frama C

Software Analyzers

E-ACSL User Manual

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

Foreword 7

1 Introduction 9

2 What the Plug-in Provides 11

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 . . . 14
2.2.1 Runtime Errors in Annotations . . . 14
2.2.2 Architecture Dependent Annotations . . . 14
2.2.3 Integers . . . 15
2.2.4 Memory-related Annotations . . . 16
2.2.5 Execution Behavior Environment . . . 17
2.3 Incomplete Program . . . 18
2.3.1 Program without Main . . . 18
2.3.2 Function without Code . . . 19
2.4 Combining E-ACSL with Others Plug-ins . . . 19
2.5 Customization . . . 19
2.6 Verbosing Policy . . . 19
2.6.1 Verbosing Level . . . 20
2.6.2 Message Categories . . . 20

3 Known Limitations 21

3.1 Undefined Value . . . 21
3.2 Program without Main . . . 21
3.3 Function without Code . . . 21
3.4 Recursive Function . . . 21
3.5 Variadic Function . . . 21
3.6 Function Pointer . . . 21

Bibliography **23**

Index **25**

Foreword

This is the user manual of the Frama-C plug-in E-ACSL[1]. The content of this document corresponds to its version 0.1+dev (May 22, 2013) compatible with the version Fluorine-20130501 of Frama-C [4, 6]. However the development of the E-ACSL plug-in is still ongoing: features described here may still evolve in the future.

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

Chapter 1

Introduction

Frama-C [4, 6] 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 another one 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" [3][1]. 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 like PathCrawler [2], 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 at runtime assertions that cannot be verified statically and thus to establish a link between monitoring tools and static analysis tools like Value [7] or Wp [5].

Annotations must be written in the E-ACSL specification language [10, 8] 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 [11].

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.

[1] 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. Next, Section 2.3 focuses 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 2.4 explains how to combine the plug-in with others Frama-C plug-ins. Finally, Section 2.5 introduces how to customize the plug-in, while Section 2.6 details the verbosing policy of the plug-in.

2.1 Simple Example

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

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.h
[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/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 to link against `first.i` several files which forms the 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`[1]. By default, the option `-e-acsl` does nothing more. It is however possible to have a look at the generated code in the Frama-C GUI. It is also possible through the command line thanks to the kernel options `-then-on` and `-print` which respectively switches to another project and pretty prints the C code [4]:

```
$ frama-c -e-acsl first.i -then-on e-acsl -print
[kernel] preprocessing with <...> share/frama-c/e-acsl/e_acsl.h
[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/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 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);
  __retres = 0;
```

[1] The notion of *project* is explained in Section 8.1 of the Frama-C user manual [4].

```
  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`) is not equal to 0 and fails otherwise. The extra arguments are only used to display nice user feedback as shown later in Section 2.1.2.

2.1.2 Executing the generated code

By using the option `-ocode` of Frama-C [4], 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
```

Then it may be executed by a standard C compiler like Gcc in the following way.

```
$ gcc -o monitored_first `frama-c -print-share-path`/e-acsl/e_acsl.c monitored_first.i
<file_path>/e_acsl.h:41:3: warning: 'FC_BUILTIN' attribute directive ignored [-Wattributes]
monitored_first.i:8:1: warning: '__FC_BUILTIN__' attribute directive ignored [-Wattributes]
monitored_first.i:19:60: warning: '__FC_BUILTIN__' attribute directive ignored [-Wattributes]
```

You may notice that the generated file `monitored_first.i` is linked against the file `` `frama-c -print-share-path`/e-acsl/e_acsl.c ``. This last file is part of the E-ACSL library installed with the plug-in. It contains an implementation of the function `e_acsl_assert`, which is required to generate an executable binary from an E-ACSL-generated code.

The warnings can be safely ignored. They refer to an attribute `FC_BUILTIN` internally used by Frama-C. You can also add the option `-Wno-attributes` to Gcc if you do not want to be polluted by these warnings.

Finally you can execute the generated binary.

```
$ ./monitored_first
Assertion failed at line 4 in function main.
The failing predicate is:
x == 1.
$ echo $?
1
```

This execution stops with an exit code of 1 and an error message indicated that the invalid E-ACSL annotation has been violated. There is no output for the valid E-ACSL annotation. So, thanks to the plug-in, we detect that the second assertion in the initial program is wrong, while the first one is correct for this execution.

2.2 Execution Environment of the Generated Code

The environment in which the code is executed is not necesseraly the same than the one assumed by Frama-C. You must take care of that when running the E-ACSL plug-in and when compiling the generated code with Gcc. Also, the plug-in offers you few possibilities of customization.

2.2.1 Runtime Errors in Annotations

The major difference between ACSL [1] and E-ACSL [10] specification languages is that the logic is total in the former language while it is partial in the latter one: the semantics of a term denoting a C expression e is undefined if e leads to a runtime error and, consequently, the semantics of any term t (resp. predicate p) containing such an expression e is undefined as soon as e has to be evaluated in order to evaluate t (resp. p). The E-ACSL Reference Manual also states that *"it is the responsability of the tools which interprets* E-ACSL *to ensure that an undefined term is never evaluated"* [10].

Accordingly, the E-ACSL plug-in prevents undefined term to be evaluated. If it should be because an annotation contains such a term, it will report a proper error at runtime instead.

Consider for instance the following annotated program.

File **rte.i**

```
/*@ behavior yes:
      assumes x % y == 0;
      ensures \result == 1;
    behavior no:
      assumes x % y != 0;
      ensures \result == 0; */
int divide(int x, int y) {
  return x % y == 0;
}

int main(void) {
  divide(2, 0);
  return 0;
}
```

The terms and predicates containing the modulo '%' in the clause `assumes` are undefined in the context of the `main`'s function call since the second argument is equal to 0.

However we can generate an instrumented code and compile it through the following command lines:

```
$ frama-c -e-acsl rte.i -then-on e-acsl -print -ocode monitored_rte.i
$ gcc -o rte `frama-c -print-share-path`/e-acsl/e_acsl.c monitored_rte.i
```

Now, when you execute it, you get the following output which explains that your function contract is invalid because it contains a division by zero.

```
$ ./rte
RTE failed at line 5 in function divide.
The failing predicate is:
division_by_zero: y != 0.
```

2.2.2 Architecture Dependent Annotations

In many cases, the execution of a C program depends on the underlying machine architecture which it is executed on. Also the program must be compiled in the very same architecture (or cross-compiled) for the compiler being able to generate a correct binary.

FRAMA-C mades assumptions about this machine when analyzed such a file. By default, it assumes a x86 32 bits platform, but it may be changed thanks to the option `-machdep` [4]. This option is of primary importance when using the E-ACSL plug-in: it must be set to the value corresponding to the machine which the generated code will be executed on if the code *or the annotation* is machine dependent.

Consider for instance the following program.

File **archi.c**

```
#define ARCH_BITS 64 /* assume a 64 bits architecture */

#if ARCH_BITS == 32
#define SIZEOF_LONG 4
#elif ARCH_BITS == 64
#define SIZEOF_LONG 8
#endif

int main(void) {
  /*@ assert sizeof(long) == SIZEOF_LONG; */
  return 0;
}
```

We can generate an instrumented code and compile it through the following command lines (the option `-pp-annot` must be used to preprocess the annotations [4]):

```
$ frama-c -pp-annot -e-acsl archi.c -then-on e-acsl -print -ocode monitored_archi.i
$ gcc -o archi `frama-c -print-share-path`/e-acsl/e_acsl.c monitored_archi.i
```

However, the generated code crashes at runtime on a x86 64 bits computer because a 32 bits architecture was assumed at generation time.

```
$ ./archi
Assertion failed at line 10 in function main.
The failing predicate is:
sizeof(long) == 8.
```

There is no assertion failure if you add the option `-machdep x86_64` when calling FRAMA-C.

```
$ frama-c -machdep x86_64 -pp-annot -e-acsl archi.c \
          -then-on e-acsl -print -ocode monitored_archi.i
```

2.2.3 Integers

E-ACSL has got a type `integer` which exactly corresponds to mathematical integers. Such a type does not fit in any integral C type. To circumvent this issue, E-ACSL uses the GMP library[2]. Thus, even if E-ACSL does its best to use standard C integral type instead of GMP [8], it may generate such integers from time to time. In such cases, the generated code must be linked against GMP to be executed.

Consider for instance the following program.

File **gmp.i**

```
unsigned long long my_pow(unsigned int x, unsigned int n) {
  int res = 1;
  while (n) {
    if (n & 1) res *= x;
    n >>= 1;
    x *= x;
  }
  return res;
}
```

[2] http://gmplib.org

```
int main(void) {
  unsigned long long x = my_pow(2, 63);
  /*@ assert (2 * x + 1) % 2 == 1; */
  return 0;
}
```

Even on a 64 bits machine, it is not possible to compute the assertion with a standard C type. In this case, the E-ACSL plug-in generates GMP code.

We can generate an instrumented code as usual through the following command line:

```
$ frama-c -e-acsl gmp.i -then-on e-acsl -print -ocode monitored_gmp.i
```

To compile it however, you need to have GMP installed and to add the option `-lgmp` to GCC as follows:

```
$ gcc -o gmp `frama-c -print-share-path`/e-acsl/e_acsl.c monitored_gmp.i -lgmp
```

We can now execute it as usual.

```
$ ./gmp
```

Since the assertion is valid, there is no output in this case.

2.2.4 Memory-related Annotations

The E-ACSL plug-in handles memory-related annotations like \valid. When using such an annotation, the generated code must be linked against a dedicated library installed with the plug-in to be executed. This library includes two C files which are installed in the E-ACSL' share directory:

- `memory_model/e_acsl_bittree.c`; and
- `memory_model/e_acsl_mmodel.c`.

Consider for instance the following program.

File **valid.c**

```
#include "stdlib.h"

extern void *malloc(size_t);
extern void free(void*);

int main(void) {
  int *x;
  x = (int*)malloc(sizeof(int));
  /*@ assert \valid(x); */
  free(x);
  /*@ assert freed: \valid(x); */
  return 0;
}
```

Assuming that we want to execute the generated code on a x86 64 bits machine, the generation of the instrumented code requires the use of the option `-machdep` since the code uses `sizeof`. However, nothing more is required here.

```
$ frama-c -machdep x86_64 -e-acsl valid.c -then-on e-acsl -print -ocode monitored_valid.i
```

However, since the original code uses \valid, the executable binary must be generated from `monitored_valid.i` as follows:

```
$ DIR=`frama-c -print-share-path`/e-acsl
$ MDIR=$DIR/memory_model
$ gcc -o valid $DIR/e_acsl.c $MDIR/e_acsl_bittree.c $MDIR/e_acsl_mmodel.c monitored_valid.i
```

Now we can execute the generate binary which fails at runtime since the second assertion is violated.

```
$ ./valid
Assertion failed at line 11 in function main.
The failing predicate is:
freed: \valid(x).
```

Like for integers, we do our best to use the dedicated library only when required [9]. So, if your program does not contain memory-related annotations, the generated one does not require to be linked against the dedicated memory library, like the examples of the previous sections.

However, if your program has annotations with pointer dereferencing (for instance), then the generated code *does* require to be linked against the dedicated library at compile time. Why? Because pointer dereferencing may lead to runtime errors, so the E-ACSL plug-in inserts runtime checks to prevent them according to Section 2.2.1 as shown by the following example.

File **pointer.c**

```
#include "stdlib.h"

extern void *malloc(size_t);
extern void free(void*);

int main(void) {
  int *x;
  x = (int*)malloc(sizeof(int));
  *x = 1;
  /*@ assert *x == 1; */
  free(x);
  /*@ assert freed: *x == 1; */
  return 0;
}
```

```
$ frama-c -machdep x86_64 -e-acsl pointer.c -then-on e-acsl -print -ocode
  monitored_pointer.i
$ DIR=`frama-c -print-share-path`/e-acsl
$ MDIR=$DIR/memory_model
$ gcc -o pointer $DIR/e_acsl.c $MDIR/e_acsl_bittree.c $MDIR/e_acsl_mmodel.c\
      monitored_pointer.i
$ ./pointer
RTE failed at line 12 in function main.
The failing predicate is:
mem_access: \valid_read(x).
```

2.2.5 Execution Behavior Environment

When a predicate is checked at runtime, the function `e_acsl_assert` is called. Its body is defined in the file `e_acsl.c` from the E-ACSL library. By default, it does nothing if the predicate is valid, while it prints an error message and exits (with status 1) if the predicate is invalid.

It is however possible to modify this behavior by providing your own definition of `e_acsl_assert`. You must only respect the signature of the function as declared in the file `e_acsl.h` of the E-ACSL library. Below is an example which prints the validity status of each property but never exits.

File **my_assert.c**

```
#include "stdio.h"

void e_acsl_assert(int predicate,
                   char *kind,
                   char *fct,
                   char *pred_txt,
                   int line)
{
  printf("%s at line %d in function %s is %s.\n\
The verified predicate was: `%s'.\n",
         kind, line, fct, predicate ? "valid" : "invalid", pred_txt);
}
```

Then you can generate the program as usual, but use your own file to compile it instead `e_acsl.c` as shown below (we reuse the initial example `first.i` of this manual).

```
$ frama-c -e-acsl first.i -then-on e-acsl -print -ocode monitored_first.i
$ gcc -o customized_first my_assert.c monitored_first.i
$ ./customized_first
Assertion at line 3 in function main is valid.
The verified predicate was: `x == 0'.
Assertion at line 4 in function main is invalid.
The verified predicate was: `x == 1'.
```

2.3 Incomplete Program

Executing a C program requires to have a complete application. However, the E-ACSL plug-in does not necessarily require to get it to generate the instrumented code.

2.3.1 Program without Main

The E-ACSL plug-in may work even if there is no `main` function. Consider for instance the following annotated program without such a `main`.

File **no_main.i**

```
/*@ behavior even:
  @   assumes n % 2 == 0;
  @   ensures \result >= 1;
  @ behavior odd:
  @   assumes n % 2 != 0;
  @   ensures x >= 0 ==> \result >= 1;
  @   ensures x < 0 ==> \result <= -1; */
unsigned long long my_pow(unsigned int x, unsigned int n) {
  int res = 1;
  while (n) {
    if (n & 1) res *= x;
    n >>= 1;
    x *= x;
  }
  return res;
}
```

The instrumented code is generated as usual, even if you get an additional warning.

```
$ frama-c -e-acsl no_main.i -then-on e-acsl -print -ocode monitored_no_main.i
<skip preprocessing command line>
[e-acsl] beginning translation.
[e-acsl] warning: cannot find entry point `main'.
                  Please use option `-main' for specifying a valid entry point.
                  The generated program may miss memory instrumentation.
[e-acsl] translation done in project "e-acsl".
```

This warning indicates that the instrumentation would be incorrect if the program contains memory-related annotations (see Section 3.2). That is not the case here, so you can safely ignore it. Now, it is possible to compile the generated code with a C compiler in a standard way, and even to link it against additional files like the following one. In this particular case, we also need to link against GMP as explained in Section 2.2.3.

File **main.c**

```
#include "stdio.h"

int main(void) {
  unsigned long long x = my_pow(2, 17);
  printf("x=%llu\n",x);
  return 0;
}
```

```
$ gcc -o no_main `frama-c -print-share-path`/e-acsl/e_acsl.c main.c -lgmp
$ ./no_main
x=131072
```

2.3.2 Function without Code

2.4 Combining E-ACSL with Others Plug-ins

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

annotations are kept in order to analyze the generated code

possible to run rte first

combination with value and wp

2.5 Customization

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

2.6 Verbosing Policy

By default, E-ACSL does not provide many information when it is running. Mainly, it prints a message when it begins the translation, and one other when the translation is done. It may also displays warnings when something usually requires the attention of the user, for instance if it is not able to translate an annotation. Such information is usually enough but, in some cases, you might want to get additional controls on what is displayed. As quite usual in Frama-C, the E-ACSL plug-in offers two different ways to do this: the verbosing level which indicates the *amount* of information to display, and the message categories which indicates the *kind* of information to display.

2.6.1 Verbosing Level

The amount of information displayed by the E-ACSL plug-in is settable by the option `-e-acsl-verbose`. It is 1 by default. Below is indicated which information is displayed according to the verbosing level. The level n also displays the information of the level $n-1$.

`-e-acsl-verbose 0`	only warnings and errors
`-e-acsl-verbose 1`	beginning and ending of the translation
`-e-acsl-verbose 2`	different parts of the translation and about functions
`-e-acsl-verbose 3`	about predicates and statements
`-e-acsl-verbose 4 and above`	about terms and expressions

2.6.2 Message Categories

The kind of information to display is settable by the option `-e-acsl-msg-key` (and unsettable by the option `-e-acsl-msg-key-unset`). The different keys refer to different parts of the translation, as detailed below.

analysis	minimization of the instrumentation for memory-related annotation (section 2.2.4)
duplication	duplication of functions with contracts (section 2.3.2)
translation	translation of an annotation into C code
typing	minimization of the instrumentation for integers (section 2.2.3)

Chapter 3

Known Limitations

The development of the E-ACSL plug-in is still ongoing. First, the whole E-ACSL reference manual [10] is not yet supported. Which annotations are already translated into C code and which are not is defined in a separated document [11]. 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 [4]). 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[1].

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 Program without Main

3.3 Function without Code

3.4 Recursive Function

3.5 Variadic Function

- Not yet duplicated

3.6 Function Pointer

[1] 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] Bernard Botella, Mickaël Delahaye, Stéphane Hong-Tuan-Ha, Nikolai Kosmatov, Patricia Mouy, Muriel Roger, and Nicky Williams. Automating structural testing of C programs: Experience with PathCrawler. In *the 4th Int. Workshop on Automation of Software Test (AST 2009)*, pages 70–78. IEEE Computer Society, 2009.

[3] 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.

[4] 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.

[5] 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.

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

[7] Pascal Cuoq, Boris Yakobowski, and Virgile Prevosto. *Frama-C's value analysis plug-in*, April 2013. http://frama-c.cea.fr/download/value-analysis.pdf.

[8] 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.

[9] Nikolaï Kosmatov, Guillaume Petiot, and Julien Signoles. Optimized Memory Monitoring for Runtime Assertion Checking of C Programs. Submitted for publication.

[10] 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.

[11] 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-msg-key`, ***20***
`-e-acsl-msg-key-unset`, ***20***
`-e-acsl-verbose`, ***20***
`e_acsl_assert`, 13, ***17***

Function
    Pointer, 21
    Recursive, 21
    Variadic, 21
    Without code, 19, 21

GMP, 15

Installation, 9
`integer`, 15

`-lgmp`, 16

`-machdep`, 15, 16

`-ocode`, 13

`-pp-annot`, 15
`-print`, 12
Program
    Without main, 18, 21

Runtime Error, 14

`-then-on`, 12

Undefined value, 21

Value, 9

`-Wno-attributes`, 13
Wp, 9