiling Frama-C from source *and* for compiling additional plug-ins. Version 3.10.2 or higher[2] must be used.

Support of some plug-ins in native compilation mode (see Section 3.2) requires the so-called *native dynamic linking* feature of OCaml. It is only available in the most recent versions of OCaml (at least 3.11.0) and only on a subset of supported platforms.

Gtk-related packages: GTK+[3] version 2.4 or higher, GtkSourceView[4] version 2.x, GnomeCanvas[5] version 2.x as well as LablGtk[6] version 2.14 or higher are required for building

[1] http://www.gnu.org/software/make
[2] http://caml.inria.fr
[3] http://www.gtk.org
[4] http://projects.gnome.org/gtksourceview
[5] http://library.gnome.org/devel/libgnomecanvas
[6] http://wwwfun.kurims.kyoto-u.ac.jp/soft/lsl/lablgtk.html

the Graphical User Interface (GUI) of Frama-C.

OcamlGraph package: Frama-C will make use of OcamlGraph[7] if already installed in version 1.4 or higher. Otherwise, Frama-C will install a local and compatible version of this package by itself. This dependency is thus non-mandatory for Frama-C.

3.2 One Framework, Four Executables

Frama-C installs four executables[8], namely:

- `frama-c`: native-compiled batch version;
- `frama-c.byte`: bytecode batch version;
- `frama-c-gui`: native-compiled interactive version;
- `frama-c-gui.byte`: bytecode interactive version.

The differences between these versions are described below.

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

The native-compiled version is usually ten times faster than the bytecode one. The bytecode version supports dynamic loading on all architectures, and is able to provide better debugging information. Use the native-compiled version unless you have a reason to use the bytecode one.

***batch* vs *interactive*:** The interactive version allows to use a GUI to select the set of files to analyze, position options, launch analyses, browse the code and observe analysis results at one's convenience (see Chapter 8 for details).

With the batch version, all settings and actions must be provided on the command-line. This is not possible for all plug-ins, nor is it always easy for beginners. Modulo the limited user interactions, the batch version allows the same analyses as the interactive version[9]. A batch analysis session consists in launching Frama-C in a terminal. Results are printed on the standard output.

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

3.3 Frama-C Command Line and General Options

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

[7] http://ocamlgraph.lri.fr

[8] On Windows OS, the usual extension `.exe` is added to each file name.

[9] For a single analysis project. Multiple projects can only be handled in the interactive version or programmatically. see Section 7.1

batch version can be made to launch the value analysis on the `foo.c` file with the command `frama-c -val foo.c`. Although the GUI allows to select files and to launch the value analysis interactively, the command `frama-c-gui -val foo.c` can be used to launch the value analysis on the file `foo.c` and starts displaying the results immediately in the GUI.

Any option requiring an argument may use the following format:

```
-option_name value
```

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

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 `-no-unicode` for limiting the messages to ACSII. Plug-ins 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 `-ltl-acceptance` is `-ltl-no-acceptance`.

3.3.1 Splitting Frama-C Execution in Several Steps

By default, Frama-C parses its command line in an *unspecified* order and runs its actions accordingly to the read options. To enforce an order of execution, you have to use the option `-then`: Frama-C parses its command line untill 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 accordingly 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 -val -ulevel 4 file.c -then -ulevel 5
```

It first runs the value analysis plug-in (option `-val`, [3]) with an unrolling level of 4 (option `-ulevel`, Section 5.3). Then it re-runs the value analysis plug-in (option `-val` is still set) with an unrolling level of 5.

It is also possible to specify a project (see Section 7.1) on which the actions applied thanks to the option `-then-on`. Consider for instance the following command.

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

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.

3.3.2 Getting Help

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

Furthermore, either the option `-help` or `-h` or `--help` lists all available plug-ins.

3.3.3 Frama-C Version

The current version of the Frama-C kernel can be obtained with the option `-version`. This option also prints the different paths where Frama-C searches objects when required.

3.3.4 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 name>-verbose` (resp. `-<plug-in name>-debug`) set the level of verbosity (resp. debugging) of the kernel and particular plug-ins. While 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.

3.3.5 Getting time

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

3.3.6 Inputs and Outputs of source code

The following options deal with inputs and outputs 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 5.3).

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

`-float-digits <n>` displays n digits when printing floats. Defaults to 12.

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

3.4 Environment Variables

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

3.4.1 Variable `FRAMAC_LIB`

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

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

3.4.2 Variable `FRAMAC_PLUGIN`

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

3.4.3 Variable `FRAMAC_SHARE`

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

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

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 an user input was invalid;

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 9);

4 Frama-C exits on an internal error. Please report a "bug report" on the Bug Tracking System (see Chapter 9);

5 Frama-C raises an error while exiting normally.

6 Frama-C raises an error while exiting abnormally.

127 Frama-C exits abnormally on an unknown error. Please report a "bug report" on the Bug Tracking System (see Chapter 9).

Chapter 4

Working with Plug-ins

The Frama-C platform has been designed to support third-party plug-ins. In the present chapter, we present how to configure, compile, install, run and update such extensions. This chapter does not deal with the development of new plug-ins (see the Plug-in Development Guide [4]). It does not deal with usage of plug-ins, which is the purpose of individual plug-in manuals.

4.1 The Plug-in Taxonomy

It is possible to distinguish 2×2 kinds of plug-ins: *internal* vs *external* plug-ins and *static* vs *dynamic* plug-ins. These different kinds are explained below.

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

static* vs *dynamic Static plug-ins are statically linked into a Frama-C executable (see Section 3.2) while dynamic plug-ins are loaded by an executable when it is run. Despite only being available on some environments (see Section 3.1), dynamic plug-ins are more flexible as explained in Section 4.4.

4.2 Installing Internal Plug-ins

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

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

Internal dynamic plug-ins may be linked statically. This is achieved by passing `configure` the option `--with-<plug-in name>-static`. It is also possible to force all dynamic plug-ins to be linked statically with the option `--with-all-static`. This option is set by default on systems unsupporting native dynamic loading.

4.3 Installing External Plug-ins

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

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

External plug-ins are always dynamic plug-ins by default. On systems where native dynamic linking is not supported, a new executable, called `frama-c-<plug-in name>`[1], is automatically generated when an external plug-in is compiled. This executable contains the Frama-C kernel, all the static plug-ins previously installed and the external plug-in. On systems where native dynamic linking is available, this executable is not necessary for normal use but it may be generated with the command `make static`.

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

4.4 Using Plug-ins

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 message).

Please refer to each plug-in's documentation for specific options.

At launch, Frama-C loads all dynamic plug-ins it finds if the option `-dynlink` is set. That is the normal behavior: you have to use its opposite form `-no-dynlink` in order to not load any dynamic plug-in. When loading dynamic plug-ins, Frama-C searches for them in directories indicated by `frama-c -print-plugin-path` (see Section 3.4.2. Frama-C can locate plug-ins in additional directories by using the option `-add-path <paths>`. Yet another solution to load a dynamic plug-in is to set the `-load-module <files>` or `-load-script <files>` options, using in both cases a comma-separated list of file names without any extension. The former option loads the specified OCaml object files into the Frama-C runtime, while the latter tries to compile the source files before linking them to the Frama-C runtime.

[1] With the extension `.exe` on Windows OS

In general, dynamic 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.
The `-load-script` option requires the OCaml compiler that was used to compile Frama-C to be available and the Frama-C compiled library to be found (see Section 3.4.1).

Chapter 5

Preparing the Sources

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

5.1 Pre-processing the Source Files

The list of files to analyse must be provided on the command line. An alternative is to choose the files interactively in the GUI. Files with the `.i` suffix are assumed to be already pre-processed C files. Frama-C pre-processes the other files with the following command.

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

The option `-cpp-command` may be used to change the default pre-processing command. If patterns `%1` and `%2` do not appear in the provided command, the pre-processor is invoked in the following way.

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

In this command, `<output file>` is chosen by Frama-C while `<input file>` is one of the filenames provided by the user. It is also possible to use the patterns `%1` and `%2` in the command as place-holders for the input files and the output file respectively. Here are some examples for using this option.

```
$ frama-c -cpp-command 'gcc -C -E -I. -x c' file1.src file2.i
$ frama-c -cpp-command 'gcc -C -E -I. -o %2 %1' file1.c file2.i
$ frama-c -cpp-command 'cp %1 %2'  file1.c file2.i
$ frama-c -cpp-command 'cat %1 > %2' file1.c file2.i
$ frama-c -cpp-command 'CL.exe /C /E %1 > %2' file1.c file2.i
```

Additionally the option `-cpp-extra-args` allows the user to extend the pre-processing command.

By default, ACSL annotations are not pre-processed. Pre-processing them requires *to use gcc as pre-processor* and to put the option `-pp-annot` on the Frama-C command line.

An experimental incomplete specific C standard library is bundled with Frama-C and installed in the sub-directory `libc` of the directory `D` printed by `frama-c -print-share-path`. It contains standard C headers, some ACSL specifications and definitions for some library functions. You may use the following command:

```
$ frama-c -cpp-extra-args='-ID/libc -nostdinc' D/libc/fc_runtime.c <input file>
```

Note that this standard library is customized for 32 bits little endian architecture. For other configurations you have to manually edit the file `D/libc/__fc_machdep.h`.

5.2 Merging the Source Code files

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

Frama-C aborts whenever any error occurs during one of these steps. However users can use the option `-continue-annot-error` in order to continue after emiting a warning when an ACSL annotation fails to type-check.

5.3 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 take place exclusively on the normalized version of the source code. The normalized version may be printed by using the option `-print` (see Section 3.3.6).

The following options allow to customize the normalization.

`-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 duplicate 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.

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

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

`-continue-annot-error` just emits a warning and discards the annotation when it fails to type-check, instead of generating an error (errors in C are still fatal).

`-force-rl-arg-eval` forces right to left evaluation order of function arguments. C standard does not enforce any evaluation order, and the default is thus to leave it unspecified.

`-keep-switch` preserves `switch` statements in the source code. Without this option, they are transformed into `if` statements. An experimental plug-in may forgot the treatment of the `switch` construct and require this option not to be used. Other plug-ins may prefer this option to be used because it better preserves the structure of the original program.

`-machdep <machine architecture name>` defines the target platform. The default value is a `x86-32bits` 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.

The list of supported platforms can be obtained by typing:

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

`-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 set by default.

`-ulevel <n>` unrolls all loops n times. This is a purely syntactic operation. Loops can be unrolled individually, by inserting the `UNROLL` pragma just before the loop statement. Do not confuse this option with plug-in-specific options that may also be called "unrolling" [3]. Below is a typical example of use.

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

5.4 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 some options for pre-processing or normalizing the source code, you can use the option `-type-check` for the same purpose. For instance:

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

Chapter 6

Analysis Options

The analysis options described in this chapter provide hypotheses that influence directly the behavior of analyzers. For this reason, the user must understand them and the interpretation the plug-ins he uses have of them.

6.1 Entry Point

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

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

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

6.2 Customizing Analyzers

The descriptions of the analysis options follow. For the first two, the description comes from the Value Analysis manual [3]. Furthermore, these options are very likely to be modified in future versions of Frama-C.

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

`-no-overflow` instructs the analyzer to assume that integers are not bounded and that the analyzed program's arithmetic is exactly that of mathematical integers. This option should only be used for codes that do not depend on specific sizes for integer types and do not rely on overflows. For instance, the following program is analyzed as "non-terminating" in this mode.

```
void main(void) {
  int x=1;
  while(x++);
  return;
}
```

The option `-no-overflow` should only be activated when it is guaranteed that the sizes of integer types do not change the concrete semantics of the analyzed code. Beware: voluntary overflows that are a deliberate part of the implemented algorithm are easy enough to recognize and to trust during a code review. Unwanted overflows, on the other hand, are rather difficult to spot using a code review. The next example illustrates this difficulty.

Consider the function `abs` that computes the absolute value of its `int` argument:

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

With the `-no-overflow` option, the result of this function is a positive integer, for whatever integer passed to it as an argument. This property is not true for a conventional architecture, where `abs(MININT)` overflows and returns `MININT`. Without the `-no-overflow` option, on the other hand, the value analysis detects that the value returned by this function `abs` may not be a positive integer if `MININT` is among the arguments.

The option `-no-overflow` may be modified or suppressed in a later version of the plug-in.

`-unsafe-arrays` is useful when the source code manipulates arrays within structures. It assumes that accesses in the array are always within the correct bounds. No warnings are then emitted about possible out-of-bounds. The opposite option, called `-safe-arrays`, is set by default.

`-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 aims at warn against such ambiguous situations.

Chapter 7

General Kernel Services

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

7.1 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 one 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'.

7.1.1 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.

7.1.2 Using Projects

The list of existing projects of a given session is visible in the graphical mode through the `Project` menu (see Section 8.2). Among other actions on projects (duplicating, renaming, removing, saving*etc*), this menu allows the user to switch between different projects during the same session.

In the batch mode, it is not possible to handle a multi-project session: there is no way to switch from one project to another one through the command line.

7.1.3 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. 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 explicitely set in the current session.

Consider for instance the following command.

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

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 7.1 *Saving the result of a time-consuming analysis before trying to use it in different settings may be a good idea.*

Beware that all the existing projects are deleted, even if an error occurs when reading the file. Thus, in the GUI, do not hesitate to save the existing projects in some file before loading another file.

7.2 Dependencies between Analyses

Usually analyses do have parameters (see Chapter 6). Whenever values of parameters changes, results of the analyses may change. In order to not display inconsistent results according to 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 -unsafe-arrays -val 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 5.3). For computing its result, the value analysis assumes that accesses in the array are always within the correct bound (option `-unsafe-arrays`, see Section 6.2). Just after, Frama-C saves the results on file `foo.sav` and exists.

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 -unsafe-arrays -val foo.c
$ frama-c -load foo.sav -safe-arrays
```

The first commands produces the very same result than above. However, when loading, Frama-C knows that one parameter changed. Thus it discards the saved results of the value analysis and recomputed it on the same source code by using the parameter `-ulevel 5 -safe-arrays` (and the default value of each other parameter).

In the same way, results of one analysis A_1 may depend on results of another one A_2. So, whenever results of A_2 change, Frama-C automatically discards results of A_1. For instance, slicing results depend on value analysis results. Thus the slicing results are discarded whenever the value analysis ones are.

7.3 Journalisation

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

By default, the name of the journal is `frama_c_journal.ml`, but it can be modified by using the option `-journal-name`.

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

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

- Replay easily a given set of analysis operations in order to reach a certain state. Once the final state is reached, further analyses can be performed normally. Beware that journals may be source dependent and thus can not necessarily be reused on different source codes to perform the same analyses.
- Act as a macro language for plug-in developers. They can perform some wished actions on the GUI to generate a journal and then adapt it to perform a more general but similar task.
- Debugging. In the GUI, a journal is always generated, even when an error occurs. The output journal usually contains information about this error. Thus it provides an easy way to reproduce the very same error. Consequently, it is advised to attach the journal when reporting an error in the Frama-C BTS (see Chapter 9).

By default, a journal is generated upon exit of the session only whenever Frama-C crashes in graphical mode. In all other cases, no journal is generated. This behaviour may be customized by using the option `-journal-enable` (resp. `-journal-disable`) that generates (resp. does not generate) a journal upon exit of the session.

Chapter 8

Graphical User Interface

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

8.1 Frama-C Main Window

Upon launching Frama-C in graphical mode on some C files, the following main window is displayed (figure 8.1):

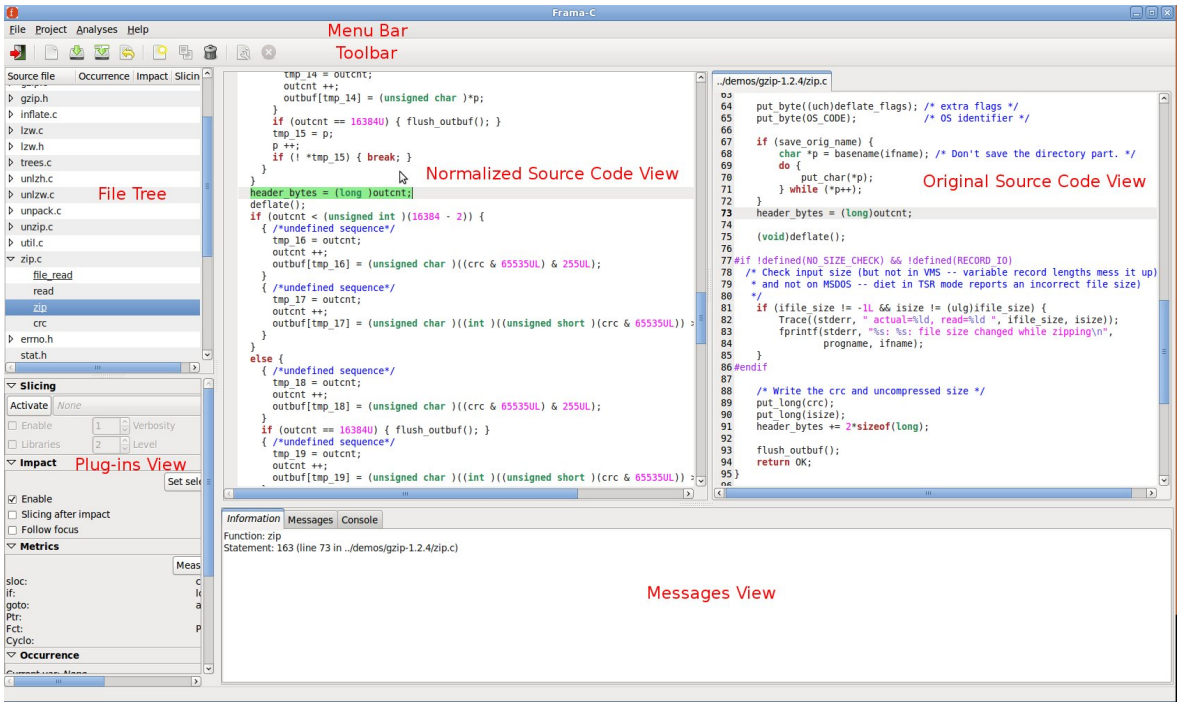

Figure 8.1: Initial View

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

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

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

The File Tree provides a tree-like structure of the source files involved in the current analysis, as well as the functions they contain. Plug-ins may also display specific information for each file and/or function.

The normalized and original source code views display the source code of the current selected element of the file tree and its normalised code (see Section 5.3). Left-clicking on an object (statement, left-value, *etc*) in the normalized source code view displays information about it in the "Information" page of the Messages View and displays the corresponding object of the original source view, while right-clicking on them open a contextual menu. Items of this menu depends of the kind of the selected object and of available plug-ins.

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

The Plug-ins View shows specific interface of plug-ins. The interface of each plug-in can be hidden.

The Messages View contains three different pages, namely:

the "Information" page which provides brief details on the currently selected object.

the "Messages" page shows most important messages, especially all the alarms, that the Frama-C kernel or plug-ins generated. Please refer to the specific documentation of each plug-in in order to get the exact form of alarms. These alarms are not stored in batch mode (for reducing memory consumption): thus this panel remains empty if the GUI loads a file saved in batch mode (see Section 7.1.3). If you want to store these alarms in batch mode, use the option `-collect-messages`.

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

8.2 Menu Bar

The menu bar is organised as follows:

File Menu proposes items for managing the current session.

Item `Set C source files` allows to change the analyzed files of the current project.

Item `Save session` saves all the current projects into a file. If the use never chooses such a file, a dialog box is opened for choosing one.

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

Item `Load Session` opens a previously saved session.

This fully resets the current session (see Section 7.1.3).

Item `Quit` exits Frama-C without saving.

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

Analyses Menu provides items for configuring and running plug-ins.

- Item `Configure and run analyses` opens the dialog box shown Figure 8.2, that allows to set all Frama-C parameters and to re-run analyses according to changes.

Figure 8.2: The Analysis Configuration Window

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

The toolbar also provides a button `Stop` which halts the running analyses and restores Frama-C in the last enable valid configuration.

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

Help Menu provides help items.

Chapter 9

Reporting Errors

If Frama-C crashes or behaves abnormally, you are invited to bug report *via* the Frama-C Bugs Tracking System (BTS) located at http://bts.frama-c.com.

Opening a BTS account is required for such a task.

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

Reporting a new issue open a webpage similar to the one shown Figure 9.1. This page has also a link to an advanced bugs reporting page that allows you to write a more detailed report. The different fields of these forms shall be filled *in English*[1] as precisely as possible, in order for the maintenance team to understand and track the problem down easily.

Below are some recommendations for this purpose[2]:

Category: select as appropriate.

Reproducibility: select as appropriate.

Severity: select the level of severity. Levels are shown in increasing order of severity.

Profile or Platform, OS and OS Version: enter your hardware and OS characteristics.

Product Version and Product Build this can be obtained through the command `frama-c -version`, see Section 3.3.3.

Summary: give a brief one line description of the nature of your bug.

Description: first, explain the *actual behavior*, that is what you actually observe on your system. Then, describe your *expected behavior* of Frama-C, that is the results you expect instead. A "bug" is sometimes due to a misunderstanding of the tool's behaviour or a misunderstanding of its results, so providing both behaviors is an essential part of the report. Please do clearly separate both parts in the description.

Steps to reproduce: provide everything necessary for a maintainer to reproduce the bug: input files, commands used, sequence of actions, *etc.* If the bug appears through the Frama-C GUI, it may be useful to attach the generated journal (see Section 7.3). Beware that this journal **does not** replace nor contain the input files, that must be added to the bug report too (see below).

[1] French is also possible for private entries.

[2] You can also have a look at the associated Frama-C wiki: http://bts.frama-c.com/dokuwiki/doku.php?id=mantis:frama-c:start.

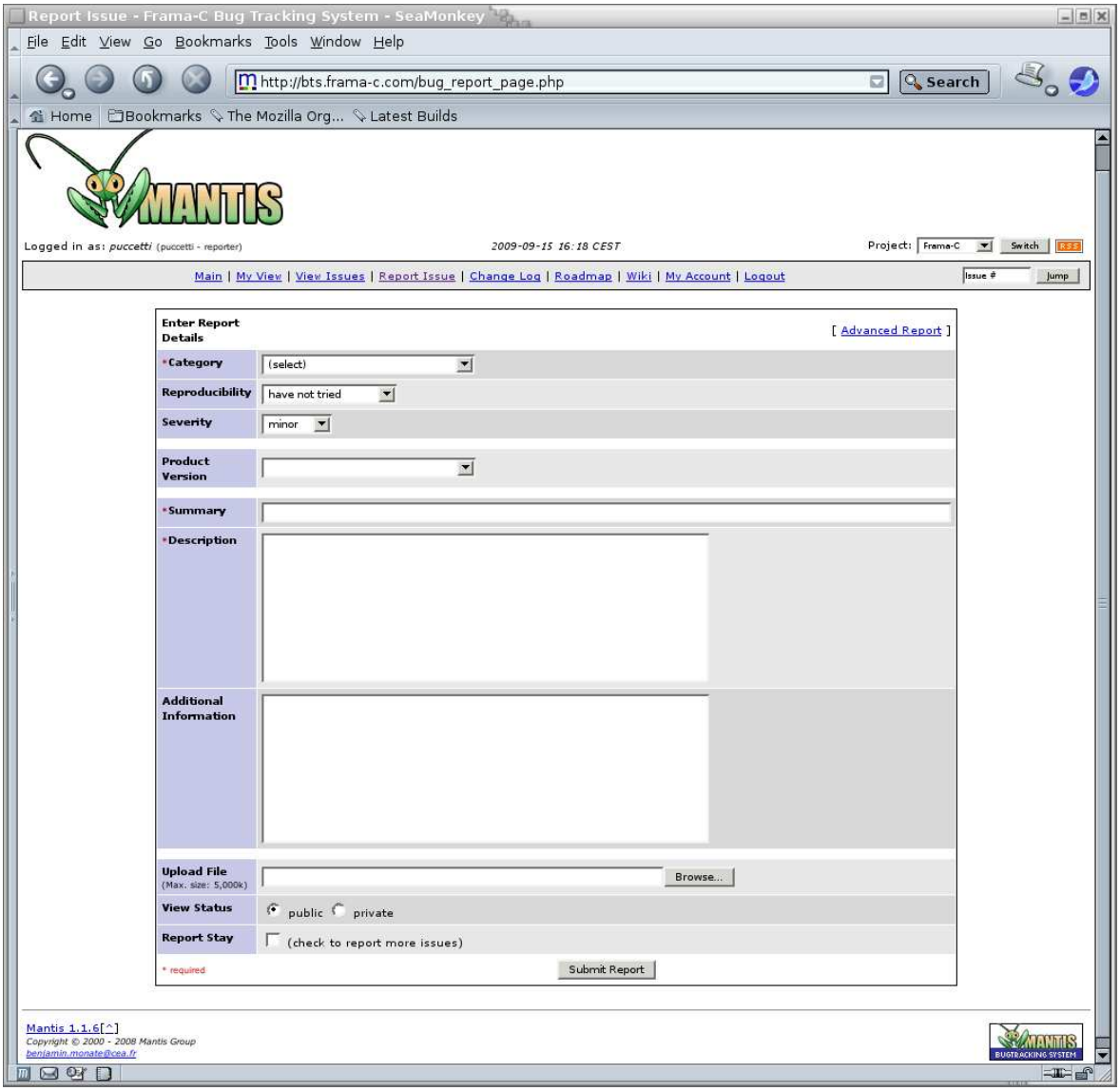

Figure 9.1: The BTS Bugs Reporting Page

Additional Information: any extra information that might help the maintainer.

Industrial: set it to `true` if you have a maintenance contract with the Frama-C development team.

Upload File: click on the `Browse` button to select a file for uploading. Typically, this is an archive that contains all files necessary for reproducing your problem. It can include C source files, shell scripts to run Frama-C with your options and environment, a Frama-C journal, *etc*. Please check the size of the archive in order to to keep it manageable: leave out any object code or executable files that can be easily rebuilt automatically (by a shell script for instance).

View Status: set it to `private` if your bug should not be visible by others users. Only yourself and the Frama-C developers will be able to see your bug report.

Report Stay: tick if this report shall remain open for further additions.

After submitting the report you will be notified by e-mail about its progress and enter interactive mode on the BTS if necessary.

Appendix A

Changes

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

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

Carbon-20101201

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

Boron-20100401

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

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

Beryllium-20090902

- First public release

Bibliography

[1] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language. Version 1.4 — Frama-C Beryllium implementation.)*, October 2009.

[2] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language (preliminary design V1.4)*, preliminary edition, October 2008.

[3] Pascal Cuoq and Virgile Prevosto. *Frama-C's value analysis plug-in*, October 2009. http://frama-c.cea.fr/download/value-analysis-Beryllium-20090902.pdf.

[4] Julien Signoles, Loïc Correnson, and Virgile Prevosto. *Frama-C Plug-in Development Guide*, September 2009. http://frama-c.cea.fr/download/plugin-developer-Beryllium-20090902.pdf.

List of Figures

8.1 Initial View 35
8.2 The Analysis Configuration Window 37

9.1 The BTS Bugs Reporting Page 40

Index

-absolute-valid-range, *29*
ACSL, 12, 13, 25, 26
-add-path, *22*
-allow-duplication, *26*
-annot, *26*

Batch version, 16
Bytecode, *16*

C pre-processor, 15
C99 ISO standard, 12
-collapse-call-cast, *26*
-collect-messages, *36*
-constfold, *26*
-continue-annot-error, *26*
-cpp-command, *25*
-cpp-extra-args, *25*

--datarootdir, *19*
-debug, *18*
-dynlink, *22*

--enable-external, *22*

-float-digits <n>, *18*
-force-rl-arg-eval, *26*
frama-c, 16
frama-c-gui, 16, 35
frama-c-gui.byte, 16, 35
frama-c.byte, 16
FRAMAC_LIB, *19*
FRAMAC_PLUGIN, *19*
FRAMAC_SHARE, *19*

GTK+, 15
GtkSourceView, 15

-h, *17*
-help, *17*
--help, *17*

Installation, 15
Interactive version, 16

Journal, *33*
-journal-disable, *33*
-journal-enable, *33*
-journal-name, *33*

-keep-comments, *18*
-keep-switch, *26*
-kernel-debug, *18*
-kernel-h, *17*
-kernel-help, *17*
-kernel-verbose, *18*

Lablgtk, 15
-lib-entry, *29*
--libdir, *19*
-load, *32*, 32, 36
-load-module, *22*, 37
-load-script, *22*, 33, 37

-machdep, *27*
-main, *29*

Native-compiled, 15, *16*

OCaml compiler, 15
OcamlGraph, 16
-ocode, *18*
Options, 16
-overflow, *29*

Plug-in
  Dynamic, *21*, 22, 23, 37
  External, *21*, 22
  Internal, *21*, 21
  Static, *21*, 22
-pp-annot, *25*
Pragma
  UNROLL, 27
-print, *18*, 26
-print-lib-path, *19*, 22
-print-plugin-path, *19*, 22
-print-share-path, *19*, 22
Project, *31*

-quiet, *18*

-safe-arrays, *30*, 32
-save, *32*, 32, 36
-semantic-const-fold, 17
-simplify-cfg, *27*

-then, *17*
-then-on, *17*
-time, *18*
-type-check, *27*

-ulevel, 17, 26, *27*, 32
-unsafe-arrays, *30*, 32
-unspecified-access, *30*

-val, 17
-verbose, *18*
-version, *18*, 39

--with-all-static, *21*
--with-no-plugin, *21*