





Frama-C's annotation generator plug-in

for Frama-C Carbon-20110201

Philippe Herrmann

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



Contents

1	Introduction	7
1.1	RTE plugin	7
1.2	Runtime errors	8
1.3	Other annotations generated	9
2	Runtime error annotation generation	11
2.1	Integer operations	11
2.1.1	Addition, subtraction, multiplication	11
2.1.2	Signed downcasting	12
2.1.3	Unary minus	13
2.1.4	Division and modulo	13
2.1.5	Bitwise shift operators	14
2.2	Left-values access	15
2.3	Unsigned overflow annotations	16
2.4	Expressions not considered by RTE	17
2.5	Undefined behaviors not covered by RTE	18
3	Precondition annotation generation	19
4	Plugin options	21
	Bibliography	23



Chapter 1

Introduction

1.1 RTE plugin

This document is a reference manual for the annotation generator plugin called RTE. The aim of the RTE plugin 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 ;
- precondition checking (*requires* and *assumes* clauses) at function's call sites, as well as postconditions (*ensures* clauses) and *assigns*, for functions having an ACSL specification.

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 [5]) 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 [6] 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 plugins, 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 [3] will be referred to as ISO C99 (of which specific paragraphs are cited, such as 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 at an out-of-bounds 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 [6], 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 C99 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 C99 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 plugins may wish to avoid them.
- generating call sites statement contracts, based on the called function's contract. This is useful for modular verification.



Runtime error annotation generation

2.1 Integer operations

According to 6.2.5.9, 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	<code>[SCHAR_MIN, SCHAR_MAX]</code>
signed short	<code>[SHRT_MIN, SHRT_MAX]</code>
signed int	<code>[INT_MIN, INT_MAX]</code>
signed long int	<code>[LONG_MIN, LONG_MAX]</code>
signed long long int	<code>[LLONG_MIN, LLONG_MAX]</code>

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:

type	size in bits	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 plugins such as RTE may not contain macros since pre-processing 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:*

```

1 |
2 | int main(void)
3 | {
4 |     signed long int lx, ly, lz;
5 |
6 |     /*@ assert
7 |         ((lx*ly <= 2147483647) &&
8 |         (lx*ly >= -2147483648));
9 |     */
10 |    lz = lx * ly;
11 |
12 |    return 1;
13 | }
```

The same program, but now annotated with an x86_64 machine dependency:

```

1 |
2 | int main(void)
3 | {
4 |     signed long int lx, ly, lz;
5 |
6 |     /*@ assert
7 |         ((lx*ly <= 9223372036854775807) &&
8 |         (lx*ly >= -9223372036854775808));
9 |     */
10 |    lz = lx * ly;
11 |
12 |    return 1;
13 | }
```

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:

```

1 | int main(void)
2 | {
3 |     signed char cx, cy, cz;
4 |
5 |     cz = cx + cy;
6 |     return 1;
7 | }

```

is in fact equivalent to:

```

1 | int main(void)
2 | {
3 |     signed char cx, cy, cz;
4 |
5 |     cz = (signed char)((int)cx + (int)cy);
6 |     return 1;
7 | }

```

Since a signed overflow can occur on expression `(int)cx + (int)cy`, the following annotation is generated by the RTE plugin:

```
| assert (((int )cx+(int )cy <= 2147483647) && ((int )cx+(int )cy >= -2147483648))
```

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

```
| assert (((int )cx+(int )cy <= 127) && ((int )cx+(int )cy >= -128))
```

Actually the RTE plugin infers this second (stronger) assertion 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 ISO C99 paragraph 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 plugin 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. The RTE plugin makes the choice of generating an annotation only for the strongest one. The selection of options `-rte-no-all -rte-signed` can be used in order to prevent signed downcasting annotation while keeping signed overflow annotation. In that case, the signed overflow annotation is generated.

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:

```

1 | int ix;
2 | // some code
3 | //@ assert (ix != -2147483648);
4 | ix = - ix;

```

2.1.4 Division and modulo

As of ISO C99 paragraph 6.5.5, 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.

```

1 | unsigned int ux;
2 | // some code
3 | //@ assert (ux != 0);
4 | 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.

```

1 | int x,y,z;
2 | // some code
3 | //@ assert (x != 0);
4 | //@ assert (!((x = -1) && (y = -2147483648)));
5 | z = y / x;
```

2.1.5 Bitwise shift operators

ISO C99 paragraph 6.5.7 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:

```

1 | int x,y,z;
2 |
3 | //@ assert ((y >= 0) && (y < 32));
4 | z = x << y; // same annotation for z = x >> y;
```

- in $E1 \ll E2$, $E1$ has signed type and negative value:

```

1 | int x,y,z;
2 |
3 | //@ assert (x >= 0);
4 | z = x << y;
```

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

```

1 | int x,y,z;
2 |
3 | //@ assert (x << y <= 2147483647);
4 | z = x << y;
```

There is also an implementation defined behavior if in $E1 \gg E2$, $E1$ 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.

```

1 | int x,y,z;
2 |
3 | //@ assert (x >= 0);
4 | z = x << y;
```

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

```

1 | long x,y,z;
2 |
3 | //@ assert ((y >= 0) && (y < 64));
4 | //@ assert (x<<y <= 9223372036854775807);
5 | //@ assert (x >= 0);
6 | z = x << y;
7 |
8 | //@ assert ((y >= 0) && (y < 64));
9 | //@ assert (x >= 0);
10 | z = x >> y;
```

2.2 Left-values access

Dereferencing a pointer is an undefined behavior if:

- the pointer has an invalid value: null pointer, misaligned address for the type of object pointed to, address of an object after the end of its lifetime (see ISO C99 paragraph 6.5.3.2.4) ;
- 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 C99 paragraph 6.5.6.8).

Since an array subscripting $E1[E2]$ is identical to $((*(E1) + (E2)))$ (ISO C99 paragraph 6.5.2.1.2), this “invalid access” undefined behavior naturally extends to array indexing.

The RTE plugin generates annotations to prevent this type of undefined behavior in a systematic way. It does so by deferring the check to 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).

Pointers to functions are not treated by RTE since, as of the time this manual was written, there is no predicate in ACSL expressing that such a pointer actually points to an existing function.

Example 2.3 *An example of RTE annotation generation for checking the validity of each memory access:*

```

1 | extern void f(int* p);
2 |
3 | int main(void)
4 | {
5 |     int *p ;
6 |     int tab[10] ;
7 |
8 |     //@ assert (\valid(p));
9 |     *p = 3;
10 |
11 |     //@ assert (\valid(&tab[3]));
12 |     //@ assert (\valid(p));
13 |     tab[3] = *p;
14 |
15 |     //@ assert (rte: \valid(p+1));
16 |     //@ assert (rte: \valid((int *)tab));
17 |     *(p + 1) = tab[0];
18 |
19 |     //@ assert (\valid((int *)tab));
20 |     f(tab); // equivalent to f(&tab[0])
21 |
22 |     return 0;

```

Note that in the call `f(tab)`, the implicit conversion from array `tab` to a pointer to the beginning of the array `&tab[0]` introduces a pointer dereferencing and thus the annotation `\valid((int) tab)`, which is equivalent to `\valid(&tab[0])`.*

Example 2.4 *An example of memory access validity annotation generation for structured types.*

```

1 | struct S {
2 |     int val;
3 |     struct S *next;
4 | };

```

```

5
6 struct C {
7     struct S cell[5];
8     int (*f)(int);
9 };
10
11 struct ArrayStruct {
12     struct C data[10];
13 };
14
15 int main()
16 {
17     int a;
18     struct ArrayStruct buff;
19     // some code
20
21     //@ assert (\valid(buff.data[1].cell[2].next));
22     //@ assert (\valid(&buff.data[1]));
23     //@ assert (\valid(&buff.data[1].cell[2]));
24     a = (buff.data[1].cell[2].next)->val;
25
26     //@ assert (\valid(&buff.data[0]));
27     (*(buff.data[0].f))(a);
28
29     return 0;
30 }

```

There is no annotation generated for the validity of the field `buff.data[0].f` because it is a function pointer. RTE emits a warning in such a case.

2.3 Unsigned overflow annotations

ISO C99 states that *unsigned* integer arithmetic is modular: overflows do not occur (paragraph 6.2.5.9 of ISO C99). 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, and left shift. Negation (unary minus) 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 nonnegative (for subtraction).

Proviso: assertion generation for unsigned overflows is only available for unsigned integer types whose bit size is at most 32. In particular, in order to be able to treat 64 bits integers, it would be necessary to use big constants such as $2^{64} - 1$ in generated assertions, which is a problem for Cil [1] at the time of writing this manual.

Example 2.5

The following file only contains unsigned arithmetic operations: no assertion is generated by RTE by using options `-rte -rte-all`.

```

1 unsigned int f(unsigned int a, unsigned int b)
2 {
3     unsigned int x, y, z;
4     x = a << 3;
5     y = b * (unsigned int )2;
6     z = x - y;
7     return (z);
8 }

```


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

```

1 unsigned int f(unsigned int a, unsigned int b)
2 {
3     unsigned int x, y, z;
4     /*@ assert rte: (a<<3 <= 4294967295); */
5     x = a << 3;
6     /*@ assert rte: (b*(unsigned int )2 <= 4294967295); */
7     y = b * (unsigned int )2;
8     /*@ assert rte: (x-y >= 0); */
9     z = x - y;
10    return (z);
11 }
12

```

2.4 Expressions not considered by RTE

An expression which is the operand of a `sizeof` (or `__alignof`, a GCC operator parsed by Cil) 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 C99 paragraph 6.5.3.4.7. 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 2.6 *Initial source code:*

```

1 #include <stddef.h>
2
3 size_t fsize3(int n)
4 {
5     char b[n + 3]; // variable length array
6     return sizeof b; // execution time sizeof
7 }
8
9 int main()
10 {
11     return fsize3(5);
12 }

```

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

```

1 typedef unsigned int size_t;
2 /* compiler builtin:
3     void *__builtin_alloca(unsigned int ) ;    */
4 size_t fsize3(int n )
5 {
6     char *b ;
7     unsigned int __lengthofb ;
8     size_t __retres ;
9     { /*undefined sequence*/
10         __lengthofb = (unsigned int )(n + 3);
11         b = (char *)__builtin_alloca(sizeof(*b) * __lengthofb);
12     }
13     __retres = sizeof(*b) * __lengthofb;
14     return (__retres);
15 }
16
17 int main(void)
18 {
19     int tmp ;
20     tmp = (int )fsize3(5);
21     return (tmp);
22 }

```

2.5 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 [3] 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 C99 6.5.8)
- Demotion of a real floating type to another type producing a value outside of the representable range (ISO C99 6.3.1.5)
- Conversion between two pointer types produces a result that is incorrectly aligned (ISO C99 6.3.2.3)
- Use of a pointer to call a function whose type is not compatible with the pointed-to type (ISO C99 6.3.2.3)

Chapter 3

Precondition annotation generation

Using ACSL, the Frama-C framework allows the user to write contracts for C functions. These contracts contain in particular:

- **requires** clauses R_1, R_2, \dots (preconditions),
- **ensures** clauses E_1, E_2, \dots (postconditions),
- **assigns** clauses A_1, A_2, \dots (set of locations assigned by the function).

The intended behavior is that any caller must make sure that the function is called in a state where the required property $R_1 \ \&\& \ R_2 \ \&\& \dots$ holds, to ensure that $E_1 \ \&\& \ E_2 \ \&\& \dots$ holds in the state returned by the function call. Contracts may contain several named behaviors as well, see the ACSL manual.

For a function having a contract, the RTE plugin can generate a statement annotation corresponding to the contract at each call site. It does so by substituting formal parameters of the contract with actual parameters at the call site. The generated statement annotation ensures that the function is called in the proper state and specifies its return state.

Example 3.1

Consider function `f` with the following contract:

```
1  /*@ ensures (\result == -\at(x,0ld));
2     behavior pos:
3         assumes (x >= 0);
4         ensures (\result <= 0);
5         assigns *y;
6     behavior neg:
7         assumes (x < 0);
8         ensures (\result > 0);
9         assigns \nothing;
10 */
11 int f(int x , int *y )
12 {
13     int __retres ;
14     if (x >= 0) { *y = x; }
15     __retres = - x;
16     return (__retres);
17 }
```

Here is an example of a statement behavior generated by RTE on a call to `f`:

```

1  int main(void)
2  {
3      int a, b ,c;
4      a = 5;
5      /*@ behavior rte_0:
6          ensures (b == -\at(a,0ld));
7          assigns b;
8          behavior rte_1:
9              assumes (a >= 0);
10             ensures (b <= 0);
11             assigns b, *(&c);
12             behavior rte_2:
13                 assumes (a < 0);
14                 ensures (b > 0);
15                 assigns b;
16         */
17     b = f(a,& c);
18     return (b + c);
19 }

```

The generated behaviors names are `rte_0`, `rte_1`, `rte_2`). Notice that `assigns` clauses are also taken into account, and that the formal `\result` is substituted with the assigned left-value.

The main restriction is that the RTE plugin only performs syntactic detection of function calls: functions called through pointers are warned about but no statement annotation is generated. This is unlikely to change until ACSL introduces contracts on function pointers.

Chapter 4

Plugin options

Enabling RTE plugin is done by adding `-rte` on the command-line of Frama-C. The plugin 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 (see fig. 4.1 for a summary).

Pretty-printing the output of RTE and relaunching the plugin on the resulting file will generate duplicated annotations, since the plugin does not check existing annotations before generation. This behaviour does not happen if RTE is used in the context of a Frama-C project [4]: in that case, the set of RTE options used for each C function will be recorded in internal states. Thus a function selected for annotation will only be annotated by RTE if at least one generation option has changed. If the function is to be annotated with a different set of options, it occurs after discarding existing annotations generated by RTE. This ensures that, for a given C function and in the context of a Frama-C project, the generated annotations only depend on the set of options that have been used the last time RTE has been launched with this C function selected. It is possible to annotate each C function with a different set of options.

Option `-rte-all` has a special behavior: if selected (which is the default case), the options `-rte-signed`, `-rte-downcast`, `-rte-div`, `-rte-mem` and `-rte-precond` are also selected, irrespective of the corresponding command-line option. For instance, annotations for division by zero will be generated even if the user explicitly specifies `-rte-no-div`, *unless* `-rte-no-all` is also selected.

The special behavior of `-rte-all` implies that RTE generates by default all runtime-errors (more precisely, undefined and implementation-defined behaviors) and precondition annotations it handles. The user should explicitly add `-rte-unsigned-ov` (not entailed by `-rte-all`) to generated unsigned overflows annotations, which are valid as of ISO C99.

On the other hand, to generate only a subset of possible annotations, one has to use `-rte-no-all` in conjunction with other positive options. For instance, used in conjunction with `-rte-precond` alone, only precondition annotations would be generated. Adding `-rte-mem`, annotations for the validity of memory access would also be generated.

Some examples:

- `frama-c -rte -rte-select f,g -rte-no-all -rte-precond`: only generate precondition annotations, and only for call-sites found in functions `f` and `g`.
- `frama-c -rte -rte-no-all -rte-unsigned-ov`: only generated annotations for unsigned overflows, for the whole C program.

Option	Type (Default)	Description
<code>-rte</code>	boolean (false)	Enable RTE plugin
<code>-rte-all</code>	boolean (true)	Enable all runtime-errors annotations, except for unsigned overflows (supersedes all <code>-rte-no</code>)
<code>-rte-print</code>	boolean (false)	Pretty print the annotated code
<code>-rte-unsigned-ov</code>	boolean (false)	Generate annotations for unsigned overflows (not entailed by <code>-rte-all</code>)
<code>-rte-signed</code>	boolean (false)	Generate annotations for signed overflows
<code>-rte-downcast</code>	boolean (false)	Generate annotations for signed integer downcast
<code>-rte-div</code>	boolean (false)	Generate annotations for division by zero
<code>-rte-mem</code>	boolean (false)	Generate annotations for validity of left-values access
<code>-rte-const</code>	boolean (true)	Generate status for annotation through constant folding
<code>-rte-precond</code>	boolean (false)	Generate contract-based statement behaviors based at call sites
<code>-rte-warn</code>	boolean (true)	Emit warning on broken annotations
<code>-rte-select</code>	set of function (all)	Run plugin on a subset of C functions

Table 4.1: RTE options

- `frama-c -rte -rte-unsigned-ov -rte-no-const`: generate all possible annotations (unsigned overflows included), but do not try to evaluate their status through constant folding.

Note that `-rte-print` is almost equivalent to `-print`: the only difference is that in the former case, the resulting C code is pretty printed only if `-rte` is enabled.

Bibliography

- [1] Berkeley University. The C Intermediate Language (CIL) library. <http://manju.cs.berkeley.edu/cil/>.
- [2] Armin Biere. Boolector. <http://fmv.jku.at/boolector/>.
- [3] International Organization for Standardization (ISO). *The ANSI C standard (C99)*. <http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf>.
- [4] Julien Signoles with Loïc Correnson and Virgile Prevosto. *Plug-in Development Guide*. CEA List, Software Reliability Laboratory.
- [5] Loïc Correnson, Zaynah Dargaye, Anne Pacalet. *WP plug-in Manual*. CEA List, Software Reliability Laboratory.
- [6] Pascal Cuoq with Virgile Prevosto. *Frama-C's value analysis plug-in*. CEA List, Software Reliability Laboratory. <http://frama-c.com/download/frama-c-value-analysis.pdf>.