frama C

Software Analyzers

E-ACSL User Manual

E-ACSL Plug-in

Release 0.4.1 compatible with Frama-C Neon

Julien Signoles

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

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

Contents

Foreword

This is the user manual of the Frama-C plug-in E-ACSL[1]. The contents of this document correspond to its version 0.4.1 (August 7, 2014) compatible with the version Neon 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.

Acknowledgements

We gratefully thank all the people who contributed to this document: Florent Kirchner, Nikolaï Kosmatov and Guillaume Petiot.

[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, version 0.4.1. The E-ACSL version you are using is indicated by the command `frama-c -e-acsl-version`. This plug-in automatically translates an annotated C program into another program that fails at runtime if an annotation is violated. If no annotation is violated, the behavior of the new program is exactly the same as 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]. This is the primary goal of E-ACSL. Second it allows the combination of Frama-C and its existing analyzers, with other analyzers for C like PathCrawler [2] that do not natively 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 assertions that cannot be verified statically at runtime 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 [12, 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. E-ACSL annotations currently handled by the E-ACSL plug-in are documented in a separated document [13].

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. Also this manual is *not* a full tutorial about Frama-C and E-ACSL, even if it provides some examples. You can still refer to any external tutorial [11] for additional examples.

[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. You can still call the option `-e-acsl-help` to get the list of available options with few lines of documentation. First, Section 2.1 shows how to run the plug-in on a toy example and how to compile the generated code with a standard C compiler and 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 are not defined or in which there are no main function. Section 2.4 explains how to combine the E-ACSL plug-in with other Frama-C plug-ins. Finally, Section 2.5 introduces how to customize the plug-in, and 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 form the E-ACSL library. Their usage 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. This is also possible through the command line thanks to the kernel options `-then-on` and `-print` which respectively switch 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`. They are part of the included E-ACSL monitoring library. You can safely ignore it right now. The translated `main` function of `first.i` is displayed at the end. After each E-ACSL annotation, a line has been added.

```
/*@ 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 precise error reports as shown 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 the E-ACSL-instrumented 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 exit code 1 and an error message indicating which 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 necessarily the same as the one assumed by Frama-C. You should take care of that when running the E-ACSL plug-in and when compiling the generated code with Gcc. In this aspect, the plug-in offers a few possibilities of customization.

2.2.1 Runtime Errors in Annotations

The major difference between ACSL [1] and E-ACSL [12] specification languages is that the logic is total in the former 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 responsibility of each tool which interprets* E-ACSL *to ensure that an undefined term is never evaluated"* [12].

Accordingly, the E-ACSL plug-in prevents an undefined term from being evaluated. Should it 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 is_dividable(int x, int y) {
  return x % y == 0;
}

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

The terms and predicates containing the modulo '%' in the `assumes` clause are undefined in the context of the `main` 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 `rte` is executed, you get the following output indicating 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 it is executed on. The program must be compiled on the very same architecture (or cross-compiled for it) for the compiler being able to generate a correct binary.

Frama-C makes assumptions about the machine architecture when analyzing such a file. By default, it assumes an X86 32-bit platform, but it may be customized 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 architecture 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-bit 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 fails at runtime on an X86 64-bit computer because a 32-bit 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 running the E-ACSL plug-in.

```
$ 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 integral types 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;
  }
```

[2] http://gmplib.org

```
  return res;
}

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

Even on a 64-bit 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 pow_gmp `frama-c -print-share-path`/e-acsl/e_acsl.c monitored_gmp.i -lgmp
```

We can now execute it as usual.

```
$ ./pow_gmp
```

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

The option `-e-acsl-gmp-only` (unset by default) may be set to always generate GMP integers, even when it is not required. If it is set, the generated program must be linked against GMP as soon as there is an integer or any integral C type in an E-ACSL annotation.

2.2.4 Memory-related Annotations

The E-ACSL plug-in handles memory-related annotations such as \valid. When using such an annotation, the generated code must be linked against a specific memory library installed with the plug-in. 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 an X86 64-bit machine, the generation of the instrumented code requires the use of the option `-machdep` since the code uses `sizeof`. However, nothing is required here.

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

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 -DE_ACSL_MACHDEP=x86_64 \
      $DIR/e_acsl.c $MDIR/e_acsl_bittree.c $MDIR/e_acsl_mmodel.c \
      monitored_valid.i
```

The GCC's option `-DE_ACSL_MACHDEP=x86_64` indicates to preprocess the memory library for an x86-64bits architecture.

Now we can execute the generated 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, the E-ACSL plug-in does its best to use the dedicated memory library only when required [10]. 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 contains some 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, witness the following example.

File **pointer.c**

```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).
```

The option `-e-acsl-full-mmodel` (unset by default) may be set to systematically instrument the code for handling potential memory-related annotations, even when it is not required. If it is set, the generated program must be always linked against the memory library.

2.2.5 Runtime Monitor Behavior

When a predicate is checked at runtime, the function `e_acsl_assert` is called. Its body is defined in the file `e_acsl.c` of 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 of `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 Programs

Executing a C program requires to have a complete application. However, the E-ACSL plug-in does not necessarily require to have it to generate the instrumented code. It is still possible to instrument a partial program with no entry point or in which some functions remain undefined, even though a few restrictions.

2.3.1 Programs 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;
  @   \result >= 1; */
unsigned long long my_pow(unsigned int x, unsigned int n) {
  unsigned long long res = 1;
```

```
  while (n) {
    if (n & 1) res *= x;
    n >>= 1;
    x *= x;
  }
  return res;
}
```

The instrumented code is generated as usual, even though 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.
                  if there are memory-related annotations.
[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 3.2.1). 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, 16);
  printf("x = %llu\n",x);
  return 0;
}
```

```
$ gcc -o with_main `frama-c -print-share-path`/e-acsl/e_acsl.c \
      monitored_no_main.i main.c -lgmp
$ ./with_main
x = 65536
```

2.3.2 Undefined Functions

The E-ACSL plug-in may also work if some functions have no implementation. Consider for instance the following annotated program for which the implementation of the function `my_pow` is not provided.

File **no_code.c**

```
#include "stdio.h"

/*@ behavior even:
  @   assumes n % 2 == 0;
  @   ensures \result >= 1;
  @ behavior odd:
  @   assumes n % 2 != 0;
  @   ensures \result >= 1; */
extern unsigned long long my_pow(unsigned int x, unsigned int n);

int main(void) {
  unsigned long long x = my_pow(2, 64);
  return 0;
}
```

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

```
[e-acsl] beginning translation.
[e-acsl] warning: annotating undefined function 'my_pow':
                  the generated program may miss memory instrumentation
                  if there are memory-related annotations.
[kernel] warning: No code nor implicit assigns clause for function my_pow,
generating default assigns from the prototype
[e-acsl] translation done in project "e-acsl".
```

Like in the previous Section, this warning indicates that the instrumentation would be incorrect if the program contains memory-related annotations (see Section 3.2.2). That is still 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 that provide an implementation for the missing functions such as the following one. In this particular case, we also need to link against GMP as explained in Section 2.2.3.

File **pow.i**

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

```
$ gcc -o no_code `frama-c -print-share-path`/e-acsl/e_acsl.c \
      pow.i monitored_no_code.i -lgmp
$ ./no_code
Postcondition failed at line 8 in function my_pow.
The failing predicate is:
\old(n%2 != 0) ==> \result >= 1.
```

The execution of the corresponding binary fails at runtime: actually, our implementation of the function `my_pow` that we use several times since the beginning of this manual may overflow in case of large exponentiations.

2.4 Combining E-ACSL with Other Plug-ins

As the E-ACSL plug-in generates a new Frama-C project, it is easy to run any plug-in on the generated program, either in the same Frama-C session (thanks to the option `-then` or through the GUI), or in another one. The only issue might be that, depending on the plug-in, the analysis may be imperfect if the generated program uses GMP or the dedicated memory library: both make intensive use of dynamic structures which are usually difficult to handle by analysis tools.

Another way to combine E-ACSL with other plug-ins is to run E-ACSL last. For instance, the RTE plug-in [9] may be used to generate annotations corresponding to runtime errors. Then the E-ACSL plug-in may generate an instrumented program to verify that there are no such runtime errors during the execution of the program.

Consider the following program.

File **combine.i**

```
int main(void) {
  int x = 0xffff;
  int y = 0xfff;
  int z = x + y;
  return 0;
}
```

To check at runtime that this program does not perform any runtime error (which are potential overflows in this case), just do:

```
$ frama-c -rte combine.i -then -e-acsl -then-on e-acsl -print \
          -ocode monitored_combine.i
$ gcc -o combine `frama-c -print-share-path`/e-acsl monitored_combine.i
$ ./combine.
```

Nevertheless if you run the E-ACSL plug-in after another one, it first generates a new temporary project in which it links the analyzed program against its own library in order to generate the Frama-C internal representation of the C program (*aka* AST), as explained in Section 2.1.1. Consequently, even if the E-ACSL plug-in keeps the maximum amount of information, the results of already executed analyzers (such as validity status of annotations) are not known in this new project. If you want to keep them, you have to set the option `-e-acsl-prepare` when the first analysis is asked for.

In this context, the E-ACSL plug-in does not generate code for annotations proven valid by another plug-in, except if you explicitly set the option `-e-acsl-valid`. For instance, Value [7] is able to prove that there is no potential overflow in the previous program, so the E-ACSL plug-in does not generate additional code for checking them if you run the following command.

```
$ frama-c -e-acsl-prepare -rte combine.i -then -val -then -e-acsl \
          -then-on e-acsl -print -ocode monitored_combine.i
```

The additional code will be generated with one of the two following commands.

```
$ frama-c -e-acsl-prepare -rte combine.i -then -val -then -e-acsl \
          -e-acsl-valid -then-on e-acsl -print -ocode monitored_combine.i
$ frama-c -rte combine.i -then -val -then -e-acsl \
          -then-on e-acsl -print -ocode monitored_combine.i
```

In the first case, that is because it is explicitly required by the option `-e-acsl-valid` while, in the second case, that is because the option `-e-acsl-prepare` is not provided on the command line which results in the fact that the result of the value analysis are unknown when the E-ACSL plug-in is running.

2.5 Customization

There are several ways to customize the E-ACSL plug-in.

First, the name of the generated project – which is `e-acsl` by default – may be changed by setting the option `-e-acsl-project`.

Second, the directory where the E-ACSL library files are searched in – which is `` `frama-c -print-share-path`/e-acsl `` by default – may be changed by setting the option `-e-acsl-share`.

Third, the option `-e-acsl-check` does not generate any new project but it only verifies that each annotation is translatable. Then it produces a summary as shown in the following example (left shifts in annotation are not yet supported by the E-ACSL plug-in).

File **check.i**

```
int main(void) {
  int x = 0;
  /*@ assert x == 0; */
  /*@ assert x << 2 == 0; */
  return 0;
}
```

```
$ frama-c -e-acsl-check check.i
<skip preprocessing commands>
check.i:4:[e-acsl] warning: E-ACSL construct 'left/right shift' is not yet supported.
                   Ignoring annotation.
[e-acsl] 0 annotation was ignored, being untypable.
[e-acsl] 1 annotation was ignored, being unsupported.
```

2.6 Verbosity Policy

By default, E-ACSL does not provide much information when it is running. Mainly, it prints a message when it begins the translation, and another one when the translation is done. It may also display warnings when something 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 control on what is displayed. As quite usual with Frama-C plug-ins, E-ACSL offers two different ways to do this: the verbosity level which indicates the *amount* of information to display, and the message categories which indicate the *kind* of information to display.

2.6.1 Verbosity 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 all the lower levels.

`-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 functions-related information
`-e-acsl-verbose 3`	predicates- and statements-related information
`-e-acsl-verbose 4 and above`	terms- and expressions-related information

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 E-ACSL reference manual [12] is not yet fully supported. Which annotations are already translated into C code and which are not is defined in a separated document [13]. Second, even though we do our best to avoid them, 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 tedious to lift. Please contact us if you are interested in lifting these limitations[1].

3.1 Uninitialized Values

As explained in Section 2.2.1, the E-ACSL plug-in should never translate an annotation into C code which can lead to a runtime error. This is enforced, except for uninitialized values which are values read before having been written.

File **uninitialized.i**

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

If you generate the instrumented code, compile it, and finally execute it, you may get no runtime error depending on your C compiler, but the behavior is actually uninitialized and should be caught by the E-ACSL plug-in. That is not the case yet.

```
$ frama-c -e-acsl uninitialized.i -then-on e-acsl -print \
          -ocode monitored_uninitialized.i
$ gcc -Wuninitialized -o pointer `frama-c -print-share-path`/e-acsl/e_acsl.c` \
      monitored_uninitialized.i
monitored_uninitialized.i: In function 'main':
monitored_uninitialized.i:44:16: warning: 'x' is used uninitialized in this function\
[-Wuninitialized]
$ ./monitored_uninitialized
```

This is more a design choice than a limitation: should the E-ACSL plug-in generate additional instrumentation to prevent such values from being evaluated, the generated code would be much more verbose and slower.

If you really want to track such uninitializations in your annotation, you have to manually add calls to the E-ACSL predicate `\initialized` [12].

[1] Read http://frama-c.com/support.html for additional details.

3.2 Incomplete Programs

Section 2.3 explains how the E-ACSL plug-in is able to handle incomplete programs, which are either programs without main, or programs containing undefined functions (*i.e.* functions without body).

However, if such programs contain memory-related annotations, the generated code may be incorrect. That is made explicit by a warning displayed when the E-ACSL plug-in is running (see examples of Sections 2.3.1 and 2.3.2).

3.2.1 Programs without Main

The instrumentation in the generated program is partial for every program without main containing a memory-related annotations, except if the option `-e-acsl-full-mmodel` is provided. In that case, violations of such annotations are undetected.

Consider the following example.

File **valid_no_main.c**

```
#include "stdlib.h"

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

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

You can generate the instrumented program as follows.

```
$ frama-c -e-acsl-full-mmodel -machdep x86_64 -e-acsl valid_no_main.c \
          -then-on e-acsl -print -ocode monitored_valid_no_main.i
<skip preprocessing commands>
[e-acsl] beginning translation.
<skip warnings about annotations from the Frama-C libc
      which cannot be translated>
[kernel] warning: no entry point specified:
                  you must call function '__e_acsl_memory_init' and '__e_acsl_memeory_clean' by your
[e-acsl] translation done in project "e-acsl".
```

The last warning states an important point: if this program is linked against another file containing a `main` function, then this main function must be modified to insert a call to the function `__e_acsl_memory_init` at the very beginning and a call to the function `__e_acsl_memory_clean` at the very end like in the following example.

File **modified_main.c**

```
extern void e_acsl_global_init(void);
extern void __clean(void);
extern void f(void);

int main(void) {
  e_acsl_global_init();
  f();
  __clean();
  return 0;
}
```

Then just compile and run it as explained in Section 2.2.4.

```
$ DIR=`frama-c -print-share-path`/e-acsl
$ MDIR=$DIR/memory_model
$ gcc -o valid_no_main $DIR/e_acsl.c $MDIR/e_acsl_bittree.c \
      $MDIR/e_acsl_mmodel.c monitored_valid_no_main.i modified_main.c
$ ./valid_no_main
Assertion failed at line 11 in function f.
The failing predicate is:
freed: \valid(x).
```

Also, if the unprovided main initializes some variables, running the instrumented code (linked against this main) could print some warnings from the E-ACSL memory library[2].

3.2.2 Undefined Functions

The instrumentation in the generated program is partial for a program p if and only if p contains a memory-related annotation a and an undefined function f such that:

- either f has an (even indirect) effect on a left-value occurring in a;
- or a is one of the post-conditions of f.

A Violation of such an annotation a is undetected. There is no workaround yet.

Also, the option `-e-acsl-check` does not verify the annotations of undefined functions. There is also no workaround yet.

3.3 Recursive Function

Programs containing recursive functions have the same limitations as the ones containing undefined functions (Section 3.2.2) and memory-related annotations.

Also, even though there is no such annotations, the generated code may call a function before it is declared. When this behavior appears remains unspecifed. The generated code is however easy to fix by hand.

3.4 Variadic Functions

Programs containing variadic undefined functions but with a function contract are not yet supported. There is no workaround yet.

3.5 Function Pointers

Programs containing function pointers have the same limitations on memory-related annotations as the ones containing undefined function or recursive functions.

[2] see https://bts.frama-c.com/view.php?id=1696 for an example

Appendix A

Changes

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

E-ACSL 0.4.1

- **Bibliography:** fix incorrect links.

E-ACSL 0.4

- No change

E-ACSL 0.3

- **Introduction:** reference the E-ACSL tutorial.
- **Memory-related Annotations:** document the `E_ACSL_MACHDEP` macro.

E-ACSL 0.2

- 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. Version 1.8*, April 2014.

[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] Philippe Herrmann and Julien Signoles. *Annotation Generation: Frama-C's RTE plug-in*, April 2013. http://frama-c.com/download/frama-c-rte-manual.pdf.

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

[11] Nikolaï Kosmatov and Julien Signoles. A lesson on runtime assertion checking with Frama-C. In *International Conference on Runtime Verification (RV 2013)*, volume 8174 of *LNCS*, pages 386–399. Springer, September 2013.

[12] Julien Signoles. *E-ACSL: Executable ANSI/ISO C Specification Language. Version 1.8*, April 2014. http://frama-c.com/download/e-acsl/e-acsl.pdf.

[13] Julien Signoles. *E-ACSL Version 1.7. Implementation in Frama-C Plug-in E-ACSL version 0.4*, April 2014. http://frama-c.com/download/e-acsl/e-acsl-implementation.pdf.

Index