

Part I

E-ACSL

long ra
for 0 <=
C1); if (m
tmp2 =
of the

tmp2[j] = 0; if (i <= (Nbr - 1)) else if (tmp1[j] >= 0) { i <= (Nbr - 1); tmp2[j] = (i <= (Nbr - 1) ? 1 : 0); else tmp2[j] = tmp1[j]; } /* Then the second part looks like the first one: */
tmp1[i][k] = 0; k <= 5; k++) tmp1[i][k] += m2[i][k] * tmp2[k]; } /* The [i,j] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1*MC1
i <= 3; tmp1[i][i] >= 1; */ Final rounding: tmp2[i][i] is now represented on 9 bits: *if (tmp1[i][i] < -256) m2[i][i] = -256; else if (tmp1[i][i] > 255) m2[i][i] = 255; else m2[i][i] = tmp1[i][i];



Executable Ansi/ISO C Specification Language

What should be?

- ▶ executable subset of ACSL
- ▶ preserve ACSL semantics as much as possible
- ▶ compatible with ALFA as much as possible

Which goals?

- ▶ runtime assertion checking
- ▶ usable by dynamic analyses tools
- ▶ usable by static verification tools like Frama-C plug-ins
- ▶ verification of mixed ADA/C programs



- ▶ last version: 1.5-3
- ▶ [Hi-Lite deliverable 3.4.1](#)
- ▶ based on ACSL v1.5
- ▶ detailed syntax
- ▶ mainly point out differences with ACSL



long n...
 for 0 <...
 C1) if m...
 tmp2...
 re of th...
 tmp2[0] = 1 << (n-1) else if (tmp1[0]) >= 1 << (n-1) - 1; else tmp2[0] = tmp1[0]; /* Then the second part looks like the first one. */
 tmp1[0] = 0; k = 5; k--> tmp1[0] = m2[0][k] * tmp2[0][k]; /* The [i][j] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1*MC1...
 l = 1; tmp1[0] >= 1; /* Final rounding: tmp2[0] is now represented on 9 bits. *if (tmp1[0] < -256) m2[0] = -256; else if (tmp1[0] > 255) m2[0] = 255; else m2[0] = t...

- ▶ similar to ACSL (e.g. mathematical integer arithmetic)
- ▶ 3-valued logic with **undefinedness**
 - ▶ $1/0 == 1/0$ is undefined
 - ▶ $f(*p)$ is undefined if p is invalid
 - ▶ tools must ensure that undefined terms are never evaluated
- ▶ **lazy operators** `&&`, `||`, `_?_:_`, `==>`
 - ▶ $\backslash\text{false} \ \&\& \ 1/0 == 1/0$ is invalid
 - ▶ $1/0 == 1/0 \ \&\& \ \backslash\text{false}$ is undefined
 - ▶ different but consistent semantics compared to ACSL
 - ▶ a valid (resp. invalid) E-ACSL predicate is valid (resp. invalid) in ACSL



- ▶ 3 different kinds of quantifications (only 1 in ACSL)
- ▶ unguarded quantification *à la* ACSL only allowed for boolean and char
- ▶ guarded integer quantification

```
\forall typ x1, ..., xn;
  a1 <= x1 <= b1 ... && an <= xn <= bn
  ==> p
```

- ▶ guarded **iterator** quantification
 - ▶ from which element does the iteration begin?
 - ▶ how to access to the next elements?
 - ▶ which guards must be true to continue to iterate?
 - ▶ the only syntax extension from ACSL



```

struct btree {
    int val;
    struct btree *left, *right;
};

/*@ iterator access(_, struct btree *t):
    @ nexts t→left, t→right;
    @ guards \valid(t→left),
           \valid(t→right); */

/*@ predicate is_even(struct btree *t) =
    @ \forallall struct btree *tt;
       access(tt, t) ==> tt→val % 2 == 0; */

```



- ▶ lose their inductive nature
- ▶ a loop invariant I is equivalent to
 - ▶ put an **assertion I** just before entering the loop
 - ▶ put the same **assertion** at the very end of the loop body

long n
for (i = 0; i < n; i++)
c[i] = 0;
tmp2 = 0;
for (k = 0; k < n; k++)
tmp1[k] = 0;
for (j = 0; j < n; j++)
for (i = 0; i < n; i++)
c[i] = c[i] + m[i][j] * tmp1[j];
tmp1[j] = 0;
tmp2 = tmp2 + c[j];
return tmp2;

tmp2[j] = 0; for (i = 0; i < n; i++) tmp1[i] = 0; for (i = 0; i < n; i++) for (j = 0; j < n; j++) c[i] = c[i] + m[i][j] * tmp1[j]; tmp1[j] = 0; tmp2 = tmp2 + c[j]; return tmp2; Then the second part looks like the first one. tmp1[k] = 0; k = 0; k < n; k++) tmp1[k] += m[0][k] * tmp2[k]; // The [0] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1 * MC1. l = 1; tmp1[l] += ... Final rounding: tmp2[l] is now represented on 9 bits. *M (tmp1[l] < -256) m2[l] = -256; else if (tmp1[l] > 255) m2[l] = 255; else m2[l] = tmp1[l];



Present:

- ▶ recursive logic definitions
- ▶ specification modules

Absent:

- ▶ lemmas and axiomatic (not computable)
- ▶ inductive predicate (not computable in general)
- ▶ polymorphism and higher order (still experimental in ACSL)
- ▶ concrete logic type (still experimental in ACSL)
- ▶ memory footprint (still experimental in ACSL)



Part II

Future Frama-C Plug-in

long ra
for 0 =>
C1) if (m
tmp2 =
of the

tmp2[j] = 0; for (k = 0; k < (Nb1 - 1); k++) tmp2[j] += m2[k][k] * tmp1[k][j]; /* The [j] coefficient of the matrix product MC2*TMP1, that is, *MC2*(TMP1) = MC2*(M1)*M1 = MC2*(M1)*M1 - 1. tmp1[j][i] >= -1.*/ Final rounding: tmp2[j] is now represented on 9 bits: *if (tmp1[j][i] < -256) m2[j][i] = -256; else if (tmp1[j][i] > 255) m2[j][i] = 255; else m2[j][i] = tmp1[j][i];



- ▶ new Frama-C plug-in called 'E-ACSL'
- ▶ takes a C program annotated with ACSL as input
- ▶ **checks** that annotations are part of E-ACSL
- ▶ roughly **converts annotations**

```
int div(int x, int y) {
    /*@ assert y != 0; */
    return x / y;
}
```

into C code

```
int div(int x, int y) {
    /*@ assert y != 0; */
    if (y == 0) e_acsl_fail();
    return x / y;
}
```



- ▶ E-ACSL integers are mathematical integers
- ▶ heavy translation via **GMP** (could be optimized)

```

/*@ assert -3 == x; */ ;
// declare temp variables
mpz_t tmp1, tmp2, tmp3, int tmp 4;
mpz_init_set_si(tmp1,3); // init tmp1 with 3
mpz_init(tmp2); // init tmp2
mpz_neg(tmp2, tmp1); // tmp2 = -tmp1 = -3
mpz_init_set_si(tmp3, x); // init tmp3 with x
// really check the assertion by comparing -3 to x
tmp4 = mpz_cmp(tmp2, tmp3);
if (tmp4 != 0) e_acsl_fail("(-3 == x)");
// deallocate temp variables
mpz_clear(tmp1); mpz_clear(tmp2); mpz_clear(tmp3);

```



in Hi-Lite, only expect to handle a big-enough subset of E-ACSL

unsupported features:

- ▶ floats and reals
- ▶ constructs for memory management like `\valid` or `\at`

partially supported features:

- ▶ memory accesses: will not check that they are valid

```

long n;
for (i = 0; i < n; i++)
  tmp2 =
  of the

```

```

tmp2[i] = 1 + (n-i) * i; // also #tmp1[i] >= 1 + (n-i) * i; // #tmp2[i] = tmp1[i]; // Then the second part looks like the first one
tmp1[i] = 0; k = 5; k--; tmp1[i] = mc2[i][k] * tmp2[k]; // The [i][j] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1 *MC1
i = 1; tmp1[i] >= 1; // Final rounding: tmp2[i] is now represented on 9 bits. #if (tmp1[i] < -255) m2[i] = -255; else #if (tmp1[i] > 255) m2[i] = 255; else #endif

```



- ▶ yet preliminary development
- ▶ first version planed **to be released in September/October**
- ▶ not possible to implement the whole stuff
- ▶ **need your feedback** to know what E-ACSL features you are going to use
- ▶ will implement required features before others



long n...
 for 0...
 C1) if...
 tmp2...
 of the...
 tmp2[0] = 1; for (k=0; k<n; k++) tmp1[k] += m2[0][k] * tmp2[k]; /* The [0] coefficient of the matrix product MC2*TMP2, that is *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1*MC1...
 tmp1[0] = 0; k = 5; k++) tmp1[k] += m2[0][k] * tmp2[k]; /* The [k] coefficient of the matrix product MC2*TMP2, that is *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1*MC1...
 l = 1; tmp1[0] += 1; /* Final rounding: tmp2[0] is now represented on 9 bits. if (tmp1[0] < -255) tmp1[0] = -255; else if (tmp1[0] > 255) tmp1[0] = 255; else tmp1[0] = tmp1[0];

Any questions?

long ra
for 0 ->
C1); if (m
tmp2 =
se of the

tmp2[ji] = 0; if (i < (Nbr - 1)) else if (tmp1[ji] >= 0) tmp2[ji] = (1 << (Nbr - 1)) - 1; else tmp2[ji] = tmp1[ji]; /* Then the second part takes like the first one: */
tmp1[0][k] = 0; k = 0; k <= 5; k++) tmp1[i][k] += m2[0][k] * tmp2[k][j]; /* The [i][j] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(M1*M1) = MC2*M1*M1
i = 1; tmp1[0][i] >= -1; /* Final rounding: tmp2[0][i] is now represented on 9 bits: *if (tmp1[0][i] < -256) m2[0][i] = -256; else if (tmp1[0][i] > 255) m2[0][i] = 255; else m2[0][i] = tmp1[0][i];

