A Lesson on Runtime Assertion Checking with Frama-C

Nikolai Kosmatov and Julien Signoles

CEA, LIST, Software Reliability Laboratory, PC 174
91191 Gif-sur-Yvette France
firstname.lastname@cea.fr

Abstract. Runtime assertion checking provides a powerful, highly automatizable technique to detect violations of specified program properties. This paper provides a lesson on runtime assertion checking with Frama-C, a publicly available toolset for analysis of C programs. We illustrate how a C program can be specified in executable specification language e-acsl and how this specification can be automatically translated into instrumented C code suitable for monitoring and runtime verification of specified properties. We show how various errors can be automatically detected on the instrumented code, including C runtime errors, failures in postconditions, assertions, preconditions of called functions, and memory leaks. Benefits of combining runtime assertion checking with other Frama-C analyzers are illustrated as well.

Keywords: runtime assertion checking, program monitoring, executable specification, invalid pointers, Frama-C, e-acsl.

1 Introduction

Runtime assertion checking has become nowadays a widely used programming practice [1]. More and more practitioners and researchers are interested in verification tools allowing to automatically check specified program properties at runtime. *Assertions* offer one of the most convenient and scalable automated techniques for detecting errors and providing information about their locations, even for errors that are traversed during execution but do not lead to failures.

This tutorial paper presents a lesson on runtime assertion checking using Frama-C [2, 3], an open-source platform dedicated to the analysis of C programs and developed at CEA LIST. It has an extensible architecture organized as a kernel with several plug-ins for individual analyses which may collaborate with each other. Frama-C offers various analyzers [3] such as control-flow graph construction, abstract-interpretation-based value analysis, dependency analysis, program slicing, automatic test generation, impact analysis, proof of programs, etc.

All static analyzers of Frama-C share a common specification language, called acsl [4]. To bridge the gap between static and dynamic tools, a recent work [5, 6] specified e-acsl, an expressive sub-language of acsl that can be

a)

```
1  #include<assert.h>
2  int absval( int x ) {
3    return ( x < 0 ) ? x : (-x);
4  }
5
6  void main(){
7    int n;
8    n=absval(0);   // test 1
9    assert(n==0);
10   n=absval(3);   // test 2
11   assert(n==3);
12   // other tests...
13 }
```

b)

```
1  // returns absolute value of x
2  int absval( int x ) {
3    return ( x < 0 ) ? x : (-x);
4  }
5
6  void main(){
7    int n;
8    n=absval(0);   // test 1
9    //@ assert n==0;
10   n=absval(3);   // test 2
11   //@ assert n==3;
12   // other tests...
13 }
```

Fig. 1: Function `absval`, specified **a)** with **`assert`** macros, **b)** with E-ACSL assertions

translated into C, compiled and used as executable specification. An automatic translator into C has been implemented in the E-ACSL plug-in [7] of Frama-C.

This paper is organized as follows. Section 2 presents the executable specification language E-ACSL and illustrates how the E-ACSL plug-in can be used to automatically translate the specified C code into an instrumented version suitable for runtime verification of specified properties. Section 3 shows how this solution can be used to automatically detect and report various kinds of errors such as wrong assertions, wrong postconditions, function calls in a wrong context (unsatisfied precondition) and memory leaks. The benefits of combining runtime assertion checking with proof of programs are discussed in Section 4. Section 5 illustrates how existing Frama-C analyzers can make runtime assertion checking even more efficient. On the one hand, automatic generation of assertions for frequent runtime errors may help to thoroughly check the program for these kinds of errors without manually writing assertions. On the other hand, some assertions may be statically validated by a static analysis tool, reducing the number of assertions to be checked. Section 6 shows how program monitoring with E-ACSL can be customized in order to define particular actions to be executed whenever a failure is detected. Finally, Section 7 concludes the paper and presents future work.

2 Executable Specifications in E-ACSL

Various ways of specifying assertions have been proposed in the literature [1]. One of the simplest ways to insert an assertion into a C program and to check it at runtime is to use the **`assert`** macro. Unless assertion checks are switched off (usually, by setting the preprocessor option NDEBUG), the condition specified as the argument of the **`assert`** macro is evaluated and whenever it is false, the execution stops and the failure is reported. Fig. 1a illustrates this approach for the function `absval` returning the absolute value of its argument, that is tested by the function `main`. The code contains an error (wrong inequality at line 3) that would be reported by the second assertion check (line 11).

a)

```
#include<limits.h>
/*@ requires x > INT_MIN;
    ensures (x >= 0 ==> \result == x) &&
      (x < 0 ==> \result == -x);
    assigns \nothing;
*/
int absval( int x ) {
  return ( x >= 0 ) ? x : (-x);
}
```

b)

```
#include<limits.h>
/*@ requires x > INT_MIN;
    assigns \nothing;
    behavior pos:
      assumes x >= 0;
      ensures \result == x;
    behavior neg:
      assumes x < 0;
      ensures \result == -x;
*/
int absval( int x ) {
  return ( x >= 0 ) ? x : (-x);
}
```

Fig. 2: Two ways to specify function `absval` with an E-ACSL contract

Specifying assertions in a programming language like C has several drawbacks. First, complex properties (e.g. with quantifications and different behaviors) can be difficult to specify and to check, may require significant additional programming effort and appear to be error-prone themselves. Comparing computed values with initial or intermediate ones may need storing some values in additional auxiliary variables. Some arithmetic properties (e.g. absence of overflows after an arithmetic operation) are simpler to express in a specification language with mathematical integers than using bounded machine integers in C. Second, clear separation of the code required only for assertion checking and the functional code may be desirable to optimize performances of the final release, but it often remains manual, even if preprocessor or configuration options may help to exclude assertion related statements in a particular build. Mixing both can make source code more heavy and less readable. Third, absence of unintended side-effects cannot be automatically ensured for assertion checking code written in C. Finally, the usage of resulting C assertions is limited to runtime verification and cannot be extended to verification techniques requiring formal specifications (such as proof of programs).

Other specification formalisms offer more expressive notations compatible with formal verification, for example, Eiffel [8], JML [9], SPEC# [10], SPARK 2014 [11] (see [1] for other references). One of them is the E-ACSL specification language [5, 6].

E-ACSL can express the C assertions of Fig. 1a using the **assert** clause as shown in Fig. 1b. However, E-ACSL language is much more expressive than C assertions. Indeed, in addition to C expressions and occasional assertions (like in Fig. 1b), an E-ACSL specification can include function contracts with preconditions and postconditions, behaviors, first-order logic predicates (with quantifications over finite intervals of integer values), mathematical integers (translated into C using GMP library[1] when required), references to the value of a variable or an expression at a particular program point (e.g. a label), memory-related clauses (specifying pointer validity, offset, memory block properties), etc. An E-ACSL

[1] http://gmplib.org

```
#include "stdlib.h"

typedef int* matrix;

/*@
requires size>=1;
requires \forall integer i,j; 0<=i<size && 0<=j<size ==> \valid(a+i*size+j);
requires \forall integer i,j; 0<=i<size && 0<=j<size ==> \valid(b+i*size+j);
ensures \forall integer i,j; 0<=i<size && 0<=j<size ==> \valid(\result+i*size+j);
ensures \forall integer i,j; 0<=i<size && 0<=j<size ==>
  \result[i*size+j] == a[i*size+j]+b[i*size+j];
*/
matrix sum(matrix a, matrix b, int size) {
  int i, j, k, idx;
  matrix c = (matrix)malloc(sizeof(int) * size * size );
  for(i = 0; i < size; i++)
    for(j = 0; j < size; j++) {
      idx = i * size + j;
      c[idx] = a[idx] + b[idx];
    }
  return c;
}
```

Fig. 3: File `sum.c` with function `sum` returning the sum of two given square matrices of given size in a new allocated matrix

clause cannot contain side-effects, so there is no risk to introduce an unintended modification of program behavior inside annotations. At the same time, specific *ghost code* statements can be used when side-effects are necessary. In addition, E-ACSL has been designed as a subset of ACSL, a specification language for C programs shared by each static analyzer of FRAMA-C. Therefore, an E-ACSL specification can also be used for instance for proof of programs as illustrated by Section 4.

Since E-ACSL statements are not limited to assertions, we will now use the term *annotation* (rather than *assertion*) to refer to any E-ACSL statement.

Fig. 2 illustrates two equivalent ways to specify the contract of the function `absval`. The contract of Fig. 2a contains a precondition and a postcondition. The precondition (**`requires`** clause) prevents the risk of an overflow since the value `-INT_MIN` cannot be represented by a machine number of type **`int`**. The postcondition contains an **`ensures`** clause and a frame clause **`assigns`** specifying that the function cannot modify any non-local variables. An equivalent specification using two behaviors is shown in Fig. 2b.

Another example of an E-ACSL contract is given in Fig. 3. Given two square matrices `a` and `b` with `size` rows and `size` columns, function `sum` allocates and returns a new matrix of the same size containing the sum of `a` and `b`. The validity clause **`\valid`**`(p)` specifies that the memory location pointed by `p` is valid. For more detail on C code specification in ACSL, we refer the reader to [4, 12].

```
#!/bin/sh
share=`frama-c -print-share-path`

frama-c -pp-annot -machdep x86_64 $1 -e-acsl -then-on e-acsl -print -ocode out.c

gcc -DE_ACSL_MACHDEP=x86_64 out.c $share/e-acsl/e_acsl.c
   $share/e-acsl/memory_model/e_acsl_bittree.c
   $share/e-acsl/memory_model/e_acsl_mmodel.c -o runme -lgmp

./runme
```

Fig. 4: Script `check.sh` that instruments the file given in its argument with E-ACSL plug-in, compiles and executes it

3 Detecting Errors at Runtime with E-ACSL Plug-in

In this section, we illustrate how the E-ACSL plug-in can be used for runtime verification of C programs. We consider several kinds of errors: failures in assertions and postconditions, function calls in a context that does not respect the callee's precondition, potential segmentation faults and memory leaks.

3.1 E-ACSL Plug-in and Assertion Failures

We assume that the FRAMA-C platform with E-ACSL plug-in[2] and the `gcc` compiler have been installed on the machine. Suppose the file `absval.c` contains the program of Fig. 1b. This program can be instrumented with the E-ACSL plug-in, compiled and executed on a Linux machine by the command `./check.sh absval.c` using the script given at Fig. 4. The options `-machdep x86_64` and `-DE_ACSL_MACHDEP=x86_64` should be used for a 64-bit machine (a 32-bit architecture `x86_32` is currently assumed by default).

The instrumentation translates the E-ACSL specification into executable C code that performs corresponding runtime checks and reports any failure. For Fig. 1b, two assertions at lines 9 and 11 will be checked at runtime exactly as for Fig. 1a. The first one is verified (producing no output), while the second one fails due to a wrong inequality at line 3. The program exits with an explicit message:
`Assertion failed at line 11 in function main. The failing predicate is: n == 3.`
Section 6 shows how the user can customize the actions when checking the validity of a predicate.

3.2 Function Contracts

Providing function contracts makes runtime assertion checking even more systematic and powerful. When a function is specified with an E-ACSL contract (with a pre- and/or a postcondition), the E-ACSL plug-in performs their systematic checks on entry (for the precondition) and on exit (for the postcondition) each time the function is called. Suppose we complete the file of Fig. 2a (or Fig. 2b) with a simple function

[2] Downloads and installation instructions available at http://frama-c.com/

```
int main(){
  absval(0);
  absval(3);
  absval(INT_MIN);
  return 0;
}
```

and instrument and run it using the script of Fig. 4. For each call of `absval`, the precondition is checked before the call and the postcondition is checked after the call without requiring any additional assertions. The checks for the first two calls succeed, while the call `absval(INT_MIN)` violates the precondition, so the program exits and reports:

```
Precondition failed at line 2 in function absval.
The failing predicate is: x > -2147483647-1.
```

If the inequality at line 8 of Fig. 2a was erroneously written again as "<", the program would exit after the call `absval(3)` and explicitly report a postcondition failure. In this way, the preconditions and postconditions are automatically ensured by the instrumented code for each function call. In particular, the precondition check guarantees that the function is called on admissible inputs and prevents from calling it in an inappropriate context. The current release of E-ACSL does not yet support runtime checking of the **assigns** clause, this feature will be integrated in a future version.

3.3 Segmentation Faults

Segmentation faults represent one of the most important issues in C programs. Some of them may lead to runtime errors, others may remain unnoticed and provoke memory corruption. To address this issue, the E-ACSL plug-in provides a memory monitoring library [13] and allows the user to check specified memory-related properties at runtime. Let us illustrate this feature on the program of Fig. 3 completed with the following test function.

```
int main(void) {
  int a[]={1,1,1,1}, b[]={2,2,2}; // one element missing, should be {2,2,2,2}
  matrix c = sum(a, b, 2);
  free(c);
  return 0;
}
```

The resulting file can be instrumented by the E-ACSL plug-in and run using the script of Fig. 4. The execution reports a failure in the precondition of `sum` at line 8 of Fig. 3 since the array `b` has 3 elements instead of 4 as expected for a 2×2 matrix. Thanks to the precondition, E-ACSL prevents an invalid access to `b[3]` at line 18 of Fig. 3 that would be out-of-bounds. This is an example of a *spatial error*, i.e. an invalid memory access due to an out-of-bounds offset or array index. To correct this error, the array `b` must be initialized with 4 elements.

Let us now illustrate some more subtle errors detected by E-ACSL. Replace the line 15 of Fig. 3 by a local array `int c[4];` and complete the resulting program with the following test function.

```
int main(void) {
  int a[]={1,1,1,1}, b[]={2,2,2,2};
```

```c
  matrix c = sum(a, b, 2);
  int trace = c[0] + c[2];
  free(c);
  return 0;
}
```

Runtime checking with the E-ACSL plug-in for this example reports a postcondition failure at line 9 of Fig. 3 since the elements of `c` are not valid after exiting the function `sum`. Thanks to the postcondition, E-ACSL prevents invalid accesses to the elements `c[0]` and `c[2]` in function `main`. This is an example of a *temporal error*, i.e. an invalid memory access to a deallocated memory object.

Another example of a temporal error is a use-after-free. To give an example, we complete the program of Fig. 3 with the function:

```c
int main(void) {
  int a[]={1,1,1,1}, b[]={2,2,2,2};
  matrix c = sum(a, b, 2);
  free(c);
  //@ assert \valid(&c[0]) && \valid(&c[2]);
  int trace = c[0] + c[2];
  return 0;
}
```

Runtime checking with the E-ACSL plug-in for this example reports an assertion failure since the accesses to `c[0]` and `c[2]` in function `main` become invalid after the array is freed.

Notice that memory access validity is only checked when it is specified by (or required for safe translation of) the provided E-ACSL annotations. We show in Section 5 how to enforce memory safety checks with the E-ACSL plug-in by automatic generation of annotation using the RTE plug-in.

3.4 Memory Leaks

Memory leaks represent another common type of defects in programs with intensive dynamic memory allocation. A *memory leak* appears when an inaddressable memory object remains uselessly stored on the system reducing the amount of available memory. In some cases, memory leaks can be a serious problem for programs running for a long time and/or containing frequent memory allocations. When the amount of available memory decreases, the developer may suspect memory leaks. In this case it can be helpful to define (or to bound) the expected difference of the size of dynamically allocated memory at two program points. If the amount of allocated memory increases unexpectedly, additional annotations specifying the difference between closer and closer points may help to find the precise function where the memory leak occurs. The E-ACSL plug-in provides such a means to precisely control the amount of dynamically allocated memory that helps to detect memory leaks.

To illustrate this feature, consider the program of Fig. 5. It includes the file `sum.c` of Fig. 3 and defines a function `sum3` computing the sum of three given matrices. If the program performs frequent calls to such a function `sum3`, the amount of available memory will decrease. The size of dynamically allocated memory in bytes can be referred by the variable `__memory_size` (line 2 of Fig. 5)

```
 1 #include "sum.c"
 2 extern size_t __memory_size;
 3
 4 matrix sum3(matrix a, matrix b, matrix c, int size) {
 5   matrix x, y;
 6   x = sum(a, b, size);
 7   y = sum(x, c, size);
 8   return y;
 9 }
10
11 void main(void) {
12   int a[] = {1,1,1,1}, b[] = {2,2,2,2}, c[] = {3,3,3,3};
13   matrix d;
14   L1: d = sum3(a, b, c, 2);
15   L2: free(d);
16   //@ assert \at(__memory_size,L2) - \at(__memory_size,L1) <= sizeof(int) * 4;
17 }
```

Fig. 5: Function `sum3` returns the sum of three given square matrices of given size

defined by the memory monitoring library of the E-ACSL plug-in. Line 16 of function `main` illustrates how the user can specify that the amount of dynamically allocated memory must not change between two points, at label `L1` before the call to `sum3` and at line 16 after deallocating the returned array `d`. This assertion fails, so the user may investigate this difference between two closer points, replacing the assertion at line 16 by another one:

```
//@ assert \at(__memory_size,L2)-\at(__memory_size,L1) <= sizeof(int)*size*size;
```

indicating that only an array of `size * size` integers can be allocated between `L1` and `L2`. This assertion fails again, indicating that the memory leak probably happens inside the function `sum3`. Inserting a postcondition of function `sum3` at line 3

```
//@ ensures __memory_size - \old(__memory_size) == sizeof(int) * size * size;
```

precisely comparing the size of dynamically allocated memory before and after the function call is another way to find out that the memory leak occurs in function `sum3`. The problem is indeed due to the allocation for the array returned at line 6 of Fig. 5 that becomes inaddressable when function `sum3` exits. To solve this issue, `free(x);` should be inserted before the **`return`** statement at line 8 of Fig. 5.

4 Runtime Checking and Analysis of Proof Failures

As we have shown in the previous section, runtime assertion checking helps to ensure that the program execution respects the provided annotations. When the annotations are supposed to be correct, an annotation failure reveals an error in the program. But runtime assertion checking can be also used in a dual way, to find a potentially incorrect annotation. It can also provide more confidence in conformance of the program to its specification when no failures are detected. This approach can be very helpful during proof of programs.

Indeed, in order to formally prove a program, the validation engineer has to specify it. This is a tedious task, and errors in the first versions of specifications are very common. Moreover, when the program proof fails, the proof failure is not necessarily due to a wrong specification or a wrong implementation, but can be also due to the incapacity of the proving tool to find the proof automatically. Proof failures have to be analyzed and fixed manually.

Runtime assertion checking provides an automatic technique allowing the validation engineer to check the conformance between program and specification at runtime on a number of program executions. The instrumented program can be executed on an available test suite, or generated test inputs (e.g. randomly, or by a structural test generation tool like PathCrawler [14], another Frama-C plug-in). Runtime checking does not give a precise answer in all cases, but it can provide a useful indication. When a failure is detected at runtime, the failed annotation and the corresponding program inputs indicate the case that has not been properly taken into consideration in the implementation or in the specification. The engineer can immediately deduce that the proof failure is not due to the limitations of the prover. For example, if the postcondition of the program of Fig. 2a were erroneously written as

```
ensures (x >= 0 && \result == x) && (x < 0 && \result == -x);
```

or as

```
ensures (x >= 0 && \result == x) || (x < 0 && \result == x);
```

the failure would be detected on a concrete program input as illustrated in Sec. 3.2. Although the specification error is obvious for this simple example, automatic runtime checking can save significant time for more complex programs.

When no failure is detected at runtime, the prover may need additional annotations (e.g. assertions, loop invariants), so the specification effort is worth to be continued (even if the risk of an error cannot be completely excluded since runtime checking was performed only on a final set of tests).

5 Combinations with Other Analyzers

Frama-C is a platform which provides a wide variety of plug-ins. The E-ACSL plug-in is only one of them. It is possible to make E-ACSL more efficient by combining it with other existing plug-ins. Section 5.1 explains how to automatically generate annotations to be checked by E-ACSL, while Section 5.2 shows how to reduce the number of dynamic checks by verifying some of them statically.

5.1 Generating Annotations Automatically

The E-ACSL plug-in may be used to dynamically verify the absence of runtime errors in C code, by combining it with the RTE plug-in. Indeed, this plug-in generates E-ACSL annotations preventing potential runtime errors: if these annotations are proven valid, then we get the guarantee that the code provokes no runtime error. For instance, RTE generates `/*@ assert \valid_read(p); */` each time a pointer p is read, and two assertions `/*@ assert 0 <= i; */` and `/*@ assert i < 10; */` each time an access `t[i]` in an array `t` of length 10 is performed. If these annotations are translated into C code by the E-ACSL plug-in, they will be checked at runtime: either a dynamic check fails and the program reports a clear assertion failure, or no dynamic check fails and the whole program does not fail with a runtime error either. Therefore, a potential runtime error does not remain unnoticed thanks to explicit annotations added by RTE and checked by E-ACSL.

Consider for instance the function `sum` of Fig. 3 in which we modify line 18 `idx = i * size + j;` into the incorrect line `idx = i * size + j + 1;`. Since the index `idx` is now too great, that introduces an access out of the bounds of the matrices `a`, `b`, and `c` when computing the sum. We also complete this program with the following test function.

```
int main(void) {
  int a[]={1,1,1,1}, b[]={2,2,2,2};
  matrix c = sum(a, b, 2);
  free(c);
  return 0;
}
```

Running the RTE plug-in on this program generates several additional annotations corresponding to potential arithmetic overflows while computing `idx` or the sum of the matrices' elements, and potential invalid memory accesses when reading the matrices' elements. You can see them in the FRAMA-C GUI by running the following command.

```
frama-c-gui -rte sum.c
```

Below are two examples of such annotations.

```
/*@ assert rte: signed_overflow: i*size <= 2147483647; */
/*@ assert rte: mem_access: \valid(c+idx); */
```

We can now combine the RTE and E-ACSL plug-ins to translate these additional annotations (and the already existing ones) into C code in the following way.

```
frama-c sum.c -rte -machdep x86_64 -e-acsl-prepare -then -e-acsl \
        -then-on e-acsl -print -ocode out.c
```

The special option `-e-acsl-prepare` tells FRAMA-C to generate an E-ACSL-compatible abstract syntax tree (AST) of the given program. It is required when the computation of the AST is needed by another analysis than E-ACSL (here by the generation of annotations by RTE).

Now, if we compile and run the resulting program `out.c` as shown in Fig. 4, we get the following output which indicates that an assertion preventing a memory-access error failed when dereferencing `c+idx` at line 19.

```
Assertion failed at line 19 in function sum.
The failing predicate is:
rte: mem_access: \valid(c+idx).
```

5.2 Verifying Annotations Statically

Frama-C comes with two static analyzers which can be used to verify ACSL specifications: Value, based on abstract interpretation [15], and WP, based on weakest precondition calculus [16].

Consider again the function `sum` of Fig. 3 with the following test function.

```
int main(void) {
  int a[]={1,1,1,1}, b[]={2,2,2,2};
  matrix c = sum(a, b, 2);
  free(c);
  return 0;
}
```

Combination with WP Plug-in Running WP on this program (with the automatic theorem prover Alt-Ergo [17]) automatically proves the preconditions of function `sum` as you can see, for instance, running the Frama-C GUI:

```
frama-c-gui -wp sum.c
```

It does not prove, however, the postconditions of the function: such a proof with WP requires to write loop invariants for both loops of the function. Nevertheless it is possible to combine WP and E-ACSL: WP proves the preconditions[3], while E-ACSL establishes that the postconditions are not violated for a given execution. By default, the E-ACSL plug-in does not perform code instrumentation to check already proved properties: if WP proves the preconditions first, the code generated by E-ACSL performs fewer runtime checks and so is more efficient. Such a combination is run by the following command.

```
frama-c -machdep x86_64 -e-acsl-prepare -wp sum.c -then -e-acsl \
        -then-on e-acsl -print -ocode out.c
```

The generated program `out.c` is then linked and executed as usual. In this particular case, it does not report any error since the program is actually correct.

Combination with Value Plug-in We can run Value on the same example as follows.

```
frama-c -val sum.c
```

Below is a summary of the output.

```
sum.c:6:[value] Function sum: precondition got status valid.
sum.c:7:[value] Function sum: precondition got status unknown.
sum.c:8:[value] Function sum: precondition got status unknown.
sum.c:19:[kernel] warning: out of bounds write. assert \valid(c+idx);
sum.c:9:[value] Function sum: postcondition got status unknown.
sum.c:10:[value] Function sum: postcondition got status unknown.
```

It indicates than Value automatically proves the first precondition `size >= 1`, but does not prove neither the other ones[4] nor the postconditions. Value also generates an ACSL annotation because it detects a potential out-of-bounds access

[3] The WP's default memory model assumes that `malloc` never returns `NULL`.

[4] The Value's default memory model assumes that `malloc` may return `NULL`.

when writing into the array cell `c[idx]` at address `c+idx` . It does not detect similar out-of-bounds accesses for reading `a[idx]` and `b[idx]` because it assumes that the preconditions of the function are valid.

In the same way as for WP, we can combine VALUE and E-ACSL to dynamically check with E-ACSL only what remains unproved by VALUE. E-ACSL will also check the additional annotation generated by VALUE. Such a combination is run by the following command.

```
frama-c -machdep x86_64 -e-acsl-prepare -val sum.c -then -e-acsl \
        -then-on e-acsl -print -ocode out.c
```

Linking and executing the generated program in the usual way does not report any failure on this correct program.

6 Customization of Runtime Monitoring

By default, as shown in the previous examples, when the evaluation of a predicate fails at runtime, the execution stops with an error message and exit code 1. This behavior is implemented by the function `e_acsl_assert` provided in the file `e_acsl.c` of the E-ACSL library. This function is called each time an annotation is checked. It is fully possible to modify this behavior by providing your own definition of `e_acsl_assert`. The prototype of the function to implement is as follows.

```
void e_acsl_assert(int, char *, char *, char *, int);
```

For each annotation *a* to be checked, the parameters are the following:

- the first one is the validity status of *a* (0 if false, non-zero if true);
- the second one is a string describing the kind of *a* (an assertion, a precondition, a postcondition, etc);
- the third one is the function name where *a* takes place;
- the fourth one is a textual description of *a*;
- the fifth one is the line number of *a* in the source file.

For instance, Fig. 6 provides an implementation which does not stop the program execution, but appends an error message at the end of file `log_file.log` when an annotation is violated.

Then, in the script of Fig. 4, we can replace the file `$share/e-acsl.c` by the one defining the function of Fig. 6. Thus, executing it on the binary generated from the first example of Section 3.3 generates a file `log_file.log` which contains the following lines:

```
Precondition failed at line 8 in function sum.
The failing predicate is:
\forall integer i, integer j;
  (0 <= i && i < size) && (0 <= j && j < size) ==> \valid((b+i*size)+j).
RTE failed at line 11 in function sum.
The failing predicate is:
mem_access:
  \valid_read(__e_acsl_at_7
  +(long long)((long long)((long long)__e_acsl_i_4*(long long)__e_acsl_at_8)
  +(long long)__e_acsl_j_4)).
```

```c
#include <stdio.h>

void e_acsl_assert
  (int predicate, char *kind, char *fct, char *pred_txt, int line)
{
  if (! predicate) {
    FILE *f = fopen("log_file.log", "a");
    fprintf(f,
            "%s␣failed␣at␣line␣%d␣in␣function␣%s.\n\
The␣failing␣predicate␣is:\n%s.\n",
            kind, line, fct, pred_txt);
    fclose(f);
  }
}
```

Fig. 6: Modifying the runtime behavior when an annotation is violated

It indicates that there were actually two failed runtime checks. The former was previously described in Section 3.3 and corresponds to the invalid precondition about the array `b` (which has 3 elements while 4 are expected). The latter corresponds to an out-of-bounds access error detected when trying to evaluate the postcondition since `b[i*size+j]` tries to access the fourth element of the array `b` of length 3 (when $i = 1$, $j = 1$ and $size = 2$).

7 Conclusion and Future Work

In this tutorial paper, we have presented how the E-ACSL plug-in of FRAMA-C can be used to perform runtime assertion checking of C programs. The user expresses the expected properties of the program statements and functions in a powerful formal specification language. These properties are then automatically translated into C code in order to be checked at runtime.

In addition to usual runtime assertion checking, E-ACSL may help to debug specifications before proving a program formally. When combined with other FRAMA-C analyzers, E-ACSL is even more efficient: it may automatically check for any runtime errors in C programs, and may discard runtime checks of properties previously statically verified.

Future work includes the support of missing parts of the E-ACSL specification language, in particular assigns clauses, loop invariants and variants, and logic functions and predicates. It also includes the verification of new kinds of properties like additional memory temporal properties, LTL properties or security properties, and new areas of application like combinations of testing and static analysis, security monitoring and teaching formal specification.

References

1. Clarke, L.A., Rosenblum, D.S.: A historical perspective on runtime assertion checking in software development. ACM SIGSOFT Software Engineering Notes **31**(3) (2006) 25–37

2. Correnson, L., Cuoq, P., Kirchner, F., Prevosto, V., Puccetti, A., Signoles, J., Yakobowski, B.: Frama-C User Manual. (April 2013) `http://frama-c.com`.
3. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C, a program analysis perspective. In: the 10th International Conference on Software Engineering and Formal Methods (SEFM 2012). Volume 7504 of LNCS., Springer (2012) 233–247
4. Baudin, P., Filliâtre, J.C., Hubert, T., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, v1.6. (April 2013) URL: http://frama-c.com/acsl.html.
5. Signoles, J.: E-ACSL: Executable ANSI/ISO C Specification Language. (May 2013) URL: http://frama-c.com/download/e-acsl/e-acsl.pdf.
6. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of C programs. In: the 28th Annual ACM Symposium on Applied Computing (SAC 2013), ACM (2013) 1230–1235
7. Signoles, J.: E-ACSL User Manual. (May 2013) URL: http://frama-c.com/download/e-acsl/
8. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall, Inc. (1988)
9. Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., Cok, D.R.: How the design of JML accommodates both runtime assertion checking and formal verification. Sci. Comput. Program. **55**(1-3) (2005) 185–208
10. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: the Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, Int. Workshop (CASSIS 2004). Volume 3362 of LNCS., Springer (2004) 49–69
11. AdaCore and Altran UK Ltd: SPARK 2014 Reference Manual (2013) `http://docs.adacore.com/spark2014-docs/html/lrm/`.
12. Kosmatov, N., Prevosto, V., Signoles, J.: A lesson on proof of programs with Frama-C. Invited tutorial paper. In: the 7th International Conference on Tests and Proofs (TAP 2013). Volume 7942 of LNCS., Springer (2013) 168–177
13. Kosmatov, N., Petiot, G., Signoles, J.: Optimized memory monitoring for runtime assertion checking of C programs. In: the 4th International Conference on Runtime Verification (RV 2013). LNCS, Springer (2013) To appear.
14. Botella, B., Delahaye, M., Hong-Tuan-Ha, S., Kosmatov, N., Mouy, P., Roger, M., Williams, N.: Automating structural testing of C programs: Experience with PathCrawler. In: the 4th Int. Workshop on Automation of Software Test (AST 2009), IEEE Computer Society (2009) 70–78
15. Cousot, P..R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. (1977) 238–252
16. Dijkstra, E.W.: A constructive approach to program correctness. BIT Numerical Mathematics (1968)
17. Conchon *et al*, S.: The Alt-Ergo Automated Theorem Prover `http://alt-ergo.lri.fr/`.