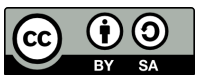


E-ACSL Plug-in

For Frama-C 34.0 beta (Selenium)

Julien Signoles, Basile Desloges, Jan Rochel, and Kostyantyn Vorobyov





Work licensed under Creative Commons BY-SA licence
https://creativecommons.org/licenses/by-sa/4.0/

© 2013-2026 CEA-List

Contents

Foreword 4

1 **Introduction** 5

2 **What the Plug-in Provides** 6

2.1 Simple Example . . . 6
2.1.1 Running E-ACSL . . . 6
2.1.2 Executing the generated code . . . 8
2.2 E-ACSL Wrapper Script . . . 9
2.2.1 Standard Usage . . . 9
2.2.2 Customising Frama-C and Gcc Invocations . . . 10
2.2.3 Verbosity Levels and Output Redirection . . . 10
2.2.4 Architecture Dependency . . . 10
2.2.5 Concurrency Support . . . 11
2.2.6 Documentation . . . 12
2.3 Execution Environment of the Generated Code . . . 12
2.3.1 Runtime Errors in Annotations . . . 12
2.3.2 Libc . . . 13
2.3.3 Integers . . . 13
2.3.4 Memory-related Annotations . . . 15
2.3.5 Runtime Monitor Behavior . . . 16
2.3.6 Using C functions in annotations . . . 19
2.4 Additional Verifications . . . 19
2.4.1 Format String Vulnerabilities . . . 19
2.4.2 Incorrect Usage of Libc Functions . . . 20
2.5 Incomplete Verification . . . 20
2.5.1 Programs with no Main Function . . . 21
2.5.2 Undefined Functions . . . 21
2.5.3 Providing a White List of Functions . . . 22
2.5.4 Partial Instrumentation (Experimental) . . . 23

2.6 Combining E-ACSL with Other Plug-ins . . . 24
2.7 Customization . . . 25
2.8 Verbosity Policy . . . 25
2.8.1 Verbosity Level . . . 25
2.8.2 Message Categories . . . 26

3 **Known Limitations** 27
3.1 Supported Systems . . . 27
3.1.1 Operating Systems . . . 27
3.1.2 Architectures and Runtime Library . . . 27
3.2 Uninitialized Values . . . 27
3.3 Incomplete Programs . . . 28
3.3.1 Programs without Main . . . 28
3.3.2 Undefined Functions . . . 29
3.3.3 Incomplete Types . . . 29
3.4 Recursive Functions . . . 30
3.5 Variadic Functions . . . 30
3.6 Function Pointers . . . 30
3.7 Requirements to Input Programs . . . 30
3.7.1 E-ACSL Namespace . . . 30
3.7.2 Memory Management Functions . . . 30

A **Changes** 31

Bibliography 34

Index 36

Foreword

This is the user manual of the Frama-C plug-in E-ACSL[1]. The contents of this document correspond to Frama-C [7, 1]'s version 34.0 beta (Selenium). The development of the E-ACSL plug-in is still ongoing. Features described by this document may evolve in the future.

Acknowledgements

We gratefully thank all the people who contributed to this document: Pierre-Loïc Garoche, Jens Gerlach, Florent Kirchner, Nikolaï Kosmatov, André Maroneze, Fonenantsoa Maurica and Guillaume Petiot.

[1] https://frama-c.com/fc-plugins/e-acsl.html

Introduction 1

Frama-C [7, 10] is a modular analysis framework for the C programming language which supports the ACSL specification language [3]. This manual documents the E-ACSL plug-in of Frama-C, version 34.0 beta (Selenium). E-ACSL 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.

E-ACSL translation brings several benefits. First, it allows a user to monitor C code and perform what is usually referred to as "runtime assertion checking" [6][1]. This is the primary goal of E-ACSL. Indirectly, in combination with the RTE plug-in [9] of Frama-C, this usage allows the user to detect undefined behaviors in its C code. Second, it allows to combine Frama-C and its existing analyzers with other C analyzers 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 and thus to establish a link between runtime monitoring and static analysis tools such as Eva [5] or Wp [2].

Annotations used by the plug-in must be written in the E-ACSL specification language [15, 8] – a subset of ACSL. E-ACSL plug-in is still in a preliminary state: some parts of the E-ACSL specification language are not yet supported. Annotations supported by the plugin are described in a separate document [16]. It is worth noting that the annotations that aim to be dynamically verified are not necessarily hand-written, but may be automatically generated instead. That is for instance the case when checking the absence of undefined behaviors in combination with RTE, as mentioned in the previous paragraph. Using E-ACSL this way is therefore a fully automatic process. Many usages, including automatic usages, are described in companion research papers [13, 18, 17].

The E-ACSL plug-in is installed with Frama-C, but this manual does *not* explain how to install Frama-C. For installation instructions please refer to the `INSTALL`[2] file in the Frama-C distribution. Furthermore, even though this manual provides examples, it is *not* a full comprehensive tutorial on Frama-C or E-ACSL.

[1] In our context, "runtime annotation checking" would be more precise.

[2] https://git.frama-c.com/pub/frama-c/blob/master/INSTALL.md

What the Plug-in Provides 2

This chapter is the core of this manual and describes how to use the E-ACSL plug-in. First, Section 2.1 shows how to run the plug-in on a toy example, compile the generated code with the Gcc compiler and detect invalid annotations at runtime. Further, Section 2.2 describes `e-acsl-gcc` – a convenience wrapper script around Frama-C and Gcc. The aim of this script is to simplify instrumentation and compilation of the instrumented code. Section 2.3 gives additional details on the execution of the generated code. Next, Section 2.5 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 2.6 explains how to combine the E-ACSL plug-in with other Frama-C plug-ins. Finally, Section 2.7 introduces how to customize the plug-in, and Section 2.8 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 whether an E-ACSL annotation is violated at runtime.

2.1.1 Running E-ACSL

Consider the following program containing two E-ACSL assertions such that the first assertion is valid, whereas 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 `first.i` consists of adding `-e-acsl` option to the Frama-C command line:

```
$ frama-c -e-acsl first.i
[kernel] Parsing first.i (no preprocessing)
[e-acsl] beginning translation.
[kernel] Parsing FRAMAC_SHARE/e-acsl/e_acsl.h (with preprocessing)
[e-acsl] translation done in project "e-acsl".
```

Even though `first.i` has already been preprocessed, E-ACSL first asks the Frama-C kernel to process and combine it against several header files provided by the E-ACSL plug-in. Further E-ACSL translates the annotated code into a new Frama-C project named `e-acsl`[1]. By default, the option

[1] The notion of *project* is explained in Section 8.1 of the Frama-C user manual [7].

`-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-last` and `-print`. The former switches to the last generated project, while the latter pretty prints the C code [7]:

```
$ frama-c -e-acsl first.i -then-last -print
[kernel] Parsing first.i (no preprocessing)
[e-acsl] beginning translation.
[kernel] Parsing FRAMAC_SHARE/e-acsl/e_acsl.h (with preprocessing)
[e-acsl] translation done in project "e-acsl".
/* Generated by Frama-C */
#include "stddef.h"
#include "stdio.h"
struct __e_acsl_contract_t;
typedef struct __e_acsl_contract_t __attribute__((FC_BUILTIN)) __e_acsl_contract_t;
struct __e_acsl_mpz_struct {
  int _mp_alloc ;
  int _mp_size ;
  unsigned long *_mp_d ;
};
typedef struct __e_acsl_mpz_struct __e_acsl_mpz_struct;
typedef __e_acsl_mpz_struct ( __attribute__((FC_BUILTIN)) __e_acsl_mpz_t)[1];
struct __e_acsl_mpq_struct {
  __e_acsl_mpz_struct _mp_num ;
  __e_acsl_mpz_struct _mp_den ;
};
typedef struct __e_acsl_mpq_struct __e_acsl_mpq_struct;
typedef __e_acsl_mpq_struct ( __attribute__((FC_BUILTIN)) __e_acsl_mpq_t)[1];
typedef unsigned long __e_acsl_mp_bitcnt_t;
/*@ requires pred != 0;
   assigns \nothing; */
 __attribute__((FC_BUILTIN)) void __e_acsl_assert(int pred,
                                          char const *kind,
                                          char const *fct,
                                          char const *pred_txt,
                                          char const *file,
                                          int line);

extern size_t __e_acsl_heap_allocation_size;

extern size_t __e_acsl_heap_allocated_blocks;

/*@ ghost extern int __fc_heap_status; */

/*@ ghost extern int __e_acsl_init; */

long valid_nstring(char *s, long n, int wrtbl);

long valid_nwstring(wchar_t *s, long n, int wrtbl);

__inline static long valid_string__fc_inline(char *s, int wrtbl)
{
  long tmp;
  tmp = valid_nstring(s,(long)(-1),wrtbl);
  return tmp;
}

__inline static long valid_wstring__fc_inline(wchar_t *s, int wrtbl)
{
```

```
  long tmp;
  tmp = valid_nwstring(s,(long)(-1),wrtbl);
  return tmp;
}

int main(void)
{
  int __retres;
  int x = 0;
  __e_acsl_assert(x == 0,"Assertion","main","x == 0","first.i",3);
  /*@ assert x == 0; */ ;
  __e_acsl_assert(x == 1,"Assertion","main","x == 1","first.i",4);
  /*@ assert x == 1; */ ;
  __retres = 0;
  return __retres;
}
```

As you can see, the generated code contains additional type declarations, constant declarations and global ACSL annotations not present in the initial file `first.i`. They are part of the E-ACSL monitoring library. You can safely ignore them for now. The translated `main` function of `first.i` is displayed at the end. Before each E-ACSL annotation, a call to `__e_acsl_assert` has been added.

```
  __e_acsl_assert(x == 0,"Assertion","main","x == 0","first.i",3);
  /*@ assert x == 0; */ ;
  __e_acsl_assert(x == 1,"Assertion","main","x == 1","first.i",4);
  /*@ assert x == 1; */ ;
```

Each call to `__e_acsl_assert`, function defined by the E-ACSL monitoring library, dynamically verifies the validity of the corresponding assertion. In other words, it checks that its first argument (here `x == 0` or `x == 1`) is not equal to 0 and fails otherwise. The extra arguments are used to display error reports as shown in Section 2.1.2.

2.1.2 Executing the generated code

Using `-ocode` Frama-C [7] option, the code generated by the E-ACSL plug-in can be redirected into a file as follows:

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

Frama-C uses architecture-dependent configuration which affects sizes of integer types, endianness and potentially other features. It can be seen that the code generated from `first.i` (shown in the previous section) defines C type `size_t` as `unsigned long`, whereas in 32-bit architectures `size_t` is typically defined as `unsigned int`. The architecture used during Frama-C translation is controlled through Frama-C `-machdep` option that specifies the architecture type to use during translation. The default value of `-machdep` is `x86_64` (a generic 64-bit x86 architecture). Note that since code generated by E-ACSL is aimed at being compiled it is important that the architecture used by Frama-C matches the architecture corresponding to your compiler and your system. For instance, in a 32-bit machine you should also pass `-machdep x86_32` option as follows:

```
$ frama-c -machdep x86_32 -e-acsl first.i -then-last \
  -print -ocode monitored_first.c
```

This file can be compile with a C compiler (for instance Gcc), as follows:

```
$ gcc -c -Wno-attributes monitored_first.c
```

However, creating an executable through a proper invocation to Gcc is painful and is not documented. Instead, E-ACSL comes with a companion wrapper script for this purpose.

2.2 E-ACSL Wrapper Script

Presently, `e-acsl-gcc` is a recommended way of running the E-ACSL plug-in. This manual further describes features of the E-ACSL plug-in using `e-acsl-gcc` as its main driver for source code instrumentation and compilation of the instrumented programs.

In this section we describe `e-acsl-gcc` and provide several examples of its use.

2.2.1 Standard Usage

The default behaviour of `e-acsl-gcc` is to instrument, compile and link an annotated C program with runtime assertions via a run of Frama-C.

```
$ e-acsl-gcc -omonitored_first.i first.i
```

The above command first instruments `first.i` with runtime assertions and outputs the generated code to `monitored_first.i`. Unless the name of the output file is specified via the `-o` option (i.e., `-omonitored_first.i` in the above example), the generated code is placed to a file named `a.out.frama.c`. It then produces two executables: `a.out` and `a.out.e-acsl`, such that `a.out` is an executable compiled from `first.i`, while `a.out.e-acsl` is an executable compiled from `monitored_first.i` generated by E-ACSL.

The `-n` option can be used to deactivate the compilation and linking and only output the instrumented code. For instance, command

```
$ e-acsl-gcc -n -omonitored_first.i first.i
```

only instruments the code in `first.i` and outputs it to `monitored_first.i`.

You can execute the generate binaries, in particular `a.out.e-acsl` which detects at runtime the incorrect annotation.

```
$ ./a.out.e-acsl
first.i: In function 'main'
first.i:4: Error: Assertion failed:
      The failing predicate is:
      x == 1.
Aborted (core dumped)
$ echo $?
134
```

The execution aborts with a non-zero exit code (134) and an error message indicating an E-ACSL annotation that has been violated. There is no output for the valid E-ACSL annotation. That is, the run of an executable generated from the instrumented source file (i.e., `monitored_first.i`) detects that the second annotation in the initial program is invalid, whereas the first annotation holds for this execution.

Named executables can be created using the `-O` option. This is such that a value supplied to the `-O` option is used to name the executable generated from the original program and the executable generated from the E-ACSL-instrumented code is suffixed `.e-acsl`.

For example, command

```
$ e-acsl-gcc -omonitored_first.i -Omonitored_first first.i
```

names the executables generated from `first.i` and `monitored_first.i`: `monitored_first` and `monitored_first.e-acsl` respectively.

The `e-acsl-gcc -C` option allows to skip the instrumentation step and compile a given program as if it was already instrumented by the E-ACSL plug-in. This option is useful if one makes changes to an already instrumented source file by hand and only wants to recompile it.

```
$ e-acsl-gcc -C -Omonitored_first monitored_first.i
```

The above command generates a single executable named `monitored_first.e-acsl`. This is because `monitored_first.i` is considered to be instrumented code and thus no original program is provided.

2.2.2 Customising Frama-C and Gcc Invocations

By default `e-acsl-gcc` uses `frama-c` and `gcc` executables by looking them up in a system's path. If either of the tools are not found the script fails with an appropriate error message. Paths to Frama-C and Gcc executables can also be passed using `-I` and `-G` options respectively, for instance as follows:

```
$ e-acsl-gcc -I/usr/local/bin/frama-c -G/usr/bin/gcc6 foo.c
```

Runs of Frama-C and Gcc issued by `e-acsl-gcc` can further be customized by using additional flags. Frama-C flags can be passed using the `-F` option, while `-e` and `-l` options allow for passing additional preprocessor and linker flags during Gcc invocations. For example, command

```
$ e-acsl-gcc -F"-unicode" -e"-I/bar -I/baz" foo.c
```

yields an instrumented program `a.out.frama.c` where ACSL formulas are output using the UTF-8 character set. Further, during the compilation of `a.out.frama.c` Gcc adds directories `/bar` and `/baz` to the list of directories to be searched for header files.

It is worth mentioning that `e-acsl-gcc` implements several convenience flags for setting some of the Frama-C options. For instance, `-E` can be used to pass additional arguments to the Frama-C preprocessor (see Section 2.2.6).

2.2.3 Verbosity Levels and Output Redirection

By default `e-acsl-gcc` prints the executed Frama-C and Gcc commands and pipes their output to the terminal. The verbosity of Frama-C output can be changed using `-v` and `-d` options used to pass values to `-verbose` and `-debug` flags of Frama-C respectively. For instance, to increase the verbosity of plug-ins but suppress the verbosity of the Frama-C kernel while instrumenting `foo.c` one can use the following command:

```
$ e-acsl-gcc -v5 -F"-kernel-verbose 0" foo.c
```

Verbosity of the `e-acsl-gcc` output can also be reduced using the `-q` option that suppresses any output except errors or warnings issued by Gcc, Frama-C, and `e-acsl-gcc` itself.

The output of `e-acsl-gcc` can also be redirected to a specified file using the `-s` option. For example, the following command redirects all output during instrumentation and compilation of `foo.c` to the file named `/tmp/log`.

```
$ e-acsl-gcc -s/tmp/log foo.c
```

2.2.4 Architecture Dependency

As pointed out in Section 2.1.2 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 source code. By default, it assumes an X86 64-bit platform, but it can be customized through `-machdep` switch [7]. 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, otherwise the generated program is likely to fail to link against the E-ACSL library. Note that the library is built using the default machine architecture.

One of the benefits of using the wrapper script is that it makes sure that Frama-C is invoked with the correct `-machdep` option. By default `e-acsl-gcc` uses `-machdep gcc_x86_64` for 64-bit architectures and `-machdep gcc_x86_32` for 32-bit ones. Compiler invocations are further passed `-m64` or `-m32` options to generate code using 64-bit or 32-bit ABI respectively.

At the present stage of implementation E-ACSL does not support generating executables with ABI different to the default one.

2.2.5 Concurrency Support

How to Activate Concurrency

E-ACSL supports the monitoring of Linux's `pthreads` concurrent code by using the option `--concurrency` of `e-acsl-gcc`.

Example of compilation with concurrency support using `e-acsl-gcc`:

```
$ e-acsl-gcc foo.c --concurrency -o foo.frama.c -O foo
```

The concurrency support is a two part activation: first the `--e-acsl-concurrency` Frama-C option is enabled to generate concurrent ready code, then when compiling the generated code, the preprocessor variable `E_ACSL_CONCURRENCY_PTHREAD` is defined and the linker option `-pthread` is added.

Example of compilation with concurrency support using Frama-C and Gcc directly (where `$E_ACSL_SHARE` points to the `share/frama-c/e-acsl` folder of Frama-C and `$E_ACSL_LIB` points to the `lib/frama-c/e-acsl` folder of Frama-C):

```
$ frama-c \
     -keep-unused-functions none \
     -machdep gcc_x86_64 \
     -cpp-extra-args="-std=c99 -D_DEFAULT_SOURCE -D__NO_CTYPE \
                      -D__FC_MACHDEP_X86_64" \
     -no-frama-c-stdlib \
     foo.c \
     -e-acsl -e-acsl-concurrency \
     -then-last -print -ocode foo.frama.c
$ gcc \
     -std=c99 -m64 -g -O2 -fno-builtin -fno-merge-constants \
     -Wno-attributes -Wno-implicit-function-declaration \
     -DE_ACSL_SEGMENT_MMODEL -DE_ACSL_CONCURRENCY_PTHREAD \
     -I$E_ACSL_SHARE \
     -o foo.e-acsl \
     foo.frama.c \
     $E_ACSL_SHARE/e_acsl_rtl.c \
     -pthread \
     $E_ACSL_LIB/libeacsl-dlmalloc.a \
     -lgmp -lm
```

Function Contracts Limitations

For now, the E-ACSL language [15] does not support concurrent access indications, so some care should be taken when writing specifications that access to the shared memory. As a workaround, the memory accesses inside the specification can be protected code outside the specification, for instance by acquiring and releasing a lock around E-ACSL assertions.

As an example, the following code acquire the lock on a mutex before accessing the shared global variable `a`. However, E-ACSL will place the translation of the precondition before acquiring the lock,

and the translation of the postcondition after releasing the lock. Therefore, their validity status may vary with respect to the thread interleaving.

```
/*@ requires \valid(a) && \initialized(a);
    requires \valid(b) && \initialized(b);
    ensures *a == \old(*a) + 1;
    ensures *b == \old(*b) + *a; */
void f() {
  pthread_mutex_lock(&lock);
  *a += 1;
  *b += *a;
  pthread_mutex_unlock(&lock);
}
```

The workaround here is to move the function contract to a statement contract surrounded by the lock:

```
void f() {
  pthread_mutex_lock(&lock);
  /*@ requires \valid(a) && \initialized(a);
      requires \valid(b) && \initialized(b);
      ensures *a == \old(*a) + 1;
      ensures *b == \old(*b) + *a; */
  {
    *a += 1;
    *b += *a;
  }
  pthread_mutex_unlock(&lock);
}
```

2.2.6 Documentation

For full up-to-date documentation of `e-acsl-gcc` see its man page:

```
$ man e-acsl-gcc
```

Alternatively, you can also use the `-h` option of `e-acsl-gcc`:

```
$ e-acsl-gcc -h
```

2.3 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.

2.3.1 Runtime Errors in Annotations

The major difference between ACSL [3] and E-ACSL [15] specification languages is that the logic is total in the former while it is partial in the latter one: the semantics of a logic construct c 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 a construct c 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"* [15].

Accordingly, the E-ACSL plug-in prevents an undefined term from being evaluated. If an annotation contains such a term, E-ACSL will report a proper error at runtime instead of evaluating it.

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 using the following command:

```
$ e-acsl-gcc -omonitored_rte.i -Orte rte.i
```

Now, when `rte.e-acsl` is executed, you get the following output indicating that your function contract is invalid because it contains a division by zero.

```
$ ./rte.e-acsl
RTE failed at line 5 in function is_dividable.
The failing predicate is:
division_by_zero: y != 0.
Aborted
```

2.3.2 Libc

By default, FRAMA-C uses its own standard library (*aka* libc) instead of the one of your system. If you do not want to use it, you have to add the option `-no-framac-stdlib` to the FRAMA-C command line.

This might be particularly useful in one of the following contexts:

- several libc functions used by the analyzed program are still not defined in the FRAMA-C libc; or
- your system libc and the FRAMA-C libc mismatch about several function types (for instance, your system libc is not 100% POSIX compliant); or
- several FRAMA-C lib functions get a contract and you are not interested in verifying them (for instance, only checking undefined behaviors matters).

Notably, `e-acsl-gcc` disables FRAMA-C libc by default. This is because most of its functions are annotated with E-ACSL contracts and generating code for these contracts results in additional runtime overheads. To enforce default FRAMA-C contracts with `e-acsl-gcc` you can pass `-L` option to the wrapper script.

2.3.3 Integers

The `integer` type in E-ACSL represents mathematical integers, i.e., integers of arbitrary size, and does therefore in general not fit into any integral C type. To circumvent this issue, E-ACSL uses the GMP library[2]. Since such arbitrary-precision arithmetic is costly, E-ACSL does its best to use standard

integral C types instead of GMP [11, 4]. However, this is not always possible and it may thus generate such GMP integers from time to time. This is notably the case if your program contains `long long`, either signed or unsigned.

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;
}

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. Note that E-ACSL library already includes GMP, thus no additional arguments need to be provided. We can generate and execute the instrumented code as usual via the following command:

```
$ e-acsl-gcc -omonitored_gmp.i -Ogmp gmp.i
$ ./gmp.e-acsl
```

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

Note that it is possible to use GMP arithmetic in all cases (even if not required). This option is controlled through `-g` switch of `e-acsl-gcc` which passes the `-e-acsl-O-gmp-only` option to the E-ACSL plug-in. However it could slow down the execution of the instrumented program significantly.

Tweaking integer arithmetic (experts only)

The analysis for determining whether a value can be represented by an integral C type, is particularly difficult in the presence of (mutually) recursive functions. However, through a process called *widening* as described in [4], E-ACSL may still be able to conclude that an integral C type may be used for a function's arguments or return type. The widening strategy can be tweaked by a number of command-line flags:

```
$ frama-c -e-acsl-widening-arguments f1:3,f2:4
          -e-acsl-widening-arguments-base 5
          -e-acsl-widening-output f1:1,f3:2
          -e-acsl-widening-output-base 0
```

Consider for instance the command-line flags above, indicating that for arguments of functions `f1` and `f2` the values 3 and 4 should be used respectively, and 5 for any other arguments. Furthermore for the result of the functions `f1` and `f3` the values 1 and 2 should be used, and 0 for any other function result. For the effects of these values, refer to [4].

[2] http://gmplib.org

2.3.4 Memory-related Annotations

The E-ACSL plug-in handles memory-related annotations such as `\valid`. 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;
}
```

Even though the first annotation holds, the second annotation (labelled *freed*) is invalid. This is because at the point of its execution memory allocated via `malloc` has been deallocated and `x` is a stale pointer referencing unallocated memory. The execution of the instrumented program thus reports the violation of the second annotation:

```
$ e-acsl-gcc -Ovalid valid.c
$ ./valid.e-acsl
valid.c: In function 'main'
valid.c:11: Error: Assertion failed:
      The failing predicate is:
      freed: \valid(x).
Aborted (core dumped)
```

To check memory-related predicates (such as `\valid`) E-ACSL tracks memory allocated by a program at runtime. E-ACSL records memory blocks in a meta-storage tracking each block's boundaries (i.e., its base address and length), per-byte initialization and the notion of an address being freeable (i.e., whether it can be safely passed to the `free` function). During an execution of the instrumented program code injected by E-ACSL transformations queries the meta-storage to identify properties of an address in question. For instance, during the execution of the above program the monitor injected by E-ACSL first records information about the memory block allocated via a call to `malloc`. Further, during the execution of the first assertion the monitor queries the meta-storage about the address of the memory location pointed to by `x`. Since the location belongs to an allocated memory block, the meta-storage identifies it as belonging to the tracked allocation. Consequently, the assertion evaluates to true and the monitors allows the execution to continue. The second assertion, however, fails. This is because the block pointed to by `x` has been removed from the meta-storage by a call to `free`, and thus a meta-storage query identifies location `x` as belonging unallocated memory space.

E-ACSL Memory Models

Memory tracking implemented by the E-ACSL library comes in two flavours dubbed *bittree-based* and *segment-based* memory models. With the bittree-based model meta-storage is implemented as a compact prefix trie called Patricia trie [12], whereas segment-based memory model is based on shadow memory [19]. The functionality of the provided memory models is equivalent, however they differ in performance. We further discuss performance considerations.

The E-ACSL models allocate heap memory using a customized version of the DLMALLOC[3] memory

[3] http://dlmalloc.net/

allocator replacing system-wide `malloc` and similar functions (e.g., `calloc` or `free`).

At the level of `e-acsl-gcc` you can choose between different memory models using the `-m` switch followed by the name of the memory model to link against. For instance, command

```
$ e-acsl-gcc -mbittree -Ovalid valid.c
```

links a program generated from `valid.c` against the library implementing the bittree-based memory model, whereas the following invocation uses shadow-based tracking:

```
$ e-acsl-gcc -msegment -Ovalid valid.c
```

It is also possible to generate executables using both models as follows:

```
$ e-acsl-gcc -msegment,bittree -Ovalid valid.c
```

In this case, `e-acsl-gcc` produces 3 executables: `valid`, `valid.e-acsl-segment` and `valid.e-acsl-bittree` corresponding to an executable generated from the original source code, and the executables generated from the E-ACSL-instrumented sources linked against segment-based and bittree-based memory models.

Unless specified, E-ACSL defaults to the segment-based memory model.

Performance Considerations

As pointed out in the previous section the functionalities provided by both segment-based and bittree-based memory models are equivalent but differ in performance. The bittree-based model uses compact memory representation of metadata. The segment-based model on the other hand uses significantly more memory. You can expect over 2x times memory overheads when using it. However It is often an order of magnitude faster than the bittree-based model [19].

Maximized Memory Tracking

By default E-ACSL uses static analysis to reduce instrumentation and therefore runtime and memory overheads [14]. With respect to memory tracking this means that E-ACSL may omit recording memory blocks irrelevant for runtime analysis. It is, however, still possible to systematically instrument the code for handling potential memory-related annotations even when it is not required. This feature can be enabled using the `-M` switch of `e-acsl-gcc` or `-e-acsl-full-mtracking` option of the E-ACSL plug-in.

The above-mentioned static analysis is probably the less robust part of E-ACSL for the time being. When handling a large piece of code, it might be necessary to deactivate it and systematically instrument the code as explained above.

2.3.5 Runtime Monitor Behavior

When a predicate is checked at runtime, function `__e_acsl_assert` is called. By default this function does nothing if the predicate is valid, and prints an error message and aborts the execution (by raising an `ABORT` signal via a call to the C function `abort`) if the predicate is invalid. If E-ACSL detects that its verdict is not trustable (see Section 2.5.4), it prints a warning with its guess and continues the execution.

The default behavior of `__e_acsl_assert` can be altered using `--fail-with-code=CODE` option of `e-acsl-gcc` that uses `exit(CODE)` instead of `abort`.

For instance, the program generated using the following command exits with code `5` on a predicate violation.

```
$ e-acsl-gcc --fail-with-code=5 foo.c
```

Forceful termination of a monitored program can be disabled using `--keep-going` option. If specified, `e-acsl-gcc` generates code that reports each property violated at runtime but allows the monitored program to continue execution. `--keep-going` should be used with caution: such unterminated executions may lead to incorrect verification results.

`e-acsl-gcc` also provides a way to use an alternative definition of `__e_acsl_assert`.

Custom implementation of `__e_acsl_assert` should be provided in a separate C (or object file) and respect the following signature:

```
void __e_acsl_assert(int predicate, __e_acsl_assert_data_t * data);
```

With `data` a structure containing the following fields:

- `blocking`: integer representing if the assertion is blocking or not;
- `kind`: C string representing a kind of annotation (e.g., "Assertion");
- `pred_txt`: stringified predicate;
- `file`: un-instrumented file of predicate placement;
- `fct`: function of predicate placement in the un-instrumented file;
- `line`: line of predicate placement in the un-instrumented file;
- `values`: values contributing to the predicate.

Additionally, it may depends on the global variable `__e_acsl_sound_verdict`. This variable of type `int` is set to `0` as soon as the verdict provided by E-ACSL is not trustworthy anymore (see Section 2.5.4).

Below is an example which prints the validity status of each property but never exits.

File **my_assert.c**

```
#include <stdio.h>
#include <e_acsl.h>

extern int __e_acsl_sound_verdict;

void __e_acsl_assert(int predicate, __e_acsl_assert_data_t * data) {
  printf("%s in file %s at line %d in function %s is %s (%s).\n\
The verified predicate was: '%s'.\n",
    data->kind,
    data->file,
    data->line,
    data->fct,
    predicate ? "valid" : "invalid",
    __e_acsl_sound_verdict ? "trustworthy" : "UNTRUSTWORTHY",
    data->pred_txt);
  if (data->values) {
    printf("With values:\n");
    __e_acsl_assert_data_value_t * value = data->values;
    while (value != NULL) {
      __e_acsl_print_value(value);
      value = value->next;
    }
  }
}
```

A monitored program that uses custom definition of `__e_acsl_assert` can then be generated as follows using `--external-assert` option of `e-acsl-gcc` to replace the default implementation of `__e_acsl_assert` with the customized one specified in `my_assert.c`.

```
$ e-acsl-gcc -X first.i -Ofirst --external-assert=my_assert.c
$ ./first.e-acsl
```

```
Assertion in file first.i at line 3 in function main is valid (trustworthy).
The verified predicate was: 'x == 0'.
Assertion in file first.i at line 4 in function main is invalid (trustworthy).
The verified predicate was: 'x == 1'.
```

The default assertion function prints the values contributing to an assertion along with the error message.

```
$ e-acsl-gcc -X first.i -Ofirst
$ ./first.e-acsl
first.i: In function 'main'
first.i:4: Error: Assertion failed:
      The failing predicate is:
      x == 1.
      With values:
      - x: 0
```

This behavior can be overridden by using the `--no-assert-print-data` option of `e-acsl-gcc`.

```
$ e-acsl-gcc -X first.i -Ofirst --no-assert-print-data
$ ./first.e-acsl
first.i: In function 'main'
first.i:4: Error: Assertion failed:
      The failing predicate is:
      x == 1.
```

As with the assertion function, `e-acsl-gcc` provides a way to use an alternative definition of `__e_acsl_print_value`.

Custom implementation of `__e_acsl_print_value` should be provided in a separate C (or object file) and respect the following signature:

```
void __e_acsl_print_value(__e_acsl_assert_data_value_t * value);
```

The details and documentation of the structure `__e_acsl_assert_data_value_t` can be found in the file `e_acsl_assert_data.h` distributed with E-ACSL in the installation folder `share/frama-c/e-acsl/instrumentation_model`.

Below is an example which prints only the name of the variable contributing to an assertion.

File **my_print_value.c**

```
#include <stdio.h>
#include <e_acsl.h>

void __e_acsl_print_value(__e_acsl_assert_data_value_t * value) {
  printf("\t- custom print for %s\n", value->name);
}
```

A monitored program that uses custom definition of `__e_acsl_print_value` can then be generated as follows using `--external-print-value` option of `e-acsl-gcc` to replace the default implementation of `__e_acsl_print_value` with the customized one specified in `my_print_value.c`.

```
$ e-acsl-gcc -X first.i -Ofirst \
   --external-print-value=my_print_value.c
$ ./first.e-acsl
first.i: In function 'main'
first.i:4: Error: Assertion failed:
      The failing predicate is:
      x == 1.
      With values:
      - custom print for x
```

The custom definitions of `__e_acsl_assert` and `__e_acsl_print_value` can be used at the same time.

```
$ e-acsl-gcc -X first.i -Ofirst \
   --external-assert=my_assert.c \
   --external-print-value=my_print_value.c
$ ./first.e-acsl
Assertion in file first.i at line 3 in function main is valid (trustworthy).
The verified predicate was: 'x == 0'.
With values:
      - custom print for x
Assertion in file first.i at line 4 in function main is invalid (trustworthy).
The verified predicate was: 'x == 1'.
With values:
      - custom print for x
```

2.3.6 Using C functions in annotations

Normally, annotations cannot contain calls to C functions. Thus, for any application of the form `a(...)` occurring inside an annotation, `a` is necessarily either a predicate or a logic function.

When invoking FRAMA-C with the flag `-e-acsl-builtins=<s>` the C functions in the comma-separated sequence `<s>` are added to the execution environment as logic functions. In this manner parts of the specification can be expressed in C. Similarly to JML pure methods, these functions should be side-effect free.

File **builtins.c**

```
int incr(int x) {
  return x + 1;
}

/*@ ensures \result == incr(i); */
int f(int i) {
  return ++i;
}
```

This code is parsable only if the command-line flag `-e-acsl-builtins=incr` is supplied.

2.4 Additional Verifications

In addition to checking annotations at runtime, E-ACSL is able to detect a few errors at runtime.

2.4.1 Format String Vulnerabilities

Whenever option `--validate-format-strings` of `e-acsl-gcc` is set, E-ACSL detects format-string vulnerabilities in `printf`-like function. Below is an example which incorrectly tries to print a string through a `%d` format.

File **printf.c**

```
#include <stdio.h>

int main(void) {
  printf("is %d really an int?", "foo");
  return 0;
}
```

This error can be detected by E-ACSL as follows.

```
$ e-acsl-gcc -Oprintf -c --validate-format-strings printf.c
$ ./printf.e-acsl
printf: directive 1 ('%d') expects argument of type 'int' but the corresponding
argument has type 'char*'
Abort
```

2.4.2 Incorrect Usage of Libc Functions

Whenever option `--libc-replacements` of `e-acsl-gcc` is set (or equivalently: the option `-e-acsl-replace-libc-functions` of Frama-C), E-ACSL is able to detect incorrect usage of the following libc functions:

- `memcmp`
- `memcpy`
- `memmove`
- `memset`
- `strcat`
- `strncat`
- `strcmp`
- `strncmp`
- `strcpy`
- `strncpy`
- `strlen`

For instance, the program below incorrectly uses `strcpy` because the destination string should contain at least 7 `char` (and not only 6).

File **strcpy.c**

```c
#include <stdlib.h>
#include <string.h>

int main(void) {
  char *dst = malloc(sizeof(char) * 6);
  strcpy(dst, "foobar");
  return 0;
}
```

This error can be reported by E-ACSL as follows.

```
$ e-acsl-gcc -Ostrcpy -c --libc-replacements strcpy.c
$ ./strcpy.e-acsl
strlen: insufficient space in destination string, at least 7 bytes required
Abort
```

2.5 Incomplete Verification

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 (Section 2.5.1) or in which some functions remain undefined (Section 2.5.2).

It is also possible to partially instrument an application in order to reduce the runtime overhead.

2.5.1 Programs with no Main Function

The E-ACSL plug-in can instrument a program without the `main` function. Consider for instance the following annotated program without `main`.

File **no_main.i**

```
/*@ behavior even:
  @   assumes n % 2 == 0;
  @   ensures \result >= 1;
  @ behavior odd:
  @   assumes n % 2 != 0;
  @   ensures \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.

```
$ e-acsl-gcc no_main.i -omonitored_no_main.i
[kernel] Parsing no_main.i (no preprocessing)
[e-acsl] beginning translation.
[kernel] Parsing FRAMAC_SHARE/e-acsl/e_acsl.h (with preprocessing)
[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 could be incorrect if the program contains memory-related annotations (see Section 3.3.1). 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 link it against additional files such as the following:

File **main.c**

```
#include "stdio.h"

int main(void) {
  unsigned long long x = my_pow(2, 16);
  printf("x = %llu\n",x);
  return 0;
}
```

```
$ e-acsl-gcc -C monitored_no_main.i main.c -Owith_main
$ ./with_main.e-acsl
x = 65536
```

2.5.2 Undefined Functions

Similar to the above section E-ACSL is capable of instrumenting a program which has definitions of arbitrary functions missing. 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-gcc -omonitored_no_code.i no_code.c
[kernel] Parsing no_code.c (with preprocessing)
[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] Parsing FRAMAC_SHARE/e-acsl/e_acsl.h (with preprocessing)
[e-acsl] translation done in project "e-acsl".
```

Similar to the previous Section the warning indicates that the instrumentation could be incorrect if the program contains memory-related annotations (see Section 3.3.2). That is still not the case here, so you can safely ignore it.

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;
}
```

Similar to the example shown in the previous section you can generate the executable by providing definition of `my_pow`.

```
$ e-acsl-gcc -C -Ono_code pow.i monitored_no_code.i
$ ./no_code.e-acsl
no_code.c: In function 'my_pow'
no_code.c:5: Error: Postcondition failed:
      The failing predicate is:
      even: \result >= 1.
Aborted (core dumped)
```

The execution of the corresponding binary fails at runtime: actually, our implementation of the function `my_pow` overflows in case of large exponentiations.

2.5.3 Providing a White List of Functions

By default, the E-ACSL plug-in generates code for checking at runtime all the annotations of the analyzed files. Yet, it is possible to handle only annotations of a given set of functions through the

option `-e-acsl-functions`. This way, no runtime check is generated for annotations of all the other functions. It allows the user to focus on a particular part of the code, while reducing the global runtime overhead.

2.5.4 Partial Instrumentation (Experimental)

Partial instrumentation is still an experimental feature. It may evolve in the future and/or not work as expected.

By default, the E-ACSL plug-in generates all the necessary pieces of code for checking at runtime the provided annotations. This amount of instrumentation may be quite large. It is possible to reduce it by manually specifying the set of functions to be instrumented through the option `-e-acsl-instrument`.

This way, contrary to the option `-e-acsl-functions`, all the annotations are still checked at runtime. However, since less code is generated, the generated program usually runs faster. However, it may lead to unsound verdicts if it would have been necessary to generate the instrumentation for one of the uninstrumented functions.

A typical usage of this option consists in specifying the set of functions to be *not* instrumented. For instance, to instrument all functions except functions `f` and `g`, you should specify `-e-acsl-instrument=+@all,-f,-g`.

Consider the following example in which all assertions are valid.

File **partial.i**

```
void f(int *p) {
  *p = 0;
}

void g(int *p) {
  *p = 1;
}

int main(void) {
  int t[3];
  f(&t[0]);
  /*@ assert \initialized(&t[0]); */
  t[1] = 1;
  g(&t[1]);
  g(&t[2]);
  /*@ assert \initialized(&t[1]); */
  /*@ assert \initialized(&t[2]); */
  return 0;
}
```

One can ask to not instrument function `g` (*i.e.* to instrument only functions `f` and `main`) through the following command.

```
$ e-acsl-gcc \
  -c --oexec-e-acsl partial.out \
  --e-acsl-extra="-e-acsl-instrument=+@all,-g" \
  partial.i
```

Therefore, running the generated executable `partial.out` leads to the following verdicts.

```
$ ./partial.out
partial.i: In function 'main'
partial.i:16: Warning: no sound verdict for Assertion (guess: ok).
      the considered predicate is:
      \initialized(&t[1])
```

```
partial.i: In function 'main'
partial.i:17: Warning: no sound verdict for Assertion (guess: FAIL).
      the considered predicate is:
      \initialized(&t[2])
```

First, there is no message for the first assertion about the initialization of `&t[0]`, meaning that this assertion is silently checked as valid by E-ACSL (see Section 2.3.5). It was made possible because of the complete instrumentation of functions `f` and `main`.

Second, since function `g` was not instrumented, no sound verdict may be provided after executing it. It leads to the printed warnings for the last two assertions. The indicated guesses (`ok` for the first assertion and `FAIL` for the second one) are the verdicts computed by E-ACSL. They can be trusted if and only if the E-ACSL instrumentation is not necessary for validating the considered annotations. In that particular case, it is not true. For instance, the second guess is incorrect since `&t[2]` is actually initialized through function `g`. E-ACSL missed it since this function is not instrumented..

2.6 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-last -print
  -ocode monitored_combine.c
$ e-acsl-gcc -C -Ocombine monitored_combine.c
$ ./combine.e-acsl
```

Automatic assertion generation can also be enabled via `e-acsl-gcc` directly via the following command:

```
$ e-acsl-gcc -Ocombine -omonitored_combine.i --rte=all combine.i
```

Note the use of `e-acsl-gcc --rte` option which asks Frama-C to combine E-ACSL with RTE plug-in. This plug-in automatically instruments the input program with runtime assertions before running the E-ACSL plug-in on it. `--rte` option of `e-acsl-gcc` also accepts a comma-separated list of arguments indicating the types of assertions to generate. Consult the `e-acsl-gcc` man page for a detailed list of arguments accepted by `--rte`.

The present implementation of `e-acsl-gcc` does not fully support combining E-ACSL with other Frama-C plug-ins. However it is still possible to instrument programs with E-ACSL directly and use `e-acsl-gcc` to compile the generated code.

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-O-valid`. For instance, Eva [5] 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 -rte combine.i -then -eva -then -e-acsl \
  -then-last -print -ocode monitored_combine.i
```

The additional code will be generated with the two following command.

```
$ frama-c -rte combine.i -then -eva -then -e-acsl \
  -e-acsl-O-valid -then-last -print -ocode monitored_combine.i
```

2.7 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` option of the E-ACSL plug-in. While there is not direct support for this option in `e-acsl-gcc` you can pass it using `-F` switch:

File **check.i**

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

```
$ e-acsl-gcc -F"-e-acsl-project foobar" check.i
[kernel] Parsing check.i (no preprocessing)
[e-acsl] beginning translation.
[kernel] Parsing FRAMAC_SHARE/e-acsl/e_acsl.h (with preprocessing)
[e-acsl] translation done in project "foobar".
```

2.8 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.

2.8.1 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`	terms- and expressions-related information

2.8.2 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 2.3.4)
preparation	duplication of functions with contracts (section 2.5.2)
translation	translation of an annotation into C code
typing	minimization of the instrumentation for integers (section 2.3.3)

Known Limitations 3

The development of the E-ACSL plug-in is still ongoing. First, the E-ACSL reference manual [15] is not yet fully supported. Which annotations can already be translated into C code and which cannot is defined in a separate document [16]. 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[1] (see Chapter 10 of the Frama-C User Manual [7]). 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[2].

3.1 Supported Systems

The only well-supported system is a Linux distribution on a 64-bit architecture.

3.1.1 Operating Systems

Non-Linux systems are almost not yet experimented. It might work but there are most probably issues, in particular if using a non-standard libc. For instance, there are known bugs under Mac OS X[3].

3.1.2 Architectures and Runtime Library

The segment-based memory model (used by default for monitoring memory properties such as `\valid`) assumes little-endian architecture and has very limited support for 32-bit architectures. When using a 32-bit machine or big-endians, we recommend using the bittree-based memory model instead.

The runtime library is also not *thread-safe.*

3.2 Uninitialized Values

As explained in Section 2.3.1, 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**

[1] https://git.frama-c.com/pub/frama-c/-/issues

[2] Read https://git.frama-c.com/pub/frama-c/blob/master/CONTRIBUTING.md for additional details.

[3] See for instance at https://bts.frama-c.com/view.php?id=2369

```
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 undefined because the assertion reads the uninitialized variable `x`. You should be caught by the E-ACSL plug-in, but that is not the case yet.

```
$ e-acsl-gcc uninitialized.i -Omonitored_uninitialized
monitored_uninitialized.i: In function 'main':
monitored_uninitialized.i:44:16: warning: 'x' is used uninitialized in this function
[-Wuninitialized]
$ ./monitored_uninitialized.e-acsl
```

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` [15].

3.3 Incomplete Programs

Section 2.5 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 2.5.1 and 2.5.2).

3.3.1 Programs without Main

The instrumentation in the generated program is partial for every program without main containing memory-related annotations, except if the option `-e-acsl-full-mtracking` or the E-ACSL plug-in (of `-M` option of `e-acsl-gcc`) 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.

```
$ e-acsl-gcc -n -M -omonitored_valid_no_main.i valid_no_main.c
[kernel] Parsing valid_no_main.c (with preprocessing)
[e-acsl] beginning translation.
[kernel] Parsing FRAMAC_SHARE/e-acsl/e_acsl.h (with preprocessing)
[kernel] Warning: no entry point specified:
  you must call functions '__e_acsl_globals_init', '__e_acsl_globals_clean',
  '__e_acsl_memory_init' and '__e_acsl_memory_clean' by yourself.
[e-acsl] translation done in project "e-acsl".
```

The last warning states an important point: if this program is linked against another file containing `main` function, then this main function must be modified to insert a calls to the functions `__e_acsl_globals_init` and `__e_acsl_memory_init` at the very beginning. These functions play a very important role: the latter initializes metadata storage used for tracking of memory blocks while the former initializes tracking of global variables and constants. Unless these calls are inserted the run of a modified program is likely to fail.

While it is possible to add such instrumentation manually we recommend using `e-acsl-gcc`. Consider the following incomplete program containing `main`:

File **modified_main.c**

```
int main(int argc, char **argv) {
  f();
  return 0;
}
```

Then just compile and run it as explained in Section 2.3.4.

```
$ e-acsl-gcc -n -M -omonitored_modified_main.i modified_main.c
$ e-acsl-gcc -C -Ovalid_no_main monitored_modified_main.i monitored_valid_no_main.i
$ ./valid_no_main.e-acsl
Assertion failed at line 11 in function f.
The failing predicate is:
freed: \valid(x).
Aborted
```

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[4].

3.3.2 Undefined Functions

The instrumentation in the generated program is partial for a program p 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.

3.3.3 Incomplete Types

The instrumentation in the generated program is partial for a program p if p contains a memory-related annotation a and a variable v with an incomplete type definition such that a depends on v (even indirectly).

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

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

3.4 Recursive Functions

Programs containing recursive functions have the same limitations as the ones containing undefined functions (Section 3.3.2) and memory-related annotations.

3.5 Variadic Functions

Programs containing undefined variadic functions with contracts are not yet supported. Using the `-variadic-translation` option of Frama-C could be a solution in some cases, but its generated code cannot always be compiled.

3.6 Function Pointers

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

3.7 Requirements to Input Programs

3.7.1 E-ACSL Namespace

While E-ACSL uses source-to-source transformations and not binary instrumentations it is important that the source code provided as input does not contain any variables or functions prefixed `__e_acsl_`. E-ACSL reserves this namespace for its transformations, and therefore an input program containing such symbols beforehand may fail to be instrumented or compiled.

3.7.2 Memory Management Functions

Programs providing custom definitions of `syscall`, `mmap` or `sbrk` should be rejected. Also, an input programs should not modify memory-management functions namely `malloc`, `calloc`, `realloc`, `free`, `cfree`, `posix_memalign` and `aligned_alloc`. E-ACSL relies on these functions in order to track heap memory. Further, correct heap memory monitoring requires to limit allocation and deallocation of heap memory to POSIX-compliant memory management functions listed above. Monitoring of programs that allocate memory using non-standard or obsolete functions (e.g., `valloc`, `memalign`, `pvalloc`) may not work correctly.

Changes A

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

E-ACSL 34.0 Selenium

- **Optimization level** : Document new option `-e-acsl-O` that configures the level of optimization for the resulting code instrumented by E-ACSL.
- Renamed command-line flags `-e-acsl-valid` to `-e-acsl-O-valid`, `-e-acsl-assert-print-data` to `-e-acsl-O-print-values`, and `-e-acsl-gmp-only` to `-e-acsl-O-gmp-only`.

E-ACSL 33.0 Arsenic

- Extended section **Integers** with more details, and a description of widening command-line flags.
- Added section **Using C functions in annotations** documenting the `-e-acsl-builtins` command-line flag.
- Added documentation of the `-e-acsl-replace-libc-functions` command-line flag to the section **Incorrect Usage of Libc Functions**

E-ACSL 25.0 Manganese

- New section **Concurrency Support**.

E-ACSL 24.0 Chromium

- **Runtime Monitor Behavior**: Document new options `--assert-print-data` and `--external-print-value` of `e-acsl-gcc`, and update documentation for option `--external-assert`.

E-ACSL 22.0 Titanium

- Update every section with changes to Frama-C and `e-acsl-gcc` output
- **Simple Example**: Remove option `-e-acsl-check`
- **Combining E-ACSL with Other PLug-ins**: `-e-acsl-prepare` is no more necessary.

E-ACSL 19.0 Potassium

- **Runtime Monitor Behavior**: Document global variable `__e_acsl_sound_verdict` usable in `__e_acsl_assert`.
- New section **Partial Instrumentation**.
- New section **Providing a White List of Functions**.

E-ACSL 18.0 Argon

- **Introduction**: Improve a bit and reference new related papers.
- **What the Plug-in Provides**: Highlight that the memory analysis is not yet robust.
- **What the Plug-in Provides**: Highlight the importance of **-e-acsl-prepare**.
- **Known Limitations**: Replace section "Limitations of E-ACSL Monitoring Libraries" by the new section "Supported Systems".
- **Known Limitations**: Add limitation about monitoring of variables with incomplete types.

E-ACSL Chlorine-20180501

- New section **Additional Verifications**.
- Update every section with respect to the changes introduced since E-ACSL Sulfur-20180101.

E-ACSL Sulfur-20180101

- no changes

E-ACSL Phosphorus-20170501

- Removed chapter **Easy Instrumentation with E-ACSL**.
- **What the Plug-in Provides**: added section **E-ACSL Wrapper Script**.
- **What the Plug-in Provides**: added description of E-ACSL runtime libraries.
- **Known Limitations**: added section **Requirements to Input Programs**.
- **Known Limitations**: added section **Limitations of E-ACSL Monitoring Libraries**.
- **Recursive Functions**: remove limitation about possible uses before being declared.
- **Variadic Functions**: mention the Variadic plug-in.

E-ACSL 0.6

– **Easy Instrumentation with E-ACSL**: new chapter.

E-ACSL 0.5

– **Libc**: new section.
– **Architecture Dependent Annotations**: add a remark about Gcc's machdep.
– Use `-then-last` whenever possible.
– Update output wrt Frama-C Sodium changes.
– **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, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The Dogged Pursuit of Bug-Free C Programs: The Frama-C Software Analysis Platform. *Communications of the ACM*, August 2021.

[2] Patrick Baudin, François Bobot, Loïc Correnson, and Zaynah Dargaye. *Frama-C's WP plug-in.* http://frama-c.com/download/frama-c-wp-manual.pdf.

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

[4] Thibaut Benjamin and Julien Signoles. Abstract Interpretation of Recursive Logic Definitions for Efficient Runtime Assertion Checking. In *International Conference on Tests and Proofs (TAP)*, July 2023. **Best Paper Award**.

[5] David Bühler, Pascal Cuoq, Boris Yakobowski, Matthieu Lemerre, André Maroneze, Valentin Perelle, and Virgile Prevosto. *EVA – The Evolved Value Analysis plug-in.* https://frama-c.com/download/frama-c-value-analysis.pdf.

[6] 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.

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

[10] 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.

[11] Nikolai Kosmatov, Fonenantsoa Maurica, and Julien Signoles. Efficient Runtime Assertion Checking for Properties over Mathematical Numbers. In *International Conference on Runtime Verification (RV)*, October 2020.

[12] Nikolaï Kosmatov, Guillaume Petiot, and Julien Signoles. An optimized memory monitoring for runtime assertion checking of C programs. In *International Conference on Runtime Verification (RV 2013)*, volume 8174 of *LNCS*, pages 167–182. Springer, September 2013.

[13] 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.

[14] Dara Ly, Nikolai Kosmatov, Frédéric Loulergue, and Julien Signoles. Soundness of a dataflow analysis for memory monitoring. In *Workshop on Languages and Tools for Ensuring Cyber-Resilience in Critical Software-Intensive Systems (HILT)*, November 2018.

[15] Julien Signoles. *E-ACSL: Executable ANSI/ISO C Specification Language.* http://frama-c.com/download/e-acsl/e-acsl.pdf.

[16] Julien Signoles. *E-ACSL. Implementation in Frama-C Plug-in E-ACSL.* http://frama-c.com/download/e-acsl/e-acsl-implementation.pdf.

[17] Julien Signoles. From Static Analysis to Runtime Verification with Frama-C and E-ACSL, July 2018. Habilitation Thesis.

[18] Julien Signoles, Nikolai Kosmatov, and Kostyantyn Vorobyov. E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs. Tool Paper. In *International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES)*, September 2017.

[19] Kostyantyn Vorobyov, Julien Signoles, and Nikolai Kosmatov. Shadow state encoding for efficient monitoring of block-level properties. In *International Symposium on Memory Management (ISMM)*, pages 47–58. ACM, June 2017.

Index

-omonitored_first.i, 9

ACSL, 5, 8, 12
--assert-print-data, 31

-C, 9
--concurrency, 11

-d, 10
-debug, 10
dlmalloc, 15

-E, 10
-e, 10
-e-acsl, 6, 7
--e-acsl-concurrency, 11
-e-acsl-full-mtracking, 16, 28
-e-acsl-functions, 23
-e-acsl-instrument, 23
-e-acsl-msg-key, 26
-e-acsl-msg-key-unset, 26
-e-acsl-O-gmp-only, 14
-e-acsl-O-valid, 25
-e-acsl-project, 25
-e-acsl-verbose, 25
-e-acsl-verbose 0, 26
-e-acsl-verbose 1, 26
-e-acsl-verbose 2, 26
-e-acsl-verbose 3, 26
-e-acsl-verbose 4, 26
e_acsl_assert, 8, ***16***, ***17***, ***19***
__e_acsl_globals_init, 29
__e_acsl_memory_init, 29
e_acsl_print_value, ***18***, ***19***
e_acsl_sound_verdict, ***17***
Eva, 5, 25
--external-assert, 17, 31
--external-print-value, 18, 31

-F, 10, 25
--fail-with-code=CODE, 16
Format String Vulnerabilities, 19
-framac-stdlib, 13
Function
  Input, 30
  Not Instrumented, 23
  Pointer, 30
  Recursive, 30
  Undefined, 21, 29
  Variadic, 30
  White List, 22

-G, 10
-g, 14
Gcc, 9
GMP, 13, 14, 24

-h, 12

-I, 10
Installation, 5
integer, 13

JML, 19

--keep-going, 17

-L, 13
-l, 10
Libc, 13, 20, 27
--libc-replacements, 20

-M, 16, 28
-m, 16
-m32, 11
-m64, 11
Mac OS X, 27
-machdep, 8, 10, 11
-machdep gcc_x86_32, 11
-machdep gcc_x86_64, 11
memcmp, 20
memcpy, 20
memmove, 20
memset, 20

-n, 9
--no-assert-print-data, 18
-no-framac-stdlib, 13

-O, 9
-o, 9
-ocode, 8

-print, 7
Program
    Without main, 21, 28
-pthread, 11

-q, 10

RTE, 5, 24
--rte, 24
Runtime Error, 12

-s, 10
strcat, 20
strcmp, 20
strcpy, 20
strlen, 20
strncat, 20
strncmp, 20
strncpy, 20

-then, 24
-then-last, 7
Type
    Incomplete, 29

Uninitialized value, 27

-v, 10
--validate-format-strings, 19
Variadic, 32
-variadic-translation, 30
-verbose, 10

WP, 5
Wp, 5