Software Analyzers

RTE
Runtime Error Annotation Generation

For Frama-C 32.0 (Germanium)

Philippe Herrmann, Julien Signoles and Allan Blanchard

Contents

Introduction 1

1.1 RTE plug-in

This document is a reference manual for the annotation generator plug-in called RTE. The aim of the RTE plug-in is to automatically generate annotations for:

- common runtime errors, such as division by zero, signed integer overflow or invalid memory access;
- unsigned integer overflows, which are allowed by the C language but may pose problem to solvers;

In a modular proof setting, the main purpose of the RTE plug-in is to seed more advanced plug-ins (such as the weakest-preconditions generation plug-in [1]) with proof obligations. Annotations can also be generated for their own sake in order to guard against runtime errors. The reader should be aware that discharging such annotations is much more difficult than simply generating them, and that there is no guarantee that a plug-in such as Frama-C's value analysis [4] will be able to do so automatically in all cases.

RTE performs syntactic constant folding in order not to generate trivially valid annotations. Constant folding is also used to directly flag some annotations with an invalid status. RTE does not perform any kind of advanced value analysis, and does not stop annotation generation when flagging an annotation as invalid, although it may generate fewer annotations in this case for a given statement.

Like most Frama-C plug-ins, RTE makes use of the hypothesis that signed integers have a two's complement representation, which is a common implementation choice. Also note that annotations are dependent of the *machine dependency* used on Frama-C command-line, especially the size of integer types.

The C language ISO standard [5] will be referred to as ISO C23 (of which specific paragraphs are cited, such as ISO C23 6.2.5 § 9).

1.2 Runtime errors

A runtime error is a usually fatal problem encountered when a program is executed. Typical fatal problems are segmentation faults (the program tries to access memory that it is not allowed to access) and floating point exceptions (for instance when dividing an integer by zero: despite its name, this exception does not only occur when dealing with floating point arithmetic). A C program may contain "dangerous" constructs which under certain conditions lead to runtime errors when executed. For instance evaluation of the expression `u / v` will always produce a floating point exception when `v = 0` holds. Writing to an out-of-bound index of an array may result in a segmentation fault, and it is dangerous even if it fails to do so (other variables may be overwritten). The goal of this Frama-C plug-in is to detect a number of such constructs, and to insert a corresponding logical annotation (a first-order property

over the variables of the construct) ensuring that, whenever this annotation is satisfied before execution of the statement containing the construct, the potential runtime error associated with the expression will not happen. Annotation checking can be performed (at least partially) by Frama-C value analysis plug-in [4], while more complicated properties may involve other plug-ins and more user interaction.

At this point it is necessary to define what one means by a "dangerous" construct. ISO C23 lists a number of *undefined* behaviors (the program construct can, at least in certain cases, be erroneous), a number of *unspecified* behaviors (the program construct can be interpreted in at least two ways), and a list of *implementation-defined* behaviors (different compilers and architectures implement different behaviors). Constructs leading to such behaviors are considered dangerous, even if they do not systematically lead to runtime errors. In fact an undefined behavior must be considered as potentially leading to a runtime error, while unspecified and implementation-defined behaviors will most likely result in portability problems.

An example of an undefined behavior (for the C language) is *signed integer overflow*, which occurs when the (exact) result of a signed integer arithmetic expression can not be represented in the domain of the type of the expressions. For instance, supposing that an `int` is 32-bits wide, and thus has domain `[-2147483648,2147483647]`, and that `x` is an `int`, the expression `x+1` performs a signed integer overflow, and therefore has an undefined behavior, if and only if `x` equals `2147483647`. This is independent of the fact that for most (if not all) C compilers and 32-bits architectures, one will get `x+1 = -2147483648` and no runtime error will happen. But by strictly conforming to the C standard, one cannot assert that the C compiler will not in fact generate code provoking a runtime error in this case, since it is allowed to do so. Also note that from a security analysis point of view, an undefined behavior leading to a runtime error classifies as a denial of service (since the program terminates), while a signed integer overflow may very well lead to buffer overflows and execution of arbitrary code by an attacker. Thus, not getting a runtime error on an undefined behavior is not necessarily a desirable behavior.

On the other hand, note that a number of behaviors classified as implementation-defined by the ISO standard are quite painful to deal with in full generality. In particular, ISO C23 allows either *sign and magnitude*, *two's complement* or *one's complement* for representing signed integer values. Since most if not all "modern" architectures are based on a *two's complement* representation (and that compilers tend to use the hardware at their disposal), it would be a waste of time not to build verification tools by making such wide-ranging and easily checkable assumptions. **Therefore, RTE uses the hypothesis that signed integers have a *two's complement* representation.**

1.3 Other annotations generated

RTE may also generate annotations that are not related to runtime errors:

- absence of unsigned overflows checking, although unsigned overflows are well-defined, some plug-ins may wish to avoid them;
- accesses to arrays that are embedded in a struct occur within valid bounds, this is stricter than verifying that the accesses occur within the struct.

Runtime error annotation generation 2

2.1 Integer operations

According to ISO C23 6.2.5, operations on unsigned integers "can never overflow" (as long as the result is defined, which excludes division by zero): they are reduced modulo a value which is one greater than the largest value of their unsigned integer type (typically 2^n for n-bit integers). So in fact, arithmetic operations on unsigned integers should really be understood as modular arithmetic operations (the modulus being the largest value plus one).

On the other hand, an operation on *signed* integers might overflow and this would produce an undefined behavior. Hence, a signed integer operation is only defined if its result (as a mathematical integer) falls into the interval of values corresponding to its type (e.g. `[INT_MIN,INT_MAX]` for `int` type, where the bounds `INT_MIN` and `INT_MAX` are defined in the standard header `limits.h`). Therefore, signed arithmetic is true integer arithmetic as long as intermediate results are within certain bounds, and becomes undefined as soon as a computation falls outside the scope of representable values of its type.

The full list of arithmetic and logic operations which might overflow is presented hereafter. Most of these overflows produce undefined behaviors, but some of them are implementation defined and indicated as such.

2.1.1 Addition, subtraction, multiplication

These arithmetic operations may not overflow when performed on signed operands, in the sense that the result must fall in an interval which is given by the type of the corresponding expression and the macro-values defined in the standard header `limits.h`. A definition of this file can be found in the `share` directory of Frama-C.

type	representable interval
`signed char`	`[SCHAR_MIN, SCHAR_MAX]`
`signed short`	`[SHRT_MIN,SHRT_MAX]`
`signed int`	`[INT_MIN,INT_MAX]`
`signed long int`	`[LONG_MIN,LONG_MAX]`
`signed long long int`	`[LLONG_MIN,LLONG_MAX]`

Since RTE makes the assumption that signed integers are represented in 2's complement, the interval of representable values also corresponds to $[-2^{n-1}, 2^{n-1}-1]$ where n is the number of bits used for the type (sign bit included, but not the padding bits if there are any). The size in bits of a type is obtained through `Cil.bitsSizeOf: typ -> int`, which bases itself on the machine dependency option of Frama-C. For instance by using `-machdep x86_32`, we have the following (the size is expressed in bits):

type	size	representable interval
`signed char`	8	`[-128,127]`
`signed short`	16	`[-32768,32767]`
`signed int`	32	`[-2147483648,2147483647]`
`signed long int`	32	`[-2147483648,2147483647]`
`signed long long int`	64	`[-9223372036854775808,9223372036854775807]`

Frama-C annotations added by plug-ins such as RTE may not contain macros since preprocessing is supposed to take place beforehand (user annotations at the source level can be taken into account by using the `-pp-annot` option). As a consequence, annotations are displayed with big constants such as those appearing in this table.

Example 2.1 *Here is a RTE-like output in a program involving* `signed long int` *with an* `x86_32` *machine dependency:*

```
int main(void) {
  signed long int lx, ly, lz;

  /*@ assert rte: signed_overflow: -2147483648 <= lx*ly; */
  /*@ assert rte: signed_overflow: lx*ly <= 2147483647; */
  lz = lx * ly;

  return 0;
}
```

The same program, but now annotated with an `x86_64` *machine dependency (option* `-machdep x86_64`*):*

```
int main(void) {
  signed long int lx, ly, lz;

  /*@ assert rte: signed_overflow: -9223372036854775808 <= lx*ly; */
  /*@ assert rte: signed_overflow: lx*ly <= 9223372036854775807; */
  lz = lx * ly;

  return 1;
}
```

The difference comes from the fact that `signed long int` *is 32-bit wide for* `x86_32`*, and 64-bit wide for* `x86_64`*.*

2.1.2 Signed downcasting

Note that arithmetic operations usually involve arithmetic conversions. For instance, integer expressions with rank lower than `int` are promoted, thus the following program:

```
int main(void) {
  signed char cx, cy, cz;

  cz = cx + cy;
  return 0;
}
```

is in fact equivalent to:

```
int main(void) {
  signed char cx, cy, cz;
```

```
  cz = (signed char)((int)cx + (int)cy);
  return 0;
}
```

Since a signed overflow can occur on expression `(int)cx + (int)cy`, the following annotations are generated by the RTE plug-in:

```
/*@ assert rte: signed_overflow: -2147483648 <= (int)cx+(int)cy; */
/*@ assert rte: signed_overflow: (int)cx+(int)cy <= 2147483647; */
```

This is much less constraining than what one would want to infer, namely:

```
/*@ assert (int)cx+(int)cy <= 127; */
/*@ assert -128 <= (int)cx+(int)cy; */
```

Actually, by setting the option `-warn-signed-downcast` (which is unset by default), the RTE plug-in infers these second (stronger) assertions when treating the cast of the expression to a `signed char`. Since the value represented by the expression cannot in general be represented as a `signed char`, and following paragraph ISO C23 6.3.1.3 § 3 (on downcasting to a signed type), an *implementation-defined behavior* happens whenever the result falls outside the range `[-128,127]`. Thus, with a single annotation, the RTE plug-in prevents both an undefined behavior (signed overflow) and an implementation defined behavior (signed downcasting). Note that the annotation for signed downcasting always entails the annotation for signed overflow.

2.1.3 Unary minus

The only case when a (signed) unary minus integer expression `-expr` overflows is when `expr` is equal to the minimum value of the integer type. Thus the generated assertion is as follows:

```
int ix;
// some code
/*@ assert rte: signed_overflow: -2147483647 <= ix; */
ix = - ix;
```

2.1.4 Division and modulo

As of ISO C23 6.5.6 § 6, an undefined behavior occurs whenever the value of the second operand of operators `/` and `%` is zero. The corresponding runtime error is usually referred to as “division by zero”. This may happen for both signed and unsigned operations.

```
unsigned int ux;
// some code
/*@ assert rte: division_by_zero: ux != 0; */
ux = 1 / ux;
```

In 2’s complement representation and for signed division, dividing the minimum value of an integer type by -1 overflows , since it would give the maximum value plus one. There is no such rule for signed modulo, since the result would be zero, which does not overflow.

```
int x,y,z;
// some code
/*@ assert rte: division_by_zero: x != 0; */
/*@ assert rte: signed_overflow: y/x <= 2147483647; */
z = y / x;
```

2.1.5 Bitwise shift operators

ISO C23 6.5.8 defines undefined and implementation defined behaviors for bitwise shift operators. The type of the result is the type of the promoted left operand.

The undefined behaviors are the following:

– the value of the right operand is negative or is greater than or equal to the width of the promoted left operand:

```
int x,y,z;

/*@ assert rte: shift: 0 <= y < 32; */
z = x << y; // same annotation for z = x >> y;
```

– in `E1 << E2`, `E1` has signed type and negative value:

```
int x,y,z;

/*@ assert rte: shift: 0 <= x; */
z = x << y;
```

– in `E1 << E2`, `E1` has signed type and nonnegative value, but the value of the result $\mathrm{E1} \times 2^{\mathrm{E2}}$ is not representable in the result type:

```
int x,y,z;

/*@ assert rte: signed_overflow: x<<y <= 2147483647; */
z = x << y;
```

There is also an implementation defined behavior if in `E1 >> E2`, `E1` has signed type and negative value. This case corresponds to the arithmetic right-shift, usually defined as signed division by a power of two, with two possible implementations: either by rounding the result towards minus infinity (which is standard) or by rounding towards zero. RTE generates an annotation for this implementation defined behavior.

```
int x,y,z;

/*@ assert rte: shift: 0 <= x; */
z = x << y;
```

Example 2.2 *The following example summarizes RTE generated annotations for bitwise shift operations, with* `-machdep x86_64`*:*

```
long x,y,z;

/*@ assert rte: shift: 0 <= y < 64; */
/*@ assert rte: shift: 0 <= x; */
/*@ assert rte: signed_overflow: x<<y <= 9223372036854775807; */
z = x << y;

/*@ assert rte: shift: 0 <= y < 64; */
/*@ assert rte: shift: 0 <= x; */
z = x >> y;
```

2.1.6 Boolean values

Boolean objects have two valid values which are `true` and `false`. Other values are non-value representations, and reading them is undefined behavior (ISO C23 6.2.6.1 § 5). One can create such a value only using strict-aliasing violation (see ISO C23 6.5 § 6), since conversions are well-defined. However, Frama-C does not handle strict-aliasing rule (see Section 2.8.1), thus we generate on boolean value reads.

Example 2.3 *An example of RTE annotation generation for verification of boolean values:*

```
bool ok2(int a, bool b)
{
  bool __retres;
  int tmp;
  /*@ assert rte: bool_value: b == 0 || b == 1; */
  tmp = g((bool)(a > 0),b);
  __retres = (bool)(tmp != 0);
  /*@ assert rte: bool_value: __retres == 0 || __retres == 1; */
  return __retres;
}
```

2.2 Pointers

According to ISO C23, different operations related to pointers can cause undefined behaviors. We can mainly distinguish three cases:

- the value of the pointer becomes invalid: creating such a pointer is an undefined behavior;
- it is illegal to dereference the pointer (which might be because of its type);
- the pointer is converted to a type that cannot always handle it.

This section covers this topic.

2.2.1 Invalid pointer value

According to ISO C23, there are two ways of creating an invalid pointer value:

- by using pointer arithmetic such that the offset leaves the bounds of the object, that is, from the beginning of the object to past-the-end (ISO C23 6.5.7 § 9),
- by a conversion that creates an unaligned pointer, either from another from an integer type (ISO C23 6.3.2.3 § 5) or from a pointer type (ISO C23 6.3.2.3 § 7).

By default, checking the first condition is **not** enabled by Frama-C. For the rationale about this choice, please refer to the Frama-C user manual [3]. The generation is controlled by the kernel option `-warn-invalid-pointer`. When this option is enabled, RTE generates annotations to check that pointers remain `NULL` or in the bounds of an existing object via the ACSL built-in predicate `\object_pointer`. `\object_pointer(s)`, with `s` being a set of pointers, holds if and only if all pointers of the set point to an object or past-the-end of the object. These annotations are generated when:

- a pointer arithmetic operation happens,
- a value with integer type is converted to a pointer type,
- the address of an lvalue is taken.

Example 2.4 *An example of RTE annotation generation for verification of pointer value validity:*

```
void f(void)
{
  int a[4];
  /*@ assert rte: pointer_value: \null == &a[4] || \object_pointer(&a[4]); */
  int *p = & a[4];
  /*@ assert rte: pointer_value: \null == p + 1 || \object_pointer(p + 1); */
  p ++;
  /*@ assert
        rte: pointer_value: \null == (int *)4 || \object_pointer((int *)4);
```

```
  */
  int *q = (int *)4;
  return;
}
```

In this example, the first assertion is verified since `p` *points to just past-the-end of* `a`*, while the second is not. The third operation depends on the status of the kernel option* `-absolute-valid-range`*.*

The alignment is controlled via the ACSL predicate `\aligned`. `\aligned(p, n)` holds if and only if `p` is aligned modulo `n`. These annotations are generated when:

- a value with integer type is converted to a pointer type,
- a value with pointer type is converted to another pointer type.

Example 2.5 *An example of RTE annotation generation for verification of pointer alignment:*

```
void f(void)
{
  char a1[4];
  char  _Alignas(_Alignof(int)) a2[4];
  /*@ assert rte: pointer_alignment: \aligned((char *)a1,alignof(int)); */
  int *p1 = (int *)(a1);
  /*@ assert rte: pointer_alignment: \aligned((char *)a2,alignof(int)); */
  int *p2 = (int *)(a2);
  /*@ assert rte: pointer_alignment: \aligned((int*)1,alignof(int)); */
  int *q = (int *)1;
  return;
}
```

In this example, the status of the first annotation is unknown since it depends on the memory layout. The second is trivially true (and thus, without the option `-rte-trivial-annotations` *it is not generated). The last one is a guaranteed runtime error on most systems.*

Because strict aliasing rule is not verified in Frama-C and RTE (see Section 2.8.1), it is possible to indirectly generate invalid pointer values. In order to circumvent this problem, additional checks are generated for both pointer value validity and alignment when an lvalue is used. For example:

```
void f(void)
{
  char c;
  short *p2;
  char *ptr = & c;
  memcpy(& p2,& ptr,sizeof(ptr));
  /*@ assert rte: pointer_value: \null == p2 || \object_pointer(p2); */
  /*@ assert rte: pointer_alignment: \aligned(p2,alignof(short)); */
  short *r2 = p2;
  return;
}
```

While no pointer conversion happens in the last instruction, RTE still generates annotations at this program point. Section 2.8.1 gives more details on this topic.

2.2.2 Access

Dereferencing a pointer is an undefined behavior if:

- the pointer has an invalid value for dereferencing: null pointer, misaligned address for the type of object pointed to, address of an object after the end of its lifetime (see ISO C23 6.5.4.2 § 4);

– the pointer points one past the last element of an array object: such a pointer has a valid value, but should not be dereferenced (ISO C23 6.5.7 § 9);
– the read access is done with a type that is not compatible with the effective type of the object [1] (ISO C23 6.5 § 6).

As explained in Section 2.8.1, the third item is currently not supported by Frama-C or RTE. Note that initialization might also cause problems, this aspect is specifically covered in Section 2.7.

Pointer alignment is handled like what we previously explained in Section 2.2.1. Validity for dereferencing is handled via the ACSL built-in predicate `\valid(p)`: `\valid(s)` (where `s` is a set of terms) holds if and only if dereferencing any $p \in s$ is safe (i.e. points to a safely allocated memory location). A distinction is made for read accesses, that generate `\valid_read(p)` assertions (the locations must be at least readable), and write accesses, for which `\valid(p)` annotations are emitted (the locations must be readable and writable).

Since an array subscripting `E1[E2]` is identical to `(*((E1) + (E2)))` (ISO C23 6.5.3.3 § 2), the "invalid access" undefined behaviors naturally extend to array indexing, and RTE will generate similar annotations. However, when the array is known, RTE attempts to generate simpler assertions. Typically, on an access `t[i]` where `t` has size `10`, RTE will generate two assertions `0 <= i` and `i < 10`, instead of `\valid(&t[i])`.

The kernel option `-safe-arrays` (or `-unsafe-arrays`) influences the annotations that are generated for an access to a multidimensional array, or to an array embedded in a struct. Option `-safe-arrays`, which is set by default in Frama-C, requires that all syntactic accesses to such an array remain in bound. Thus, if the field `t` of the struct `s` has size `10`, the access `s.t[i]` will generate an annotation `i < 10`, even if some fields exist after `t` in `s`.[2] Similarly, if `t` is declared as `int t[10][10]`, the access `t[i][j]` will generate assertions `0 <= i < 10` and `0 <= j < 10`, even though `t[0][12]` is also `t[1][2]`.

Finally, dereferencing a pointer to a function leads to the emission of a `\valid_function` predicate, to protect against a possibly invalid pointer (ISO C23 6.3.2.3 § 8). Those assertions are generated provided option `-rte-pointer-call` is set.

Example 2.6 *An example of RTE annotation generation for checking the validity of each memory access:*

```
int i;
unsigned int j;

int main(void)
{
  int *p;
  int tab[10];
  /*@ assert rte: mem_access: \valid(p); */
  *p = 3;
  /*@ assert rte: index_bound: 0 <= i; */
  /*@ assert rte: index_bound: i < 10; */
  /*@ assert rte: mem_access: \valid_read(p); */
  tab[i] = *p;
  /*@ assert rte: pointer_alignment: \aligned(p + 1,alignof(int)); */
  /*@ assert rte: mem_access: \valid(p + 1); */
  /*@ assert rte: index_bound: j < 10; */
  *(p + 1) = tab[j];
}
```

Note that in this example, some trivial annotations are optimized away.

[1] Also known as "strict-aliasing rule"

[2] Thus, by default, RTE is more stringent than the norm. Use option `-unsafe-arrays` if you want to allow code such as `s.t[12]` in the example above.

Example 2.7 *An example of memory access validity annotation generation for structured types, with options* `-safe-arrays` *and* `-rte-pointer-call` *set.*

```
struct S {
   int val;
   struct S *next;
};

struct C {
   struct S cell[5];
   int (*f)(int);
};

struct ArrayStruct {
   struct C data[10];
};

unsigned int i, j;

int main() {
  int a;
  struct ArrayStruct buff;
  // some code

  /*@ assert rte: index_bound: i < 10; */
  /*@ assert rte: index_bound: j < 5; */
  /*@ assert rte: mem_access: \valid_read(&(buff.data[i].cell[j].next)->val); */
  a = (buff.data[i].cell[j].next)->val;

  /*@ assert rte: index_bound: i < 10; */
  /*@ assert rte: function_pointer: \valid_function(buff.data[i].f); */
  (*(buff.data[i].f))(a);

  return 0;
}
```

Notice the annotation generated for the call `(*(buff.data[i].f))(a)`.

2.2.3 Downcasting pointers

Converting a pointer to an integer type is an undefined behavior when the result of the conversion cannot be represented in this integer type (ISO C23 6.3.2.3 § 6). An annotation is emitted when such a cast might lead to an undefined behavior.

Example 2.8

```
void f(void)
{
  int x = 0;
  int *y = & x;
  intptr_t z1 = (intptr_t)y;
  /*@ assert rte: pointer_downcast: (unsigned long)y <= 2147483647; */
  int z3 = (int)y;
  /*@ assert rte: pointer_downcast: (unsigned long)y <= 32767; */
  short z4 = (short)y;
  return;
}
```

Note that most of the time analyzers will be unable to prove these annotations that strongly rely on the runtime memory layout.

2.3 Unsigned overflow annotations

ISO C23 states that *unsigned* integer arithmetic is modular: overflows do not occur (ISO C23 6.2.5). On the other hand, most first-order solvers used in deductive verification (excluding dedicated bit-vector solvers such as [2]) either provide only non-modular arithmetic operators, or are much more efficient when no modulo operation is used besides classic full-precision arithmetic operators. Therefore, RTE offers a way to generate assertions preventing unsigned arithmetic operations to overflow (*i.e.* involving computation of a modulo).

Operations which are considered by RTE regarding unsigned overflows are addition, subtraction, multiplication. Negation (unary minus), left shift. and right shift are not considered. The generated assertion requires the result of the operation (in non-modular arithmetic) to be less than the maximal representable value of its type, and non-negative (for subtraction).

Example 2.9

The following file only contains unsigned arithmetic operations: no assertion is generated by RTE by default.

```
unsigned int f(unsigned int a, unsigned int b) {
  unsigned int x, y;
  x = a * (unsigned int)2;
  y = b - x;
  return y;
}
```

To generate assertions w.r.t. unsigned overflows, options `-warn-unsigned-overflow` *must be used. Here is the resulting file on a 32 bits target architecture (*`-machdep x86_32`*):*

```
unsigned int f(unsigned int a, unsigned int b) {
  unsigned int x, y;
  /*@ assert rte: unsigned_overflow: 0 <= a*(unsigned int)2; */
  /*@ assert rte: unsigned_overflow: a*(unsigned int)2 <= 4294967295; */
  x = a * (unsigned int)2;
  /*@ assert rte: unsigned_overflow: 0 <= b-x; */
  /*@ assert rte: unsigned_overflow: b-x <= 4294967295; */
  y = b - x;
  return y;
}
```

2.4 Unsigned downcast annotations

Downcasting an integer type to an unsigned type is a well-defined behavior, since the value is converted using a modulo operation just as for unsigned overflows (ISO C23 6.3.1.3 § 2). The RTE plug-in offers the possibility to generate assertions preventing such occurrences of modular operations with the `-warn-unsigned-downcast` option.

Example 2.10

On the following example, the sum of two `int` *is returned as an unsigned char:*

```
unsigned char f(int a, int b) {
  return a+b;
}
```

Using RTE with the `-warn-unsigned-downcast` *option gives the following result:*

```
unsigned char f(int a, int b) {
  unsigned char __retres;
  /*@ assert rte: unsigned_downcast: a+b <= 255; */
  /*@ assert rte: unsigned_downcast: 0 <= a+b; */
  /*@ assert rte: signed_overflow: -2147483648 <= a+b; */
  /*@ assert rte: signed_overflow: a+b <= 2147483647; */
  __retres = (unsigned char)(a + b);
  return (__retres);
}
```

2.5 Cast from floating-point to integer types

Casting a value from a real floating type to an integer type is allowed only if the value fits within the integer range (ISO C23 6.3.1.4), the conversion being done with a truncation towards zero semantics for the fractional part of the real floating value. The RTE plug-in generates annotations that ensure that no undefined behavior can occur on such casts.

```
int f(float v) {
  int i = (int)(v+3.0f);
  return i;
}
```

Using RTE with the `-rte-float-to-int` option, which is set by default, gives the following result:

```
int f(float v) {
  int i;
  /*@ assert rte: float_to_int: v+3.0f < 2147483648; */
  /*@ assert rte: float_to_int: -2147483649 < v+3.0f; */
  i = (int)(v + 3.0f);
  return i;
}
```

2.6 Expressions not considered by RTE

An expression which is the operand of a `sizeof` or `alignof` is ignored by RTE, as are all its sub-expressions. This is an approximation, since the operand of `sizeof` may sometimes be evaluated at runtime, for instance on variable sized arrays: see the example in ISO C23 6.5.4.4 § 8. Still, the transformation performed by Cil on the source code actually ends up with a statically evaluated `sizeof` (see the example below). Thus, the approximation performed by RTE seems to be on the safe side.

Example 2.11 *Initial source code:*

```
#include <stddef.h>

size_t fsize3(int n) {
  char b[n + 3]; // variable length array
  return sizeof b; // execution time sizeof
}
```

```
int main() {
  return fsize3(5);
}
```

Output obtained with `frama-c -print` *with* `gcc` *preprocessing:*

```
typedef unsigned long size_t;
/* compiler builtin:
   void *__builtin_alloca(unsigned int);   */
size_t fsize3(int n)
{
  size_t __retres;
  char *b;
  unsigned int __lengthofb;
  {
    /*undefined sequence*/
    __lengthofb = (unsigned int)(n + 3);
    b = (char *)__builtin_alloca(sizeof(*b) * __lengthofb);
  }
  __retres = (unsigned long)(sizeof(*b) * __lengthofb);
  return __retres;
}

int main(void)
{
  int __retres;
  size_t tmp;
  tmp = fsize3(5);
  __retres = (int)tmp;
  return __retres;
}
```

2.7 Initialization

Reading an uninitialized value can be an undefined behavior in the two following cases:

- ISO C23 6.3.2.1 § 1 a variable whose address is never taken is read before being initialized,
- a memory location that has never been initialized is read, and it happens that it was a non-value representation for the type used for the access.

More generally, reading an uninitialized location always results in an indeterminate representation (ISO C23 6.7.11). Such a representation is either an unspecified value or a non-value representation. Only reading a non-value representation is an undefined behavior (ISO C23 6.2.6.1 § 5). It corresponds to the second case above.

However, for (most) types that do not have non-value representation, reading an unspecified value is generally not a desirable behavior. Thus, RTE is stricter than the ISO C on many aspects and delegates one case of undefined behavior to the use of compiler warnings. We now detail the chosen tradeoff.

If a value of a fundamental type (integers, floating point types, pointers, or a typedef of those) is read, it must be initialized except if it is a formal parameter or a global variable. We exclude formal parameters as its initialization status must be checked at the call point (RTE generates an annotation for this). We exclude global variables as they are initialized by default and any value stored in this variable must be initialized (RTE generates an annotation for this).

As structures and unions never have non-value representation, they can (and they are regularly) be manipulated while being partially initialized. Consequently, RTE does not require initialization for

reads of a full union or structure (while reading fields with fundamental types is covered by the previous paragraph). As a consequence, the case of structures and unions *whose address is never taken, and being read before being initialized* is **not** covered by RTE. It is worth noting that this particular case is efficiently detected by a compiler warning (see `-Wuninitialized` on Gcc and Clang for example) as it only requires local reasoning that is easy for compilers (but not that simple to express in ACSL).

If you really need RTE to cover the previous case, please contact the Frama-C team.

Finally, there are some cases when reading uninitialized values via a type that cannot have non-value representation (for example `unsigned char`) should be allowed, for example writing a `memcpy` function. In this case, one can exclude this function from the range of function annotated with initialization properties by removing them from the set of functions to annotate (e.g. `-rte-initialized="@all,-f"`). Note that the excluded functions must preserve the initialization of global variables, as no assertions are generated for them.

2.8 Undefined behaviors not covered by RTE

One should be aware that RTE only covers a small subset of all possible undefined behaviors (see annex J.2 of [5] for a complete list).

In particular, undefined behaviors related to the following operations are not considered:

- Use of relational operators for the comparison of pointers that do not point to the same aggregate or union (ISO C23 6.5.9 § 6)
- Demotion of a real floating type to a smaller floating type producing a value out of the representable range (ISO C23 6.3.1.5 § 2)
- Read access is done with a type that is not compatible with the effective type of the object [3] (ISO C23 6.5 § 6).

2.8.1 Unsupported strict-aliasing rule

The main consequence of unsupported strict-aliasing rule is assertions redundancy. Basically, since it is currently impossible to generate an assertion when reading an lvalue with an effective type different from the stored one, we are forced to generate on lvalue usage all assertions that may be hidden by the uncaught strict-aliasing rule violation. We have already shown several cases of this along the manual.

The only optimization is related to local variables and function parameters, when such variables are read (without field/index accesses) while their address is not taken in the function, then RTE does not generate assertions since a strict aliasing violation cannot happen: there is no aliasing at all.

Example 2.12

```
void example(int *p, int *q)
{
  int *r = p;
  /*@ assert rte: pointer_alignment: \aligned(q,alignof(int)); */
  int *s = q;
  & q ; // the function takes the address of q
}
```

RTE generates an assertion for the lvalue `q` *because the address of* `q` *is taken in the function, while it is not the case for* `p`.

[3] Also known as "strict-aliasing rule"

Plug-in Options 3

Enabling RTE plug-in is done by adding `-rte` on the command-line of Frama-C. The plug-in then selects every C function which is in the set defined by the `-rte-select`: if no explicit set of functions is provided by the user, all C functions defined in the program are selected. Selecting the kind of annotations which will be generated is performed by using other RTE options:

`-rte` **(boolean, defaults to false)**
Enable RTE plug-in

`-rte-div` **(boolean, defaults to true)**
Generate annotations for division by zero

`-rte-float-to-int` **(boolean (defaults to true))**
Generate annotations for casts from floating-point to integer

`-rte-initialized` **(set of function (defaults to none))**
Generate annotations for initialization for the given set of functions

`-rte-mem` **(boolean (defaults to true))**
Generate annotations for validity of left-values access

`-rte-pointer-call` **(boolean (defaults to true))**
Generate annotations for validity of calls via function pointers

`-rte-shift` **(boolean (defaults to true))**
Generate annotations for left and right shift value out of bounds

`-rte-select` **(set of function (defaults to all))**
Run plug-in on a subset of C functions

`-rte-trivial-annotations` **(boolean (defaults to false))**
Generate all annotations even when they trivially hold

`-rte-warn` **(boolean (defaults to true))**
Emit warning on broken annotations

Combined with the following Kernel options:

`-warn-unsigned-overflow` **(boolean, defaults to false)**
Generate annotations for unsigned overflows

`-warn-unsigned-downcast` **(boolean, defaults to false)**
Generate annotations for unsigned integer downcast

`-warn-signed-overflow` **(boolean, defaults to true)**
Generate annotations for signed overflows

`-warn-signed-downcast` **(boolean, defaults to false)**
Generate annotations for signed integer downcast

`-warn-pointer-downcast` **(boolean, defaults to true)**
Generate annotations for downcast of pointer values

`-warn-left-shift-negative` **(boolean, defaults to true)**
Generate annotations for left shift on negative values

`-warn-right-shift-negative` **(boolean, defaults to false)**
Generate annotations for right shift on negative values

`-warn-invalid-bool` **(boolean, defaults to true)**
Generate annotations for `bool` non-value representations

`-warn-special-float` **(string:** `non-finite` **(default),** `nan` **or** `none`**)**
Generate annotations when special floats are produced: infinite floats or NaN (by default), only on NaN or never.

`-warn-invalid-pointer` **(boolean, defaults to false)**
Generate annotations for invalid pointer arithmetic[1]

`-warn-unaligned-pointer` **(boolean, defaults to true)**
Generate annotations for unaligned pointers.

Pretty-printing the output of RTE and relaunching the plug-in on the resulting file will generate duplicated annotations, since the plug-in does not check existing annotations before generation. This behavior does not happen if RTE is used in the context of a Frama-C project [6]: the annotations are not generated twice.

[1] For the rationale about this default value, please refer to the Frama-C user manual [3].

Bibliography

[1] Patrick Baudin, François Bobot, Loïc Correnson, Zaynah Dargaye, and Allan Blanchard. *WP plug-in Manual*. CEA List, Software Reliability Laboratory.

[2] Armin Biere. Boolector. https://boolector.github.io/.

[3] Loïc Correnson, Pascal Cuoq, Florent Kirchner, André Maroneze, Virgile Prevosto, Armand Puccetti, Julien Signoles, and Boris Yakobowski. *Frama-C User Manual*. CEA List, Software Reliability Laboratory. https://www.frama-c.com/download/frama-c-user-manual.pdf.

[4] Pascal Cuoq David Bühler, Valentin Perrelle Boris Yakobowski. With Matthieu Lemerre, André Maroneze, and Virgile Prevosto. *The Eva plug-in*. CEA List, Software Reliability Laboratory. https://www.frama-c.com/download/frama-c-eva-manual.pdf.

[5] International Organization for Standardization (ISO). *The ANSI C standard (C23)*. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3220.pdf.

[6] Julien Signoles with Thibaut Antignac, Loïc Correnson, Matthieu Lemerre and Virgile Prevosto. *Plug-in Development Guide*. CEA List, Software Reliability Laboratory.