VERIFYING CONCURRENT C PROGRAMS WITH VCC, BOOGIE AND Z3

Michał Moskal (EMIC)
joint work with:
Ernie Cohen (Windows), Thomas Santen (EMIC),
Wolfram Schulte (RiSE), Stephan Tobies (EMIC),
Herman Venter (RiSE), and others
VCC

- VCC stands for Verifying C Compiler
- developed in cooperation between RiSE group at MSR Redmond and EMIC
- a sound C verifier supporting:
  - concurrency
  - ownership
  - typed memory model
- VCC translates annotated C code into BoogiePL
  - Boogie translates BoogiePL into verification conditions
  - Z3 (SMT solver) solves them or gives counterexamples
current main client:
- verification in cooperation between EMIC, MSR and the Saarland University
- kernel of Microsoft Hyper-V platform
- 60 000 lines of concurrent low-level C code (and 4 500 lines of assembly)
- own concurrency control primitives
- complex data structures
Annotate C code

Verify with **VCC**

Compile with regular C compiler

Verified

Error

Timeout

Executable

Inspect counterexample with **Model Viewer**

Inspect Z3 log with **Z3 Visualizer**

Fix code or specs with **VCC VS plugin**
Overview

- Naive modeling of flat C memory means annotation and prover overhead
  - Force a typed memory/object model
- Information hiding, layering, scalability
  - Spec#-style ownership
  - + Flexible invariants spanning ownership domains
- Modular reasoning about concurrency
  - Two-state invariants
When modeling memory as array of bytes, those functions wouldn’t verify.

```c
void bar(int *p, int *q) {
    *p = 12;
    *q = 42;
    assert(*p == 12);
}

void foo(int *p, short *q) {
    *p = 12;
    *q = 42;
    assert(*p == 12);
}
```
In VCC-1 you needed:

```c
void bar(int *p, int *q)
    requires(!overlaps(region(p, 4), region(q, 4)))
{
    *p = 12;
    *q = 42;
    assert(*p == 12);
}
```

- high annotation overhead, esp. in invariants
- high prover cost: disjointness proofs is something the prover does all the time
keep a set of **disjoint**, top-level, typed objects

- **check typedness at every access**

- pointers = pairs of memory address and type

- state = map from pointers to values

```c
struct A {
  int x;
  int y;
};
struct B {
  struct A a;
  int z;
};
```

```
⟨42, B⟩
⟨42, A⟩
⟨42, int⟩
⟨46, int⟩
⟨50, int⟩
```
REINTERPRETATION

- memory allocator and unions need to change type assignment
- allow explicit reinterpretation only on top-level objects
  - havoc new and old memory locations
  - possibly say how to compute new value from old (byte-blasting) [needed for memzero, memcpy]
- cost of byte-blasting only at reinterpretation
\[ \text{struct } \tau \{ ... \tau' f; ... \} \]
\[ \forall \sigma, r \cdot \text{typed}(\sigma, \langle r, \tau \rangle) \Rightarrow \]
\[ \text{dot}(\langle r, \tau \rangle, f) = \langle r + o, \tau' \rangle \land \]
\[ \text{typed}(\sigma, \text{dot}(\langle r, \tau \rangle, f)) \land \]
\[ \text{emb}(\sigma, \text{dot}(\langle r, \tau \rangle, f)) = p \land \]
\[ \text{path}(\sigma, \text{dot}(\langle r, \tau \rangle, f)) = f \]

- if you compute field address
- (within a typed object)
- the field is typed
- the field is embedded in the object (unique!)
- the only way to get to that location is through the field
```c
int *p, *q;
short *r;
struct A { int x, y; } *a;
struct B { int z; } *b;
```

```
path(...)  
a->x -----> a->y
emb(...)  
b->z
```

```
p !:= q
p
r
*q
```
Bitfields and flat unions

```c
struct X64VirtualAddress {
    i64 PageOffset:12; // <0:11>
    u64 PtOffset : 9; // <12:20>
    u64 PdOffset : 9; // <21:29>
    u64 PdptOffset: 9; // <30:38>
    u64 Pml4Offset: 9; // <39:47>
    u64 SignExtend:16; // <48:64>
};
union X64VirtualAddressU {
    X64VirtualAddress Address;
    u64 AsUINT64;
};
union Register {
    struct {
        u8 l;
        u8 h;
    } a;
    u16 ax;
    u32 eax;
};
```

- bitfields axiomatized on integers
- select-of-store like axioms
- limited interaction with arithmetic
Typed memory: summary

- forces an object model on top of C
- disjointness largely for free
  - for the annotator
  - for the prover
  - at the cost of explicit reinterpretation
- more efficient than the region-based model
VERIFICATION METHODOLOGY

- VCC-1 used **dynamic frames**
  - nice bare-bone C-like solution, but...
  - doesn’t scale (esp. when footprints depend on invariants)
  - no idea about concurrency
**Spec#-style Ownership**

- **Open object**: modification allowed
- **Closed object**: invariant holds

**System invariant:**

+ Hierarchical opening

- Invariants depend on ownership domain

**Owner link**
SEQUENTIAL OBJECT LIFE-CYCLE

- open
- mutable
- wrapped
- nested
- closed
- wrap
- unwrap
- wrap owner
- unwrap owner
- thread-owned

object can be modified
invariant holds

wrap/unwrap grand-owner
PROBLEMS

- for concurrency, we need to restrict changes to shared data
  - two-state invariants (preserved on closed objects across steps of the system)
  - updates on closed objects
  - but how to check invariants without the hierarchical opening?
- even in sequential case, invariants sometimes need to span natural ownership domains
  - for example...
Symbols of syntax tree nodes depend on the symbol table, but they cannot all own it!

```c
struct SYMBOL_TABLE {
    volatile char *names[MAX_SYM];
    invariant(forall(uint i; old(names[i]) != NULL ==> old(names[i]) == names[i]))
};
struct EXPR {
    uint id;
    SYMBOL_TABLE *s;
    invariant(s->names[id] != NULL)
};
```

But in reality they only depend on the symbol table growing, which is guaranteed by symbol table’s two-state invariant.
An invariant is **admissible** if updates of other objects (that maintain their invariants) cannot break it.

The idea:

- check that all invariants are admissible
  - in separation from verifying code
- when updating closed object, check only its invariant

By admissibility we know that all other invariants are also preserved
Two-state invariants are OK across system transitions:

\[ \forall \sigma_0, \sigma_1. \sigma_0 \triangleright \sigma_1 \Rightarrow \]
\[ \forall o. \sigma_0 (o, \text{closed}) \lor \sigma_1 (o, \text{closed}) \Rightarrow \]
\[ \text{inv}(\sigma_0, \sigma_1, o) \land \]
\[ \forall f. \neg \text{volatile}(f) \Rightarrow \sigma_0 (o, f) = \sigma_1 (o, f) \]

Things that you own are closed and have the owner set to you:

\[ \forall \sigma, o, c. \sigma (o, \text{closed}) \land c \in \sigma (o, \text{owns}) \Rightarrow \]
\[ \sigma (c, \text{closed}) \land \sigma (c, \text{owner}) = o \]
An invariant is **admissible** if updates of other objects (that maintain their invariants) cannot break it.

- non-volatile fields cannot change while the object is closed (implicitly in all invariants)
- if you are closed, objects that you own are closed (system invariant enforced with hierarchical opening)
- if everything is non-volatile, “changes” preserving its invariant are not possible and clearly cannot break your invariant
  - the Spec# case is covered
How can expression know the symbol table is closed?

- expression cannot own symbol table (which is the usual way)
- expression can own a handle (a ghost object)
  - handle to the symbol table has an invariant that the symbol table is closed
  - the symbol table maintains a set of outstanding handles and doesn’t open without emptying it first
    - which makes the invariant of handle admissible
struct Handle {
    obj_t obj;
    invariant(obj->handles[this] && closed(obj))
};

struct Data {
    bool handles[Handle*];
    invariant(forall(Handle *h; closed(h) ==> (handles[h] <=> h->obj == this)))
    invariant(old(closed(this)) && !closed(this) ==> !exists(Handle *h; handles[h]))
    invariant(is_thread(owner(this)) ||
       old(handles) == handles ||
       inv2(owner(this)))
};
Claims

- inline, built-in, generalized handle
- can claim (prevent from opening) zero or more objects
- can state additional property, much like an invariant
  - subject to standard admissibility check (with added assumption that claimed objects are closed)
  - checked initially when the claim is created
- allow for combining of invariants
- everything is an object! even formulas.
**Lock-free algorithms**

```c
struct LOCK {
    volatile int locked;
    spec( obj_t obj; )
    invariant( locked == 0 ==> obj->owner == this )
};

int TryAcquire(LOCK *l spec(claim_t c))
  requires(wrapped(c) && claims(c, closed(l)))
  ensures(result == 0 ==> wrapped(l->obj))
{
    int res, *ptr = &l->locked;
    atomic(l, c) {
        res = InterlockedCmpXchg(ptr, 0, 1);
        // inline: res = *ptr; if (res == 0) *ptr = 1;
        if (res) l->obj->owner = me;
    }
    return res;
}
```

- Havoc to simulate other threads; assume invariant of (closed!) lock
- Check two-state invariant of objects modified
- Pass claim to make sure the lock stays closed (valid)

Verified locks, rundowns, concurrent stacks, sequential lists...
Heap partitioning:

- Ownership domains of threads
- Shared state

Threads are also considered objects.

“Owns” is inverse of the owner link and can be marked “volatile.”

Volatile vs. non-volatile
Concurrent meets sequential

- operations on thread-local state only performed by and visible to that thread
- operations on shared state only in `atomic(...) {...} blocks`
- effects of other threads simulated only at the beginning of such block
  - their actions can be squeezed there because they cannot see our thread-local state and vice versa
- otherwise, Spec#-style sequential reasoning
SEQUENTIAL FRAMING

also for claims!

explicitly in domain

writes

possibly modified

havoc
**WHAT’S LEFT TO DO?**

- **superposition** – injecting ghost code around an atomic operation performed by a function that you call

- we only went that **low**
  - address manager/hardware \(\leftrightarrow\) flat memory
  - thread schedules \(\leftrightarrow\) logical VCC threads

- **annotation** overhead

- **performance!**
  - VC splitting, distribution
  - axiomatization fine tuning, maybe decision procedures
THE END

• questions?