







# FRAMA-C's E-ACSL Plug-in

Release 0.1+dev compatible with Fluorine-20130501

Julien Signoles

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

©2013 CEA LIST

This work has been supported by the 'Hi-Lite' FUI project (FUI AAP 9).



# Contents

<b>Foreword</b>	<b>7</b>
<b>1 Introduction</b>	<b>9</b>
<b>2 What the Plug-in Provides</b>	<b>11</b>
2.1 Simple Example . . . . .	11
2.1.1 Running E-ACSL . . . . .	11
2.1.2 Executing the generated code . . . . .	13
2.2 Execution Environment of the Generated Code . . . . .	13
2.3 Potential Runtime Errors in the Generated Code . . . . .	13
2.4 Handling Incomplete Program . . . . .	13
2.5 Combining E-ACSL with Others Plug-ins . . . . .	14
2.6 Customizing the Plug-in . . . . .	14
2.7 Verbosing Policy . . . . .	14
<b>3 Known Limitations</b>	<b>15</b>
3.1 Undefined Value . . . . .	15
3.2 Function without Code . . . . .	15
3.3 Recursive Function . . . . .	15
3.4 Variadic Function . . . . .	15
3.5 Function Pointer . . . . .	15
<b>Bibliography</b>	<b>17</b>
<b>Index</b>	<b>19</b>



# Foreword

This is the user manual of the FRAMA-C plug-in E-ACSL<sup>1</sup>. The content of this document corresponds to its version 0.1+dev (May 17, 2013) compatible with the version Fluorine-20130501 of FRAMA-C [3, 5]. However the development of the E-ACSL plug-in is still ongoing: features described here may still evolve in the future.

---

<sup>1</sup><http://frama-c.com/eacsl>





## Chapter 1

# Introduction

FRAMA-C [3, 5] is a modular analysis framework for the C language which supports the ACSL specification language [1]. This manual documents the E-ACSL plug-in of FRAMA-C. This plug-in automatically translates an annotated C code into a C code which fails at runtime if an annotation is violated. If no annotation is violated, the behavior of the new program is exactly the same than the one of the original program.

Such a translation brings several benefits. First it allows the user to monitor a C code, in particular to perform what is usually called runtime assertion checking [2]<sup>1</sup>. That is the primary goal of E-ACSL. Second it allows to combine FRAMA-C and its existing analyzers, with other analyzer tools for C, even those who do not understand the ACSL specification language. Third, the possibility to detect invalid annotations during a concrete execution may be very helpful while writing a correct specification of a given program, *e.g.* for later program proving. Finally, an executable specification makes it possible to check runtime assertions that cannot be verified statically and to establish a link between monitoring tools and static analysis tools like VALUE [6] or WP [4].

Annotations must be written in the E-ACSL specification language [9, 7] which is a subset of ACSL. This plug-in is still in a preliminary state: some parts of E-ACSL are not yet supported. Which E-ACSL annotations are currently handled by the E-ACSL plug-in is documented in a separated document [10].

This manual does ***not*** explain how to install the plug-in. Please have a look at file `INSTALL` of the E-ACSL tarball for this purpose.

---

<sup>1</sup>In our context, “runtime annotation checking” would be a better more-general expression.



## Chapter 2

# What the Plug-in Provides

This chapter is the core of this manual and describes how to use the plug-in. First, Section 2.1 shows how to run the plug-in on a trivial example and how to execute the generated code with a standard C compiler to detect invalid annotations at runtime. Then, Section 2.2 provides additional details on the execution of the generated code. Section 2.3 explains how the plug-in handles potential runtime errors in the generated code. Next, Section 2.4 focus on how to deal with incomplete programs, *i.e.* in which some functions have no body or in which there are no main function. After, Section 2.5 explains how to combine the plug-in with other plug-ins of FRAMA-C. Finally, Section 2.6 introduces how to customize the plug-in, while Section 2.7 details the verbosing policy of the plug-in.

### 2.1 Simple Example

---

This Section is a mini-tutorial which explains from scratch how to detect at runtime that an E-ACSL annotation is violated thanks to the use of the plug-in.

#### 2.1.1 Running E-ACSL

Consider the following simple program in which the first assertion is valid while the second one is not.

File **first.i**

```
int main(void) {
    int x = 0;
    /*@ assert x == 0; */
    /*@ assert x == 1; */
    return 0;
}
```

Running E-ACSL on this file just consists in adding the option `-e-acsl` to the FRAMA-C command line:

```
$ frama-c -e-acsl first.i
[kernel] preprocessing with <...> share/frama-c/e-acsl/e_acsl_gmp_types.h
[kernel] preprocessing with <...> share/frama-c/e-acsl/e_acsl_gmp.h
[kernel] preprocessing with <...> share/frama-c/e-acsl/e_acsl.h
[kernel] preprocessing with <...> share/frama-c/e-acsl/memory_model/e_acsl_mmodel_api.h
[kernel] preprocessing with <...> share/frama-c/e-acsl/memory_model/e_acsl_bittree.h
[kernel] preprocessing with <...> share/frama-c/e-acsl/memory_model/e_acsl_mmodel.h
[e-acsl] beginning translation.
[e-acsl] translation done in project "e-acsl".
```

Even if `first.i` is already preprocessed, E-ACSL first asks the FRAMA-C kernel to preprocess and link with `first.i` several files which forms the so-called E-ACSL library. Its usefulness will be explain later, mainly in Section 2.2.

Then E-ACSL takes the annotated C code as input and translates it into a new FRAMA-C project named `e-acsl`<sup>1</sup>. By default, the option `-e-acsl` does nothing more. It is however possible to have a look at the generated code on the FRAMA-C GUI. It is also possible through the command line thanks to the kernel options `-then-on` and `-print`:

```
$ frama-c -e-acsl first.i -then-on e-acsl -print
[kernel] preprocessing with <...> share/frama-c/e-acsl/e_acsl_gmp_types.h
[kernel] preprocessing with <...> share/frama-c/e-acsl/e_acsl_gmp.h
[kernel] preprocessing with <...> share/frama-c/e-acsl/e_acsl.h
[kernel] preprocessing with <...> share/frama-c/e-acsl/memory_model/e_acsl_mmodel_api.h
[kernel] preprocessing with <...> share/frama-c/e-acsl/memory_model/e_acsl_bittree.h
[kernel] preprocessing with <...> share/frama-c/e-acsl/memory_model/e_acsl_mmodel.h
[e-acsl] beginning translation.
[e-acsl] translation done in project "e-acsl".
/* Generated by Frama-C */
struct __anonstruct__mpz_struct_1 {
    int _mp_alloc ;
    int _mp_size ;
    unsigned long *_mp_d ;
};
typedef struct __anonstruct__mpz_struct_1 __mpz_struct;
typedef __mpz_struct ( __attribute__((__FC_BUILTIN__)) mpz_t)[1];
typedef unsigned int size_t;
/*@
model __mpz_struct { integer n };
*/
/*@ requires predicate != 0;
    assigns \nothing; */
extern __attribute__((__FC_BUILTIN__)) void e_acsl_assert(int predicate,
                                                         char *kind,
                                                         char *fct,
                                                         char *pred_txt,
                                                         int line);

int __fc_random_counter __attribute__((__unused__));
unsigned long const __fc_rand_max = (unsigned long)32767;
/*@ ghost extern int __fc_heap_status; */

/*@
axiomatic
dynamic_allocation {
    predicate is_allocable{L}(size_t n)
        reads __fc_heap_status;

}
*/
extern __attribute__((__FC_BUILTIN__)) void __clean(void);

extern size_t __memory_size;

/*@
predicate diffSize{L1, L2}(integer i) =
    \at(__memory_size, L1) - \at(__memory_size, L2) == i;
*/
int main(void)
{
    int __retres;
    int x;
    x = 0;
    /*@ assert x == 0; */
    e_acsl_assert(x == 0, (char *)"Assertion", (char *)"main", (char *)"x == 0", 3);
    /*@ assert x == 1; */
    e_acsl_assert(x == 1, (char *)"Assertion", (char *)"main", (char *)"x == 1", 4);
}
```

<sup>1</sup>The notion of *project* is explained in Section 8.1 of the FRAMA-C user manual [3].

```

    __retres = 0;
    return __retres;
}

```

As you can see, the generated code contains additional type declarations, constant declarations and global ACSL annotations that are not in the initial file `first.i`. That is a part of the included E-ACSL library. You can safely ignore it right now. The translated `main` function of `first.i` is displayed at the end. Two lines have been added. The first one is just after the first E-ACSL annotation, while the second one is just after the second one.

```

/*@ assert x == 0; */
e_acsl_assert(x == 0, (char *) "Assertion", (char *) "main", (char *) "x == 0", 3);
/*@ assert x == 1; */
e_acsl_assert(x == 1, (char *) "Assertion", (char *) "main", (char *) "x == 1", 4);

```

They are function calls to `e_acsl_assert` which is defined in the E-ACSL library. Each call performs the dynamic verification that the corresponding assertion is valid. More precisely, it checks that its first argument (here `x == 0` or `x == 1` corresponding to the annotations) is not equal to 0 and fails otherwise. The extra arguments are only used to display nice user feedback as shown later in Section 2.2.

### 2.1.2 Executing the generated code

By using the option `-ocode` of FRAMA-C, we can redirect the generated code into a C file as follows.

```

$ frama-c -e-acsl first.i -then-on e-acsl -print -ocode monitored_first.i

$ gcc -o monitored_first frama-c -print-share-path/e-acsl/e_acsl.c
monitored_first.i
$ ./monitored_first
Assertion failed at line 4 of function main.
The failing predicate is:
x == 1.

```

## 2.2 Execution Environment of the Generated Code

---

- takes care of architecture (32, 64 bits)
- memory model (linking issue) [8]
- GMP (linking issue)
- customizing `e_acsl_assert`

## 2.3 Potential Runtime Errors in the Generated Code

---

- runtime error in annotations

## 2.4 Handling Incomplete Program

---

- function without code
- program without main

## 2.5 Combining E-ACSL with Others Plug-ins

---

- -e-acsl-valid
- -e-acsl-prepare

## 2.6 Customizing the Plug-in

---

- -e-acsl-project
- -e-acsl-check
- -eacsl-share

## 2.7 Verbosing Policy

---

- level
- category

## Chapter 3

# Known Limitations

The development of the E-ACSL plug-in is still ongoing. First, the whole E-ACSL reference manual [9] is not yet supported. Which annotations are already translated into C code and which are not is defined in a separated document [10]. Second, even if we do our best, bugs may exist. If you find a new one, please report it on the bug tracking system (see Chapter 10 of the FRAMA-C User Manual [3]). Third, there are some additional known limitations, which could be annoying for the user in some cases, but are hard to lift. Thus they could be there for a while. Lifting them could be part of commercial supports<sup>1</sup>.

### 3.1 Undefined Value

---

- use of undefined values are not tracked in annotations
- missing guards for `\offset`, `\block_length` and `texbase_addr (offset(p))` must ensures that `p` is valid)

### 3.2 Function without Code

---

### 3.3 Recursive Function

---

### 3.4 Variadic Function

---

- Not yet duplicated

### 3.5 Function Pointer

---

---

<sup>1</sup>Contact us or read <http://frama-c.com/support.html> for additional details.





# Bibliography

- [1] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language. Version 1.7*, April 2013.
- [2] Lori A. Clarke and David S. Rosenblum. A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Software Engineering Notes*, 31(3):25–37, 2006.
- [3] Loïc Correnson, Pascal Cuoq, Florent Kirchner, Armand Puccetti, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. *Frama-C User Manual*, May April. <http://frama-c.cea.fr/download/user-manual.pdf>.
- [4] Loïc Correnson, Zaynah Dargaye, and Anne Pacalet. *Frama-C's WP plug-in*, April 2013. <http://frama-c.com/download/frama-c-wp-manual.pdf>.
- [5] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C, A software Analysis Perspective. In *Software Engineering and Formal Methods (SEFM)*, October 2012.
- [6] Pascal Cuoq, Boris Yakobowski, and Virgile Prevosto. *Frama-C's value analysis plug-in*, April 2013. <http://frama-c.cea.fr/download/value-analysis.pdf>.
- [7] Mickaël Delahaye, Nikolai Kosmatov, and Julien Signoles. Common specification language for static and dynamic analysis of C programs. In *the 28th Annual ACM Symposium on Applied Computing (SAC)*, pages 1230–1235. ACM, March 2013.
- [8] Nikolai Kosmatov, Guillaume Petiot, and Julien Signoles. Optimized Memory Monitoring for Runtime Assertion Checking of C Programs. Submitted for publication.
- [9] Julien Signoles. *E-ACSL: Executable ANSI/ISO C Specification Language. Version 1.7*, May 2013. URL: <http://frama-c.com/download/e-acsl/e-acsl.pdf>.
- [10] Julien Signoles. *E-ACSL Version 1.7. Implementation in Frama-C Plug-in E-ACSL version 0.2*, May 2013. URL: <http://frama-c.com/download/e-acsl/e-acsl-implementation.pdf>.



# Index

`-e-acsl`, [11](#), [12](#)  
`e_acsl_assert`, [13](#)

## Function

  Pointer, [15](#)  
  Recursive, [15](#)  
  Variadic, [15](#)  
  Without code, [15](#)

Installation, [9](#)

`-ocode`, [13](#)

`-print`, [12](#)

`-then-on`, [12](#)

Undefined value, [15](#)