C/C++ code analysis with TIS Analyzer: some challenges for deductive verification

2018-11-23

Presentation

1. Introduction
2. Memory model needs
3. Memory Model and Side Effects
4. Arithmetic
5. C/C++ Language support
6. Conclusion

TRUST IN SOFT

Introduction

TrustInSoft's ProofInUse goals related to deductive verification

Our methodology:

- Prove the absence of undefined behavior by Value analysis
- Use counter-example
  - to help the user understand the generated alarms
  - to eliminate false-alarms
- Write annotations to understand which properties hold where
- Write annotations to help the value analysis
  - mainly relational properties
  - also properties on ranges (validity, initialization, ...)
- Functional proof for very specific use-cases: basic libraries, small pieces of code within a very large code base

Collaboration between WP and Value analysis

```c
extern int T[100];
//@ requires val: n < 100;
void need_relation (unsigned int x, unsigned int n) {
  if (x <= n) {
    unsigned int i = n - x;
    //@ assert wp: i < 100;
    T[i] = 0;
  }
};
int main (...) { ... };
```

- precondition checked by Value
- needed by WP to check the assertion
- used by Value to ensure the memory access validity

Memory model needs

Missing separation hypothesis

The separation hypotheses are easily forgotten:

```
/*@ requires hyp: \valid(x) && \valid(y);
  @ ensures wp: \result == \old(*y); */
int separation (int *x, int *y) {
    *x = *y;
    *y = 3;
    return *x;
}
```

- Missing hypothesis: `requires \separated (x, y);`
- Difficult to find in complex proof obligations
- A readable counter example would be useful!

Proving Initialization

When there is a relation between a pointer and a size,

WP is often needed to prove properties about them:

```
void init (char * p, int n) {
  for (int i = 0; i < n; i++)
    *(p+i) = i;
  //@ assert wp2: \initialized (p + (0 .. n-1));
}
```

Memory model needs to support the notion of `\initialized` memory

Harder counter-example: missing hypothesis

```
/*@ requires \valid (p + (0..n-1));
  @ ensures wp: p[0] == 0; */
void reset (char * p, unsigned int n) {
  char * pi = p;
  /*@ loop assigns i, pi, p[0..n-1];
    @ loop invariant wp: pi == p + i;
    @ loop invariant wp:
           \forall integer k; 0 <= k < i ==> p[k] == 0; */
  for (unsigned int i = 0; i < n; i++, pi++)
    *pi = 0;
}
```

- Missing: `requires n != 0;`
- Another fix could be: `ensures wp: n == 0 || p[0] == 0;`
- Which counter-examples can we generate?

Function-local `(void*) -> (char*)` pointer

The previous example would often be written using `void*` pointer:

```
/*@ requires n != 0 && \valid (((char*)p) + (0..n-1));
  @ ensures wp: ((char*)p)[0] == 0; */
void reset (void * p, unsigned int n) {
  char * pi = (char*)p;
  /*@ loop assigns i, pi, ((char*)p)[0..n-1];
    @ loop invariant wp: pi == ((char*)p) + i;
    @ loop invariant wp:
      \forall integer k; 0 <= k < i ==> ((char*)p)[k] == 0; */
  for (unsigned int i = 0; i < n; i++, pi++)
    *pi = 0;
}
```

Memory model should support this.

(void*) pointer for memcpy callers

```
typedef unsigned long size_t;
/*@ assigns \result, ((char*)dest)[0..n-1];
  @ ensures hyp: \initialized (((char*)dest)+ (0 .. n-1)); */
void *memcpy (void *dest, const void *src, size_t n);

typedef struct { int a; int b; int c; } data;
/*@ requires valid_p: \valid_read (buf + (0 .. len-1));
  @ requires init: \initialized (buf + (0 .. len-1));
  @ ensures wp2: \result == -1 || \initialized (info); */
int use_memcpy (const char * buf, size_t len, data * info) {
  size_t sz = sizeof (*info);
  if (len < sz) return -1;
  memcpy (info, buf, sz);
  return sz;
}
```

Needed to be able to use library functions

Unions (with or without pointers)

```
typedef struct {
          int kind;
          union { int i; char c; long l; } value;
         } data;
void union_access (data * d) {
  switch (d->kind) {
    case 1: d->value.i = 3; break;
    case 2: d->value.c = 3; break;
    case 3: d->value.l = 3; break;
  }
  //@ assert wrong: wp: d->value.i == 3;
  //@ assert wp: d->kind == 1 ==> d->value.i == 3;
}
```

Expected counter-example: `d->kind = 42`

Heterogeneous access is not very common.

Memory Model and Side Effects

Side effects: the problem

- often need to move a property from one point to another one
- propagation through calls and loops which do not interfere
- guess (absence of) side effects as much as possible
- help the user to find missing hypotheses

Side effect: relying on scopes

Much information can be deduced from the scopes:

Since `p` is valid, it cannot point to the `x` or `y` local variables:

```
//@ requires hyp: \valid (p); ensures e_p: wp: *p == 4;
void local_scope (int * p) {
  int x = 10;
  int y = 20;
  *p = 4;
  x++; //@ assert a_x: wp: x == 11; // because \separated (p, &x);
  y++; //@ assert a_y: wp: y == 21; // because \separated (p, &y);
  p = &x; //@ assert a_p: wp: *p == 11;
}
```

Moreover, the local `p` is different from the post-condition parameter.

Memory model needs to reflect these scoping issues

Side effect: without specification

In some occasion, no specification is needed:

```
int f (int, int);

void skip_with_no_effect (int x, int y) {
  if (x < y) {
    int z = f (x, y);
    //@ assert wp: x < y;
  }
}
```

Whatever `f` is doing, it cannot change `x` or `y`.

Side effect: using function assigns properties

```
int compute (int * p, int * q);

void skip_with_assigns_call (int x, int y) {
  if (x < y) {
    int z = compute (&x, &y);
    //@ assert wp: x < y;
  }
}
```

The `compute` function may have changed `x` or `y`.

Can we generate a readable counter-example?

Side effect: using loop assigns properties

Similar to assigns

Arithmetic

Integer signs

- casts between signed and unsigned number are often used;
  - signed: check overflow;
  - unsigned: modulo arithmetic;

```
void sign_vs_unsigned (int x, unsigned int u) {
  unsigned int u2 = u + 1;
  //@ assert wrong: wp: u2 > u; // missing hyp
  int x2 = x + 1;
  //@ assert wp: x2 > x; // OK
  unsigned int ux = (unsigned int)x;
  //@ assert wrong: wp: x == ux; // missing hyp
  int y = (int)u;
  //@ assert wrong: wp: y == u; // missing hyp
}
```

TRUST IN SOFT

Integer sizes

- casts between different integer sizes are also very often used
  - especially implicit ones on function calls
  - need bound checking.

```
//@ assigns \result \from s; ensures \result == s + 1;
short incr (short s);
//@ ensures wp: \result == n + 1; // missing hypothesis
int int_cast (int n) {
  return incr (n);
}
```

TRUST IN SOFT

C/C++ Language support

Supporting `goto` statements

- Error management
- `break` / `continue` / `return`,
- Logical conjunctions/disjunctions

```
#define CHECK(f) do { if (( ret = f) != 0) goto cleanup; } while(0)
//@ assigns \result \from n; ensures n < 10 ==> \result == 1;
int compute (int n);
int goto_on_error (int n) {
  int ret;
  CHECK(compute (n));
  //@ assert wp: n >= 10;
  ret = 0;
cleanup:
  return( ret );
}
```

TRUST IN SOFT

Generate jumps inside blocks of C++ code.

```
int main() {
    try { throw 42; }
    catch(int &x) { return x; }
}
```

goto for C++ exception 2/2

Generate jumps inside blocks of C++ code.

```
int main(void) {  int __retres;
__tis_exc_stack_depth ++;
__tis_exc_stack[__tis_exc_stack_depth - 1].payload = tis_alloc((unsigned long)sizeof(int));
*((int *)__tis_exc_stack[__tis_exc_stack_depth - 1].payload) = 42;
__tis_exc_stack[__tis_exc_stack_depth - 1].typeinfo = & __tis_typeinfo_i;
__tis_exc_stack[__tis_exc_stack_depth - 1].inheritance = (struct __tis_inheritance const *)0U;
__tis_exc_stack[__tis_exc_stack_depth - 1].refcount = (long *)tis_alloc ((unsigned long)sizeof(int));
*(__tis_exc_stack[__tis_exc_stack_depth - 1].refcount) = 1L;
__tis_exc_stack[__tis_exc_stack_depth - 1].dtor = (void (*)(void *))0;
__tis_unwinding = 1;
goto __tis_unwinding_label;
if (0) {
  __tis_unwinding_label: __tis_unwinding = 0;
    if (__tis_exc_stack[__tis_exc_stack_depth - 1].typeinfo == & __tis_typeinfo_i) {
        int *x; int ____tis_exn_guard_CtorGuard; struct __tis_exn_guard __tis_exn_guard;
        x = (int *)__tis_exc_stack[__tis_exc_stack_depth - 1].payload;
        __tis_caught_stack_depth ++;
        *(__tis_exc_stack[__tis_exc_stack_depth - 1].refcount) += (long)1;
        __tis_caught_stack[__tis_caught_stack_depth - 1] = __tis_exc_stack[__tis_exc_stack_depth - 1];
        __tis_exc_stack_depth --;
        ____tis_exn_guard_CtorGuard = 1;
        __retres = *x;
        if (____tis_exn_guard_CtorGuard) __tis_exn_guard::Dtor(& __tis_exn_guard);
        goto return_label;
    } else { __tis_unwinding = 0; __tis_std_terminate(); } }
  return_label: return __retres;
}
```

Supporting dynamic calls/function pointers 1/2

```
struct Foo {
    //@ ensures \result == 12;
    virtual int f() { return 12; }
};

int main(void) {
    Foo foo;
    int r = foo.f();
    //@ assert virtual_call: r == 12;
    return r;
}
```

Supporting dynamic calls/function pointers 2/2

```
struct Foo {
   struct __tis_typeinfo const *__tis_typeinfo ;
   struct __tis_vmt_entry const *__tis_pvmt ;
};
/*@ requires \valid(this); ensures this->pvmt == __pvmt; */
void Foo::Ctor(struct Foo *this, struct __tis_vmt_entry const *__pvmt);
/*@ requires \valid(this); ensures \result ≡ 12; */
int Foo::f(struct Foo *this);
int main(void) {
    struct Foo foo; int r;
    Foo::Ctor(& foo,(struct __tis_vmt_entry const *)(& Foo::__tis_class_vmt));
    struct __tis_vmt_entry const *__virtual;
    __virtual = foo.__tis_pvmt + 1;
    r = (*((int (*)(struct Foo *))__virtual->method_ptr))
        ((struct Foo *)((char *)&foo + __virtual->shift_this));
    /*@ assert virtual_call: r ≡ 12; */ ;
    return r; }
struct __tis_vmt_entry const Foo::__tis_class_vmt[2U] =
  {{.method_ptr = (void (*)(void))(& Foo::__tis_class_inheritance),
    .shift_this = (long)0U,
    .shift_return = (long)0U},
   {.method_ptr = (void (*)(void))(& Foo::f),
    .shift_this = (long)0U,
    .shift_return = (long)0U}};
```

Supporting dynamic calls/function pointers 3/2

Function pointers in specifications:

```
struct Foo {
    //@ ensures \result == 12;
    virtual int f() { return 12; }
};
/*@ requires foo->f(void) == Foo::f(void); */
int h(Foo *foo) {
    int r = foo->f();
    //@ assert virtual_call: r == 12;
    return r;
}
```

Conclusion

We need readable counter-examples

- even with `goto` statements
- with relations between the output and the initial source code:
    - need meaningful names
    - even more when some interactive proof is required
- even with dynamic allocation: pervasive in C++ code using the STL
- need to detect unimplemented features: it is okay to refuse to prove something on a function if one can explain to the user why this code is out-of-scope.

`file.c: P is not proved (Timeout)` vs

```
file.c:42 cannot prove P in f because at line 44
          p is aliased to q
Counter-example: p == &V, q == &V
```