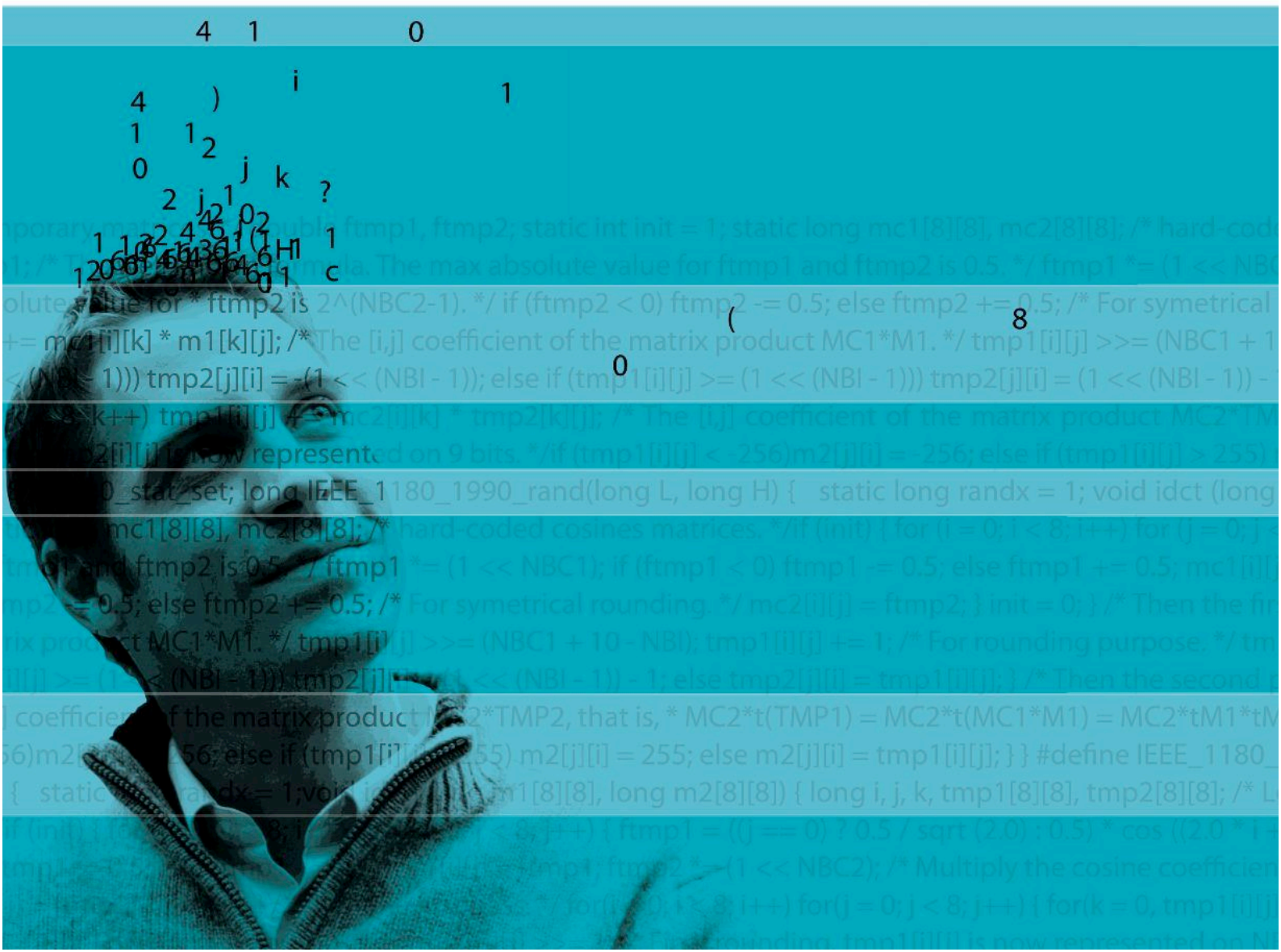
Software Analyzers

FRAMA-CLANG User Manual

version 0.0.7

for FRAMA-CLANG version 0.0.7

and FRAMA-C version 19.0 Potassium



Frama-Clang Plug-in User Manual

Release 0.0.7 for Frama-Clang 0.0.7

David R. Cok

CEA LIST
Software Reliability & Security Laboratory

Document printed 30 August 2019

Contents

Foreword

This is the user manual for the Frama-C plug-in Frama-Clang.[1] The contents of this document correspond to version 0.0.7 of the plug-in compatible with the 19.0 Potassium version of Frama-C [3, 4]. The development of the Frama-Clang plug-in is still ongoing. Features described by this document will certainly evolve in the future.

Acknowledgements

We gratefully thank the people who contributed to this document: David R. Cok, Virgile Prevosto, Armand Puccetti, Franck Védrine.

This document was written in the context of project VESSEDIA, which received funding from the European Union's 2020 Research and Innovation Program under grant agreement No. 731453.

[1] https://frama-c.com/frama-clang.html

Chapter 1

Introduction

FRAMA-C [3, 4] is a modular analysis framework for the C programming language that supports the ACSL specification language [1]. This manual documents the FRAMA-CLANG plug-in of FRAMA-C, version 0.0.7. The FRAMA-CLANG plug-in supports the ACSL++ extension of ACSL for C++ programs and specifications; it is built on the CLANG[1] compiler infrastructure and uses CLANG for parsing C++. The plug-in extends CLANG to parse ACSL++, translating source files containing C++ and ACSL++ into FRAMA-C's intermediate language for C and ACSL.

The FRAMA-CLANG plug-in intends to provide a full translation of C++ and ACSL++ into the FRAMA-C internal representation, and from there to allow C++ programs and ACSL++ specifications to be analyzed by other FRAMA-C plug-ins. *This is a work in progress.* The following sections describe the current status and limitations of the current implementation.

- The plug-in aims for the C++11 version of C++
- ACSL++ is described in the companion ACSL++ reference manual [2], also a part of the FRAMA-C release.
- The plug-in is compatible with version 6.0-8.0 of CLANG. This version of CLANG supports C++ versions through C++17 (cf. `https://clang.llvm.prg/cxx_status.html`). However, FRAMA-CLANG may not support all of the features of C++ within annotations.

[1] https://clang.llvm.org/

Chapter 2

Installation

Frama-Clang is currently still experimental and not part of regular Frama-C releases. It must be built from source and added to a Frama-C installation. The instructions for doing so are provided at https://frama-c.com/frama-clang.html.

Frama-Clang depends on two software packages:

- A current version of Frama-C itself. It is highly recommended to install Frama-C using `opam`, as described in the installation procedures for Frama-C (https://frama-c.com/download.html). Version 0.0.7 of Frama-Clang is compatible with version 19.0 Potassium of Frama-C.
- An installation of Clang, which is available as part of LLVM, which itself is available from http://releases.llvm.org. Version 0.0.7 of Frama-Clang is compatible with version 6.0-8.0 of clang.

Building and installing Frama-Clang has two effects:

- The Frama-Clang executable files are installed within the Frama-C installation. In particular, if Frama-C has been installed using `opam`, then the principal executable `framaCIRGen` will be installed in the `opam bin` directory. You must be sure that this directory is on your system `$PATH` (try `which framaCIRGen` after installation to be sure).
- Include files containing ACSL++ specifications of C++ library functions are copied to `$FRAMAC_SHARE/libc` and `$FRAMAC_SHARE/frama-clang/libc++`, where `$FRAMAC_SHARE` is the path given by the command `frama-c-config -print-share-path`.

These include files are replacements for the standard system include files. They should have the same definitions of C and C++ functions and classes, but with ACSL++ annotations giving their specifications.

The plugin can be built by hand from source using the following commands. Create a new directory to which you download and unpack the source distribution. Then `cd` into the source directory itself (one level down) and execute:

```
./configure
make
make install
```

By default, Frama-Clang will install its files under the same root directory as Frama-C itself. In particular, if Frama-C has been installed from `opam`, the installation will be done under `$(opam var prefix)` directory. Standard configure options for manipulating installation directories are available, notably `--prefix`.

Chapter 3

Running the plug-in

3.1 C++ files

Once installed the plugin is run automatically by FRAMA-C on any C++ files listed on the command-line. C++ files are identified by their filename suffixes. The default suffixes recognized as C++ are these: `.cpp`, `.C`, `.cxx`, `.ii`, `.ixx`, `.ipp`, `.i++`, `.inl`, `.h`, `.hh`

Currently this set of suffixes is compiled in the plugin (in file `frama_Clang_register.ml`) and can only be changed by recompiling and reinstalling the plugin.

3.2 Frama-clang executable

The plug-in operates by invoking the executable `framaCIRGen` (which must be on the system `$PATH`) on each file identified as C++, in turn. For each file it produces a temporary output file containing an equivalent C AST, which is then translated and passed on as input to FRAMA-C. This executable is a single-file-at-a-time command-line executable only. Various options control its behavior.

The file-system path identifying the executable is provided by the **-cxx-clang-command <cmd>** option and is `framaCIRGen` by default. The path may be absolute; if it is a relative path, it is found by searching the system `$PATH`.

The PARSING section of the output of `frama-c -kernel-h` lists some options for controlling the behavior described above. Also see the options listed by `frama-c -fclang-h` such as `-cxx-stdlib-path`, `-cxx-cstdlib-path`, `-cxx-nostdinc`, `-cxx-stdinc`.

3.3 Frama-clang options

The options controlling FRAMA-CLANG are of four sorts:

- options known to the FRAMA-C kernel
- options the FRAMA-CLANG plug-in has registered with the FRAMA-C kernel. These also are recognized by the `frama-c` command-line.
- options known to `framaCIRGen` directly (and not to `frama-c`), These must be included in the internal command that invokes `framaCIRGen` using the `-cpp-extra-args` option. These options are described in §4.

- CLANG options, which must also be supplied using the `-cpp-extra-args` option, and are passed through `framaCIRGen` to `clang`. See §4.

The options in the first two categories are processed by the `frama-c` kernel when listed on the `frama-c` command-line. The use of the `frama-c` command-line is described in the core `frama-c` user guide. There are many kernel options that affect all plugins and many options specific to FRAMA-CLANG. The command

```
frama-c -kernel-h
```

shows all kernel options; the command

```
frama-c -fclang-h
```

shows all `frama-clang`-specific options.

The most important of the options are these:

- `--help` or `-h` – introduction to FRAMA-C help
- `-kernel-h`, `-fclang-h` – help information about `frama-c`, the `frama-c` kernel and the `frama-clang` plug-in
- `-cpp-extra-args <string>` – the single string argument to this option is *prepended* to the command-line when `frama-clang` is invoked internally. It is particularly important for adding include directories (`-I`) and other options to be passed on to the clang compiler or the `framaCIRGen` executable. Multiple instances of this option have a cumulative effect, in order (rather than later instances replacing earlier ones).
- `-print` – prints out the input file seen by `frama-c`; when `frama-clang` is being used this is the input file after pre-processing and translation from C++ to C. Thus this output can be useful to see (and debug) the results of `frama-clang`'s transformations.
- `-kernel-warn-key=annot-error=<val>` sets the behavior of FRAMA-C, including FRAMA-CLANG, when a parsing error is encountered. The default value (set by the kernel) is `abort`, which terminates processing upon the first error; a more useful alternative is `active`, which reports errors but continues processing further annotations.
- `-machdep <arg>` – sets the target machine architecture, cf. §3.5
- `-kernel-msg-key <categories>` – sets the amount of informational messages according to different categories of messages. See `-kernel-msg-key help` for a list of informational categories.
- `-kernel-warn-key <categories>` – sets the amount and behavior of warnings. See `-kernel-warn-key help` for a list of warning categories.
- `-fclang-msg-key <categories>` – sets the amount of informational messages according to different categories of messages. See `-fclang-msg-key help` for a list of informational categories.
- `-fclang-warn-key <categories>` – sets the amount and behavior of warnings. See `-fclang-warn-key help` for a list of warning categories.
- `-fclang-verbose <n>` – sets the amount of information from the FRAMA-CLANG plug-in
- `-fclang-debug <n>` – sets the amount of debug information from the FRAMA-CLANG plug-in

- `-annot` – enables processing ACSL++ annotations (enabled by default)
- `-no-annot` – disables processing ACSL++ annotations

Note that the FRAMA-C option `-no-pp-annot` is ignored by FRAMA-CLANG. Preprocessing is always performed on the source input (unless annotations are ignored entirely using `-no-annot`).

3.4 Include directories

By default `framaCIRGen` is given the paths to the two directories containing the `frama-clang` and `frama-c` header files, which include ACSL++ specifications for the C++ library functions. The default paths (`$FRAMAC_SHARE/libc++` and `$FRAMAC_SHARE/libc` respectively) to these directories can be overriden by the `frama-clang` options `-cxx-c++stdlib-path` and `-cxx-cstdlib-path` options.

Users typically have additional header files for their own projects. These are supplied to the `frama-clang` preprocessor using the option `-cpp-extra-args`.

You can use `-fclang-cpp-extra-args` instead of `cpp-extra-args`; multiple such options also have a cumulative effect. The `frama-clang` option only affects the `frama-clang` plugin, whereas `-cpp-extra-args` may be seen by other plugins as well, if such plugins do their own preprocessing. Also note that the presence of any instance of `-fclang-cpp-extra-args` will cause uses of `-cpp-extra-args` to be ignored.

The system header files supplied by `frama-clang` does not include all C++ system files. Omissions should be reported to the `frama-c` team.

As an example, to perform `wp` checking of files `a.cpp` and `inc/a.h`, one might use the command-line

```
frama-c -cpp-extra-args="-Iinc" -wp a.cpp
```

3.5 32 and 64-bit targets

ACSL++ is for the most part machine-independent. There are some features of C++ that can be environment-dependent, such as the sizes of fundamental datatypes. Consequently, FRAMA-C has some options that allow the user to state what machine target is intended.

- The `-machdep` option to FRAMA-C. See the allowed values using the command
  `frama-c -machdep help`.
  For example, with a value of `x86_32`, `sizeof(long)` has a value of 4, whereas with the option `-machdep x86_64`, `sizeof(long)` has a value of 8.

3.6 Warnings, errors, and informational output

Output messages arise from multiple places: from the `frama-clang` plugin, from the `framaCIRGen` lexer and parser, from the CLANG parser, and from the FRAMA-C kernel (as well as from any other plugins that may be invoked, such as the `wp` plug-in). They are controlled by a number of options within the FRAMA-C kernel and each plugin. Remember that `clang` and `framaCIRGen` options must be put in the `-cpp-extra-args` option.

Output messages, including errors, are written to standard out, not to standard error.

3.6.1 Errors

Error messages are always output. The key question is whether processing stops or continues upon encountering an error. Continuing can result in a cascade of unhelpful error messages, but stopping immediately can hide errors that occur later in source files.

- `--stop-annot-error` is a `framaCIRGen` option that causes prompt termination on annotations errors (the `framaCIRGen` default is to continue); this does not respond to errors in C++ code
- `-kernel-warn-key=annot-error=abort` is a `frama-clang` plug-in option that will invoke `framaCIRGen` with `--stop-annot-error`. `error` and `error_once` (instead of `abort`) have the same effect; other values for the key will allow continuing after errors. The default is `abort`.

3.6.2 Warnings

Warning messages from `framaCIRGen` can be controlled with the `-warn` option of `framaCIRGen`.

- `-Werror` is a clang and `framaCIRGen` option that causes any parser warnings to be treated as errors
- `-w` is a clang and `framaCIRGen` option that causes any parser warnings to be ignored
- the `framaCIRGen` option `--no-warn` or `--warn=0` turns off `framaCIRGen` warning messages
- the `framaCIRGen` option `--warn=<n>`, with $n > 0$ turns on `framaCIRGen` warning messages; the higher the value n the more messages
- the `framaCIRGen` option `--warn` is the same as `--warn=1`

The CLANG *options are not currently integrated with the* `frama-c` *warning and error key system.*

3.6.3 Informational output

This section is not yet written

The clang informational output is not currently integrated with the `frama-c` *warning and error key system.*

Chapter 4

Running the Frama-Clang front-end standalone

In normal use within Frama-C, the `framaCIRGen` executable is invoked automatically. However, it can also be run standalone. In this mode it accepts command-line options and a single input file; it produces a C AST representing the translated C++, in a text format similar to Cabs.

The exit code from `framaCIRGen` is

- 0 if processing is successful, including if only warnings or informational messages are emitted
- 0 if there are some non-fatal errors but `--no-exit-code` is enabled (the default)
- 1 if there are some non-fatal errors but `--exit-code` is enabled, or if there are warnings and `-Werror` is enabled, but `-w` is not.
- 2 if there are fatal errors

Fatal errors are those resulting from misconfiguration of the system; non-fatal errors are the result of errors in the user input (e.g. parsing errors).

The `-Werror` option causes warnings to be treated as errors.

All output is sent to the standard output.[1]

4.1 `framaCIRGen` specific options

These options are specific to `framaCIRGen`.

- `-h` – print help information
- `-help` – print more help information
- `--version` – print version information

[1] Currently clang output goes to std err.

- `-o <file>` – specifies the name and location of the output file (that is, the file to contain the generated AST). The output path may be absolute or relative to the current working directory. *This option is required.*
- `-w` – suppress warnings (overrides `-Werror`)
- `-Werror` – treat warnings as errors
- `--info=<n>` – sets the level of informational messages to **n**; 0 is completely quiet; increasing values are more verbose.
  `--info` sets the level to 1
  `--no-info` sets the level to 0
  The `frama-c` option `-fclang-msg-key=parse` is equivalent to setting a value of 1.
- `--warn=<n>` – sets the level of parser warning messages to **n**; 0 is completely quiet; increasing values are more verbose.
  `--warn` sets the level to 1
  `--no-warn` sets the level to 0
  The `frama-c` option `-fclang-warn-key=parse` is equivalent to setting a value of 1.
- `--debug=<n>` – sets the level of parser debug messages to **n**; 0 is completely quiet; increasing values are more verbose
  `--debug` sets the level to 1
  `--no-debug` sets the level to 0
  The `frama-c` option `-fclang-debug=<n>` is equivalent to setting a value of **n**. In particular, a debug value of 1 shows the command-line that invokes `framaCIRGen`.
- `--stop-annot-error` – if set, then parsing stops on the first error; default is off
- `--exit-code` – if set, then the exit code of `framaCIRGen` is 1 if errors occur; this is not the default because then `frama-c` would terminate upon any error in `framaCIRGen`
- `--no-exit-code` – disables `--exit-code`, so that the exit code is always 0 for non-fatal errors.
- `--annot` – enables processing ACSL++ annotations (enabled by default)
- `--no-annot` – disables processing ACSL++ annotations

4.2 Clang options

Frama-Clang is built on the CLANG C++ parser. As such, the `framaCIRGen` executable accepts the clang compiler options and passes them on to clang. There are many of these. Many of these are irrelevant to `frama-clang` as they relate to code generation, whereas `frama-clang` only uses CLANG for parsing, name and type resolution, and producing the AST. You can see a list of options by running `framaCIRGen -help`

The most significant `clang` options are these:

- `-I <dir>` – adds a directory to the include file search path. Using absolute paths is recommended; relative paths are relative to the current working directory.
- `-w` – suppress clang warnings

- `-Werror` – treat warnings as errors

Although CLANG can process languages other than C++, C++ is the only one usable with FRAMA-CLANG.

4.3 Default command-line

The launching of `framaCIRGen` by FRAMA-C includes the following options by default. The `frama-c` option `-fclang-msg-key=clang` will show (among other information) the internal command-line being invoked.

- `-target <target>` with the target being set according to the `-machdep` and `-target` options given to FRAMA-C (cf. §3.5)
- `-D__FC_MACHDEP_86_32` – also set according to the chosen architecture
- `-std=c++11` – target C++11 features
- `-nostdinc` – use `frama-clang` and FRAMA-C system header files, and not the compiler's own header files
- `-I$FRAMAC_SHARE/frama-clangs/libc++ -I$FRAMAC_SHARE/libc` – include the FRAMA-CLANG and FRAMA-C header files, which contain system library definitions with ACSL++ annotations (the paths used are controlled by the `frama-c` options `-cxx-c++stdlib-path` and `-cxx-cstdlib-path`).
- `--annot` or `--no-annot` according to the `-annot` or `-no-annot` FRAMA-C kernel option
- `-stop-annot-error` if the corresponding option (`-fclang-warn-key=annot-error=abort`) is given to FRAMA-C
- options to set the level of info messages and warning messages, based on options on the `frama-c` command-line

Chapter 5

Known Limitations

The development of the Frama-Clang plug-in is still ongoing. Frama-Clang does not implement all of current C++ nor all of ACSL++ as defined in its language definition [2]. The most important such limitations are listed in this section.

These lists are not (nearly) complete

5.1 Implementation of C++

The following C++ features are not implemented in ACSL++.

- preprocessing is restricted within ACSL++ annotations (cf. §6)
- uses of typename
- uses of templates are not robust
- uses of typeid

5.2 Implementation of ACSL++

These ACSL++ features are not yet implemented

- Frama-Clang cannot process annotations that are separate from the source file
- ACSL++ specifications for standard C++ library functions are still quite limited
- ACSL++ definitions within template declarations
- ghost code is not yet implemented
- model declarations
- set comprehensions
- using (namespace) declarations (parsed but has no effect)
- pure functions (parsed but incompletely implemented)
- throws clause (parsed but not implemented in Frama-C)

- interaction of throws and noexcept
- parallel \let
- \count and \data are parsed but not yet implemented in Frama-C
- formal parameters that are references have pre and post states
- dynamic casting not yet implemented in Frama-C
- rounding mode and related builtin functions
- builtin types list and \set and related builtin functions
- \valid_function \allocable \freeable \fresh are not yet implemented by Frama-C
- extended quantifiers are not yet implemented by Frama-C
- global invariants are not yet implemented by Frama-C
- generalized invariants are not yet implemented by Frama-C
- assigns with both \from and = is not yet implemented

5.3 Other `frama-clang` limitations

- `-fclang-version` is not implemented
- parsing routines need work to improve robustness, to improve accuracy of locations, and to guard against leaking memory when parses fail
- the term/predicate parsing methods should be refactored to avoid deep call stacks
- resolve issues of tset representations
- cannot change the set of C++ suffixes
- frama-clang info/warn/error messages are not yet properly categorized and integrated with -fclang-log, -fclang-msg-key, fclang-warn-key. In particular, clang messages are completely independent of the Frama-C logging framework

Chapter 6

Preprocessing

This section describes the implementation of the C++ preprocessor for ACSL++ annotations. Recall that the C++ preprocessor replaces comments (including ACSL++ comments) by white space, without operations such as handling preprocessor directives. Accordingly, Frama-Clang must handle standard preprocessing within ACSL++ annotations itself.

As a refresher, the C/C++ preprocessor (CPP) (cf. https://gcc.gnu.org/onlinedocs/cpp/) conceptually implements the following operations on a C++ source file:

- The input is translated into a basic set of characters, including replacing trigraph sequences by their source character set equivalents
- Any backslash-whitespace-line-terminator sequence is removed and the line that it ends is combined with the following line, producing a sequence of logical lines.
- Comments are replaced by single spaces. This requires tokenizing the input to avoid recognizing comment markers within strings as indicating a comment. Note that this allows block comments to hide line terminations.
- The input text is divided into preprocessing tokens grouped in logical lines. Each preprocessor token becomes a compiler token (except where ## concatenation occurs). However, ACSL/ACSL++ tokens are slightly different, as described below.
- The source text is transformed according to any preprocessing directives contained within it. Each preprocessing directive must be contained within one logical line. The result has no preprocessing directives remaining.
- Adjacent string literals (separated only by white-space or line-endings) are concatenated into a single string literal.

The result is a sequence of preprocessing tokens that is passed on to the remaining compiler phases.

6.1 Frama-Clang preprocessor implementation

The Frama-Clang implementation operates as follows, on each ACSL++ annotation comment in turn:

- A simplified custom lexer converts the text into preprocessor tokens, without doing macro substitution, to find instances of forbidden preprocessor directives. If possible and reasonable, these are elided from the input text and processing continues.
- The text is then submitted to `clang` to obtain the complete sequence of preprocessor tokens, now with full preprocessing (except for adjacent string concatenation).
- `frama-clang` transforms the clang preprocessor tokens into ACSL++ tokens, which are then passed on to the ACSL++ parser to produce the desired AST.

6.2 Trigraphs

Trigraphs are defined for C++ but will currently be removed in C++17. Since trigraph processing by clang occurs before any recognition of comments, trigraphs in ACSL++ annotations are translated, if enabled in CLANG. As they will be removed from C++, they are not recommended for use in ACSL++ annotations. Preprocessing of trigraphs is enabled by default.

6.3 Digraphs

Digraphs are alternate spellings of preprocessor tokens, in particular, of punctuation character sequences. Digraphs in ACSL++ annotations are translated just as they are in C++ (by CLANG). Using digraphs is not recommended.

6.4 Preprocessor tokens

Preprocessor tokens (per CPP) belong to one of several categories: identifiers, literals (including numeric, character and string literals), header names, operators, punctuation, and single other characters. White space (space, tab) serves only to separate tokens; it is not needed between tokens whose concatenation is not a single token. Line terminators also separate tokens and also delineate certain features: preprocessing directives and string literals do not extend over more than one (logical) line.

Dollar signs are also allowed as non-digit identifier characters if the clang option `-fdollars-in-identifiers` is enabled, which it is by default.
Enable with `-fdollars-in-identifiers` ;
disable with `-fno-dollars-in-identifiers` .

Numeric literals are more general than a C++ or ACSL number. Nevertheless, aside from token concatenation, each preprocessing token becomes a compiler token, which then may be an illegal compiler token.

The token definitions imply that arbitrary text can always be broken into legitimate preprocessor tokens, with the exception of a few characters and of badly formed unicode sequences.

Note that not all preprocessor tokens are valid C/C++ parser tokens. Tokens in the other category have no meaning to C/C++ and the `number` category allows many sequences that are not legal C/C++ numeric tokens. These tokens will generally provoke compiler errors. For example in C/C++, 0..2 is one token and is not interpreted as two consecutive numeric tokens.

ACSL and ACSL++ have slightly different tokens than the preprocessor, so the preprocessor tokens need to be re-tokenized in some cases:

- The `@` token is a whitespace character in the interior of a ACSL/ACSL++ annotation.
- There are some ACSL/ACSL++ multi-character punctuator tokens that are not single preprocessor tokens:
  - all ACSL/ACSL++ keywords that begin with a backslash, such as `\result`.
  - `==>` (logical implies)
  - `-->` (bit-wise implies)
  - `<==>` (logical equality)
  - `<-->` (bit-wise equality)
  - `^^` (logical inequality)
  - `^*` (list repetition)
  - `[|` and `|]` (list creation)

  These ACSL/ACSL++ tokens need to be assembled from multiple CPP tokens (and those CPP tokens must not be separated by white space)
- A CPP numeric token that contains .. will not be a legal C/C++ number, but may be a sequence of legal ACSL/ACSL++ tokens with the .. representing the range operator. For example, the single CPP token `0..last` is retokenized for ACSL/ACSL++ as if it were written `0 .. last` .
- ACSL/ACSL++ allows certain built-in non-ASCII symbols. An example is ∀ (unicode 0x2200) to designate a universal quantifier, which is an alternative form of `\forall`. A complete list of such tokens is given in the ACSL/ACSL++ language definition.

6.5 Preprocessor directives

A preprocessing directive consists of a single logical line (after the previous preprocessing phases have been completed) that begins with optional white space, the `#` character, additional optional white space, and a preprocessor directive identifier. The preprocessing language contains a fixed set of preprocessing directive identifiers:

- `define`, `undef`
- `if`, `ifdef`, `ifndef`, `elif`, `else`, `endif`
- `warning`, `error`
- `include`
- `line`
- `pragma`

In addition, identifiers that have been defined (by a `#define` directive) as macros are expanded according to the macro expansion rules (not described here).

Because ACSL/ACSL++ annotations are contained in C/C++ comments, any directives contained in the annotation are not seen when the source file is processed simply as a C/C++ source file. However, the effect of some directives lasts until the end of the source file. Accordingly, ACSL++ imposes constraints on the directives that may be present within annotations:

- `define` and `undef` are not permitted in an annotation. (If they were to be allowed, their scope would have to extend only to the end of the annotation, which could be confusing to readers.)
- macros occurring in an annotation but defined by `define` statements prior to the annotation are expanded according to the normal rules, including concatenation by `##` operators. The context of macro definitions corresponds to the textual location of the annotation, as would be the case if the annotation were not embedded in a comment.
- `if`, `ifdef`, `ifndef`, `elif`, `else`, `endif` are permitted but must be completely nested within the annotation in which they appear (an `endif` and its corresponding `if`, `ifdef`, `ifndef`, or `elif` must both be in the same annotation comment.)
- `warning` and `error` are permitted
- `include` is permitted, but will cause errors if it contains, as is almost always the case, other disallowed directives
- `line` is not permitted
- `pragma` and the `_Pragma` operator are not permitted
- stringizing (`#`) and string concatenation (`##`) operators are permitted
- the `defined` operator is permitted
- the standard predefined macro names are permitted: `__cpluscplus` (in C++ compilers), `__DATE__`, `__TIME__`, `__FILE__`, `__LINE__`, `__STDC_HOSTED__`

Chapter 7

Grammar and parser for ACSL++

This section summarizes some of the technical implementation considerations in writing a parser for ACSL++ within Frama-Clang. This material may be useful for developers and maintainers of Frama-Clang; it is not needed by users of Frama-Clang.

Recall that Frama-Clang uses clang to parse C++ and a custom parser to parse ACSL++ annotations, jointly producing a representation of the C++ and ACSL++ source input in the Frama-C intermediate language. The first version of the ACSL++ custom parser, written during the STANCE project, used a hand-written bison-like parser, but with function pointers to record state and actions rather than using a tool-generated table to drive the parsing. This design proved to be too brittle and difficult to efficiently evolve as new features were added to ACSL++. Consequently during the VESSEDIA project, the scanner and parser were completely rewritten, largely retaining the previous design's connections to clang, token definitions, name lookup and type resolution, and AST generation.

The new parser uses a recursive descent design in which the names of functions doing the parsing can match the names of non-terminals in the grammar. Consequently the implementation of the parser is much more readable, human checkable, and modifiable as the ACSL++ language evolves. The drawback of this design is that ACSL++ is not LL(1); it is not even LL(k) for fixed k. Thus some amount of lookahead and backtracking is required. The bulleted paragraphs below describe the problematic aspects of ACSL++ and how they are addressed.

The principal goal of an LL(k) formulation of a grammar is to be able to predict which grammar production is being seen in the input stream from a small amount of look-ahead. Most ACSL++ constructs start with a unique keyword (e.g., clauses begin with `requires`, `ensures` etc.) which serves this purpopse. But the constructs inherited from C++ pose some challenges.

- **Left recursion**. Expression grammars are typically left recursive, which is problematic for recursive descent parsers. However, it is well-known how to factor out left recursion. The precedence order of operators is largely hard-coded into the grammar implementation; the usual left recursion poses no particular challenge.

- **terms vs. predicates**. ACSL++ distinguishes terms and predicates, following the distinction between propositional and predicate logic. However, terms and predicates have very similar grammars. Furthermore, ACSL++ allows boolean-value terms to be implicitly converted to predicates and allows predicates to be used within terms (such as for the conditional sub-expression in a ternary expression). This makes it not possible to distinguish terms and predicates in top-down parsing. However, Frama-C has different

AST nodes for the two, so it would require a very significant refactoring of Frama-C and all its plugins to unify terms and predicates (as other specification languages have done). Note that this problem is a challenge for any parser design. The previous and current parser designs adopted the same solution: maintain two parallel parses of expressions — one as a term and one as a predicate. Error messages are emitted only when both parses fail or when a particular grammar production calls for a particular type of AST (term or predicate) and that one is not available.

- **terms vs. tsets**. Similarly, the ACSL++ language definition defines tsets (sets of locations) with grammar productions separately from terms. However, the grammars for the two are very similar. ACSL++ is much easier to parse and to implement if tsets are seen as terms with a specific type, namely sets. Many operations on a data type are also simply element-by-element operations on sets of such data types. Also, errors found during type-checking can be associated with more readable error messages than those found during parsing.

- **cast vs. parenthesized expression** To determine whether an input like `(T)-x` is (a) the difference of the parenthesized expression `(T)` and `x` or (b) a cast of `-x` to the type `T`, one must know whether `T` is a variable or type. This is a classic problem in parsing C++; it requires that identifiers be known to be either type names or variable names in the scanner. In addition, `T` here can be an arbitrary type expression. For example, in C++, a type expression can contain pointer operators that can look, at least initially like multiplications and they can contain template instantiations that look initially like comparisons. Frama-Clang handles this situation by allowing a backtrackable parse. When a left parenthesis is parsed in an expression context, the parser proceeds by attempting a parse of a cast expression. If the contents of the parenthesis pair is successfully parsed as a type expression, it is assumed to be a cast expression.

  If such a parse fails, no error messages are emitted; rather the parse is rewound and proceeds again assuming the token sequence to be a parenthesized expression.

- **set comprehension**. The syntax of the set comprehension expression follows traditional mathematics: `{` *expr* `|` *binders* `;` *predicate* `}`. This structure poses two difficulties for parsers. First, the expression *expr* may contain bitwise-or operators, so it is not known to the parser whether an occurence of `|` is the beginning of the binders or is just a bitwise-or operator. Second, the expression will use the variables declared in the binders section. However, the binders are not seen until after the expression is scanned. Note that these problems are not unique to a recursive descent design; they would challenge a LR parser just as much. *This particular feature is not yet implemented in* Frama-Clang*, nor in Frama-C and so is not yet resolved in the parser implementation.*

- **labeled expressions**. ACSL++ allows expressions to have labels, designated by a `id :` prefix. So the parser cannot know whether an initial `id` is a variable or just a label until the colon is parsed. Thus this situation requires a lookahead of 2 tokens.

  Ambiguity arises with the use of a colon for the else part of a conditional expression. So in an expression such as `a ? b ? c : d : e : f`, it is ambiguous whether `c` or `d` or `e` is a label. Parenthesizing must be used to solve this problem. `frama-clang` presumes that if the *then* part of a conditional expression is being parsed, a following colon is always the colon introducing the *else* part. That is, the binding to a conditional expression has tighter precedence than to a naming expression.

Appendix A

Changes

This chapter summarizes the changes in this documentation between each Frama-Clang release, with the most recent releases first.

Version 0.0.7

- First release of this manual.

Bibliography

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

[2] David R. Cok. *ACSL++: ANSI/ISO C++ Specification Language.*

[3] Loïc Correnson, Pascal Cuoq, Florent Kirchner, André Maroneze, Virgile Prevosto, Armand Puccetti, Julien Signoles, and Boris Yakobowski. *Frama-C User Manual.* http://frama-c.cea.fr/download/user-manual.pdf.

[4] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A Software Analysis Perspective. *Formal Aspects of Computing*, pages 1–37, jan 2015.