E-ACSL User Manual

E-ACSL Plug-in

Release Phosphorus-20170501

Julien Signoles and Kostyantyn Vorobyov

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

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 Phosphorus-20170501 (January 2017) compatible with Phosphorus-20170501 version of FRAMA-C [5, 10]. The development of the E-ACSL plug-in is still ongoing. Features described by this document may evolve in the future.

Acknowledgements

---

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

Chapter 1

Introduction

FRAMA-C [5, 10] is a modular analysis framework for the C programming language which supports the ACSL specification language [2]. This manual documents the E-ACSL plug-in of FRAMA-C, version Phosphorus-20170501. The E-ACSL version you are using is indicated by the command `frama-c -e-acsl-version`. 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" [4][1]. This is the primary goal of E-ACSL. 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 VALUE [3] or WP [1].

Annotations used by the plug-in must be written in the E-ACSL specification language [12, 6] – 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. Annotation supported by the plugin are described in a separate document [13].

This manual does *not* explain how to install the E-ACSL plug-in. For installation instructions please refer to the `INSTALL` file in the E-ACSL 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.

What the Plug-in Provides

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.sh` – 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 2.4 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 2.5 explains how to combine the E-ACSL plug-in with other FRAMA-C plug-ins. Finally, Section 2.6 introduces how to customize the plug-in, and Section 2.7 details the verbosing policy of the plug-in.

2.1   Simple Example

This section is a mini-tutorial which explains from scratch how to 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 FRAMAC_SHARE/libc/__fc_builtin_for_normalization.i (no preprocessing)
[kernel] Parsing FRAMAC_SHARE/e-acsl/e_acsl.h (with preprocessing)
[kernel] Parsing FRAMAC_SHARE/e-acsl/e_acsl_gmp.h (with preprocessing)
[kernel] Parsing FRAMAC_SHARE/e-acsl/e_acsl_mmodel_api.h (with preprocessing)
[kernel] Parsing first.i (no preprocessing)
[e-acsl] beginning translation.
<skip a warning when translating the Frama-C libc>
[e-acsl] translation done in project "e-acsl".
```

Even though `first.i` has already been pre-processed, 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 `-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 [5]:

```
$ frama-c -e-acsl first.i -then-last -print
[kernel] Parsing FRAMAC_SHARE/libc/__fc_builtin_for_normalization.i (no preprocessing)
[kernel] Parsing FRAMAC_SHARE/e-acsl/e_acsl.h (with preprocessing)
[kernel] Parsing FRAMAC_SHARE/e-acsl/e_acsl_gmp.h (with preprocessing)
[kernel] Parsing FRAMAC_SHARE/e-acsl/e_acsl_mmodel_api.h (with preprocessing)
[kernel] Parsing first.i (no preprocessing)
[e-acsl] beginning translation.
[e-acsl] translation done in project "e-acsl".
/* Generated by Frama-C */
typedef unsigned int size_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];
/*@ requires pred != 0;
    assigns \nothing; */
extern  __attribute__((__FC_BUILTIN__)) void __e_acsl_assert(int pred,
                                                             char *kind,
                                                             char *fct,
                                                             char *pred_txt,
                                                             int line);

/*@ ghost extern int __e_acsl_init; */

/*@ ghost extern int __e_acsl_internal_heap; */

extern size_t __e_acsl_heap_allocation_size;

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

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

  }
 */
/*@
predicate diffSize{L1, L2}(integer i) =
  \at(__e_acsl_heap_allocation_size,L1) -
  \at(__e_acsl_heap_allocation_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;
  return __retres;
}
```

---

[1]The notion of *project* is explained in Section 8.1 of the FRAMA-C user manual [5].

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. After each E-ACSL annotation, a call to `__e_acsl_assert` 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);
```

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 [5] 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.i
```

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 int`, whereas in 64-bit architectures `size_t` is typically defined as `unsigned long`. 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_32` (a generic 32-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 64-bit machine you should also pass `-machdep x86_64` option as follows:

```
$ frama-c -machdep x86_64 -e-acsl first.i -then-last \
  -print -ocode monitored_first.i
```

An executable checking the validity of annotations in `first.i` at runtime can be generated using a C compiler (for instance GCC), as follows:

```
$ gcc monitored_first.i -omonitored_first \
  -leacsl-rtl-bittree -leacsl-gmp -leacsl-jemalloc -lpthread -lm
monitored_first.i:9:1: warning: '__FC_BUILTIN__' attribute directive ignored [-Wattributes]
 typedef __e_acsl_mpz_struct ( __attribute__((__FC_BUILTIN__)) __e_acsl_mpz_t)[1];
 ^
monitored_first.i:16:62: warning: '__FC_BUILTIN__' attribute directive ignored [-Wattributes]
                                                              int line);
                                                                  ^
monitored_first.i:24:1: warning: '__FRAMA_C_MODEL__' attribute directive ignored [-Wattributes]
 int __fc_random_counter __attribute__((__unused__, __FRAMA_C_MODEL__));
```

The displayed warnings can be safely ignored as they refer to the attribute `FC_BUILTIN` used internally by FRAMA-C. The warnings can be suppressed using `-Wno-attributes` GCC option. You may also notice that the generated executable (`monitored_first`) is linked against the E-ACSL static library (i.e., via `-leacsl-rtl-bittree`) installed alongside the E-ACSL plug-in. This library provides definitions of the functions used by E-ACSL during its runtime analysis. The monitored executable is also linked against the customized versions of the JEMALLOC

memory allocator (`-leacsl-jemalloc`) and the GMP library (`-leacsl-gmp`) used by E-ACSL. Even though in this example they are not strictly needed (since the transformed executable neither tracks memory nor reasons about arbitrary precision integers) we show all libraries potentially used by a transformed executable for completeness purposes. We describe the libraries used by E-ACSL later on in Section 2.3.4.

Finally you can execute the generated binary.

```
$ ./monitored_first
Assertion failed at line 4 in function main.
The failing predicate is:
x == 1.
Aborted
$ echo $?
134
```

The execution aborts with a non-zero exit code (134) and an error message indicating 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.

2.2 E-ACSL Wrapper Script

You may have noticed that generating executables using a combination of FRAMA-C, E-ACSL and GCC may be tedious. To simplify this process we developed `e-acsl-gcc.sh` – a convenience wrapper around FRAMA-C and GCC allowing to instrument and compile instrumented code using a single command and a few parameters.

Presently, `e-acsl-gcc.sh` 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.sh` as its main driver for source code instrumentation and compilation of the instrumented programs.

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

2.2.1 Standard Usage

The default behaviour of `e-acsl-gcc.sh` is to instrument an annotated C program with runtime assertions via a run of FRAMA-C.

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

The above command 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`.

Compilation of the original and the instrumented sources is enabled using the `-c` option. For instance, command

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

instruments the code in `first.i`, outputs it to `monitored_first.i` and produces 2 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.

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.sh -c -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.sh -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.sh -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.sh` 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.sh -I/usr/local/bin/frama-c -G/usr/bin/gcc6 foo.c
```

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

```
$ e-acsl-gcc.sh -c -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.sh` 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 pre-processor (see Section 2.2.5).

2.2.3 Verbosity Levels and Output Redirection

By default `e-acsl-gcc.sh` 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.sh -v5 -F"-kernel-verbose 0" foo.c
```

Verbosity of the `e-acsl-gcc.sh` 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.sh` itself.

The output of `e-acsl-gcc.sh` 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.sh -c -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 32-bit platform, but it can be customized through `-machdep` switch [5]. 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.sh` 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 Documentation

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

```
$ man e-acsl-gcc.sh
```

A brief listing of the basic options of `e-acsl-gcc.sh` can be viewed using its `-h` option:

```
$ e-acsl-gcc.sh -h
e-acsl-gcc.sh - instrument and compile C files with E-ACSL
Usage: e-acsl-gcc.sh [options] files
Options:
  -h          show this help page
  -c          compile instrumented code
  -l          pass additional options to the linker
  -e          pass additional options to the prepreprocessor
  -E          pass additional arguments to the Frama-C preprocessor
  -p          output the generated code to STDOUT
  -o <file>   output the generated code to <file> [a.out.frama.c]
  -O <file>   output the generated executables to <file> [a.out, a.out.e-acsl]
  -M          maximize memory-related instrumentation
  -g          always use GMP integers instead of C integral types
  -q          suppress any output except for errors and warnings
  -s <file>   redirect all output to <file>
  -I <file>   specify Frama-C executable [frama-c]
  -G <file>   specify C compiler executable [gcc]
```

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 [2] 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 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"* [12].

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.sh -c -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.

It 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 undefine behaviors matters).

Notably, `e-acsl-gcc.sh` 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.sh` you can pass `-L` option to the wrapper script.

2.3.3 Integers

E-ACSL uses an `integer` type corresponding to mathematical integers which 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 [6, 8], it may generate such integers from time to time. It 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.sh -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.sh` which passes the `-e-acsl-gmp-only` option to the E-ACSL plug-in. However it could slow down the execution of the instrumented program significantly.

---

[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**

```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.sh -c -Ovalid valid.c
$ ./valid.e-acsl
Assertion failed at line 11 in function main.
The failing predicate is:
freed: \valid(x).
Aborted
```

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 [11], whereas segment-based memory model [3] is based on shadow memory [14].

Each of the supported memory models is implemented as a separate static library installed along with E-ACSL plug-in itself. This is such that `libeacsl-rtl-bittree.a` implements

---

[3]Segment-based model of E-ACSL has not yet appeared in the literature.

memory tracking using Patricia-trie and `libeacsl-rtl-segment.a` implements shadow-based tracking. The functionality of the provided memory models (and therefore libraries implementing them) is equivalent, however they differ in performance. We further discuss performance considerations.

Both libraries use the same API and differ only in implementation, therefore it is safe to link an E-ACSL-instrumented program against any of these libraries. For instance, in Section 2.1.2 we have linked the instrumented program in `monitored_first.i` against `libeacsl-rtl-bittree.a`, but you can also link it against the library implementing the segment-based memory model via `-leacsl-rtl-segment`.

The E-ACSL libraries allocate heap memory using a customized version of the JEMALLOC[4] memory allocator replacing system-wide `malloc` and similar functions (e.g., `calloc` or `free`). Mathematical integers used by E-ACSL are implemented using a customized version of the GMP library. Both, JEMALLOC and GMP should be added at link time as shown in Section 2.1.2. We, however, recommend using `e-acsl-gcc.sh` who adds necessary compile- and link-time dependencies and allows for easy switching between different memory models as discussed below.

At the level of `e-acsl-gcc.sh` 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.sh -mbittree -c -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.sh -msegment -c -Ovalid valid.c
```

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

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

In this case, `e-acsl-gcc.sh` 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 bittree-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 [14].

Maximized Memory Tracking

By default E-ACSL uses static analysis to reduce instrumentation and therefore runtime and memory overheads [9]. 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

---

[4]http://jemalloc.net/

when it is not required. This feature can be enabled using the `-M` switch of `e-acsl-gcc.sh` or `-e-acsl-full-mmodel` option of the E-ACSL plug-in.

2.3.5 Debug Libraries

E-ACSL runtime libraries mentioned in Section 2.3.4 (i.e., `libeacsl-rtl-bittree.a` and `libeacsl-rtl-segment.a`) are optimized for performance (via `-O2` compiler flag). An E-ACSL installation also provides unoptimized versions of these libraries (suffixed `-dbg.a`) aimed at being used during debugging.

The main difference between unoptimized (debug) and optimized (production) libraries is that debug libraries include assertions validating the correctness of the RTL itself. For instance, during memory tracking such assertions ensure that during a run of a program all tracked memory blocks are disjoint.

Consider the following program that violates an annotation in function `foo` via a call `bar(p+1)` in the main function:

File **rte_debug.i**

```
void foo(int *p) {
  /*@assert \valid(p); */
  *p = 1;
}

void bar(int *p) {
  foo(p);
}

int main(void) {
  int i = 0;
  int *p = &i;
  bar(p+1);
  return 0;
}
```

A monitored executable linked against a debug version of E-ACSL RTL can ge generated using `--rt-debug` flag of `e-acsl-gcc.sh` as follows:

```
$ e-acsl-gcc.sh --rt-debug -c -omonitored_rte_debug.c -Orte_debug rte_debug.i
```

A run of the monitored executable shows violation of the E-ACSL annotation.

```
$ rte_debug.e-acsl
Assertion failed at line 2 in function foo.
The failing predicate is:
\valid(p).
/** Backtrace *************************/
trace at e_acsl_trace.h:45
 - exec_abort at e_acsl_assert.h:58
 - runtime_assert at e_acsl_assert.h:93
 - foo at monitored_rte_debug.c:92
 - bar at monitored_rte_debug.c:102
 - main at monitored_rte_debug.c:119
/*************************************/
Aborted
```

A run of an executable linked against an unoptimized RTL shows a stack trace on the instrumented program saved to `monitored_rte_debug.c`. In the example above calls to `exec_abort` and `runtime_assert` belong to the E-ACSL RTL, whereas calls to `foo`, `bar` and `main` belong to the input program.

It should be noted that because debug versions of E-ACSL RTL are compiled without compile-time optimizations and execute additional assertions the runtime overheads of the

instrumented programs linked against the debug versions of E-ACSL RTL can be up to 10 times greater than those linked against the production versions.

2.3.6   Runtime Monitor Behavior

When a predicate is checked at runtime, the function `__e_acsl_assert` is called. By default, it does nothing if the predicate is valid, and prints an error message and aborts the execution if the predicate is invalid.

`__e_acsl_assert` function is defined as a weak symbol in the E-ACSL library, therefore it is possible to modify this behavior by providing an alternative definition of `__e_acsl_assert`. You must only respect the following signature:

```
void __e_acsl_assert(int pred, char *kind,
                     char *func_name, char *pred_text, int line);
```

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

Then you can generate the program as usual, (we reuse the initial example `first.i` of this manual).

```
$ e-acsl-gcc.sh -c first.i my_assert.c -Ocustomized_first
$ ./customized_first.e-acsl
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'.
```

2.4   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.

2.4.1   Programs without Main

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.sh no_main.i -omonitored_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 could 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 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.sh -C monitored_no_main.i main.c -Owith_main
$ ./with_main.e-acsl
x = 65536
```

2.4.2 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.sh -omonitored_no_code.i no_code.c
[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.
no_code.c:9:[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".
```

Similar to the previous Section the warning indicates that the instrumentation could 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.

File **pow.i**

```c
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.sh -C -Ono_code pow.i monitored_no_code.i
$ ./no_code.e-acsl
Postcondition failed at line 5 in function my_pow.
The failing predicate is:
\old(n % 2 == 0) ==> \result >= 1.
Aborted
```

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 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 [7] 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**

```c
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.i
$ e-acsl-gcc.sh -C -Ocombine  monitored_combine.i
$ ./ combine.
```

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

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

Note the use of `e-acsl-gcc.sh --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.sh` also accepts a comma-separated list of arguments indicating the types of assertions to generate. Consult the `e-acsl-gcc.sh` man page for a detailed list of arguments accepted by `--rte`.

The present implementation of `e-acsl-gcc.sh` 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.sh` to compile the generated code.

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 [3] 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-last -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-last -print -ocode monitored_combine.i
$ frama -c -rte combine.i -then -val -then -e-acsl \
        -then-last -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.

2.6   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.sh` 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.sh -F"-e-acsl-project foobar" check.i
<skip preprocessing commands>
[e-acsl] beginning translation.
check.i:4:[e-acsl] warning: E-ACSL construct 'right shift' is not yet supported.
  Ignoring annotation.
[e-acsl] translation done in project "foobar".
```

Further, the E-ACSL plug-in option `-e-acsl-check` does not generate a new project but verifies that each annotation is translatable. Then it produces a summary as shown in the following example (left or right shifts in annotations are not yet supported by the E-ACSL plug-in [13]).

```
$ 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.
```

2.7   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.

2.7.1   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

2.7.2 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)
duplication	duplication of functions with contracts (section 2.4.2)
translation	translation of an annotation into C code
typing	minimization of the instrumentation for integers (section 2.3.3)

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 can already be translated into C code and which cannot is defined in a separate 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[1] (see Chapter 10 of the FRAMA-C User Manual [5]). 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 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**
```
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.sh uninitialized.i -c -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 [12].

---

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

3.2 Incomplete Programs

Section 2.4 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 2.4.1 and 2.4.2).

3.2.1 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-mmodel` or the E-ACSL plug-in (of `-M` option of `e-acsl-gcc.sh`) is provided. In that case, violations of such annotations are undetected.

Consider the following example.

File **valid_no_main.c**

```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.sh -ML -omonitored_valid_no_main.i valid_no_main.c
<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' 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 call to the function `__e_acsl_memory_init` at the very beginning. This function plays a very important role: it initializes metadata storage used for tracking of memory blocks. Unless this call is inserted the run of a modified program is likely to fail.

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

File **modified_main.c**

```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.sh -M  -omonitored_modified_main.i modified_main.c
$ e-acsl-gcc.sh -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[3].

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.

3.4 Variadic Functions

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

3.5 Function Pointers

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

3.6 Requirements to Input Programs

3.6.1 E-ACSL Namespace

While E-ACSL uses source-to-source transformations and not binary instrumentations it is important that the source code provided at input does not contain any variables or functions

---

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

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.

3.6.2   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.

3.7   Limitations of E-ACSL Monitoring Libraries

E-ACSL monitoring library implementing segment-based memory model is fairly recent and presently has very limited support for 32-bit architectures. When using a 32-bit machine we recommend using the bittree-based memory model instead.

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 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, Loïc Correnson, and Zaynah Dargaye. *Frama-C's WP plug-in*. http://frama-c.com/download/frama-c-wp-manual.pdf.

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

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

[4] 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.

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

[6] 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.

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

[8] Arvid Jakobsson, Nikolai Kosmatov, and Julien Signoles. Rester statique pour devenir plus rapide, plus précis et plus mince. In David Baelde and Jade Alglave, editors, *Journées Francophones des Langages Applicatifs (JFLA'15)*, January 2015. In French.

[9] Arvid Jakobsson, Nikolai Kosmatov, and Julien Signoles. Fast as a Shadow, Expressive as a Tree: Optimized Memory Monitoring for C. *Science of Computer Programming*, pages 226–246, October 2016.

[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] 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.

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

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

[14] Kostyantyn Vorobyov, Julien Signoles, and Nikolai Kosmatov. Shadow State Encoding for Efficient Monitoring of Block-level Properties. Submitted for publication.

Index