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 couterexamples

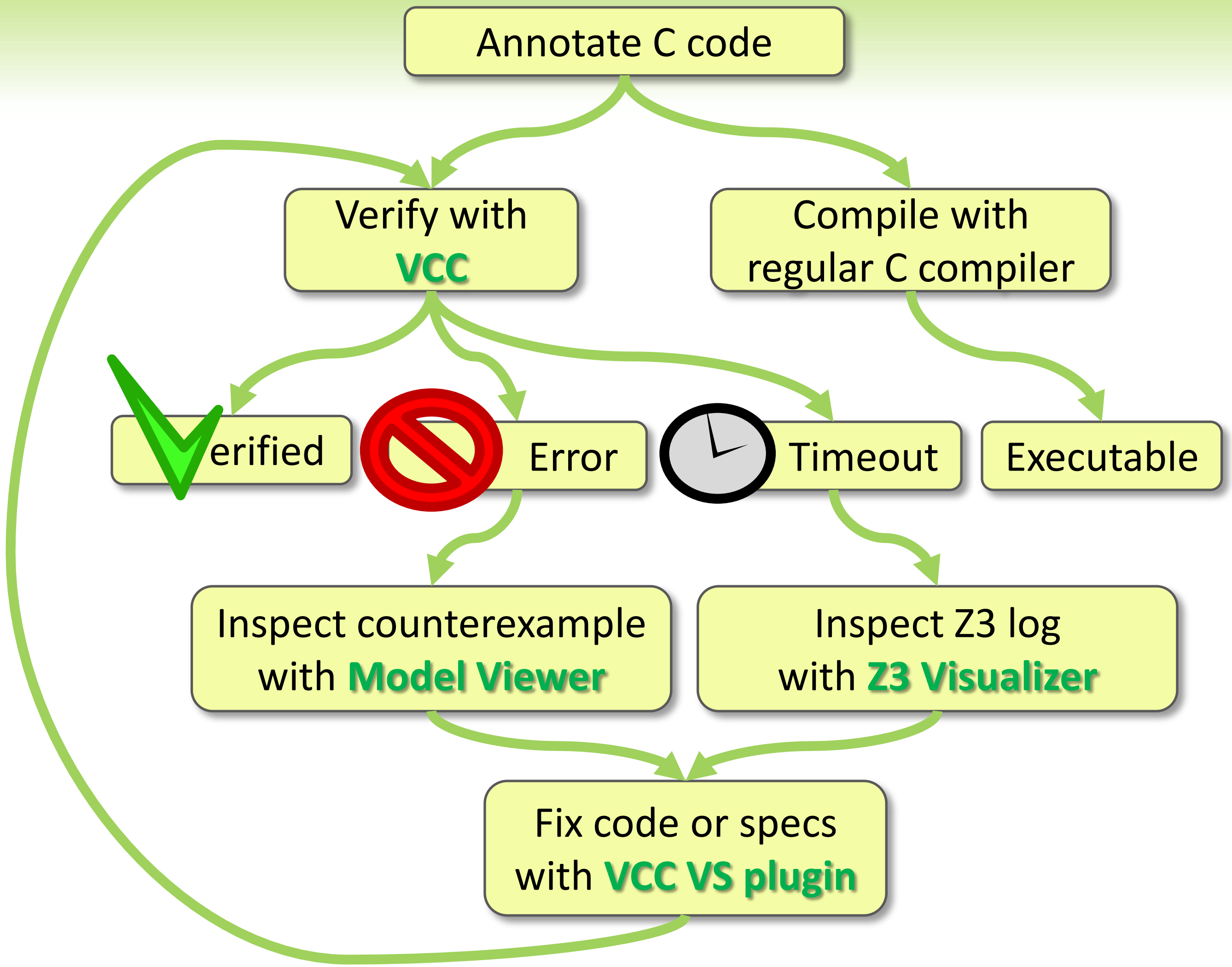
Research in Software Engineering

European Microsoft Innovation Center, Aachen

Hypervisor

- 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

VCC WORKFLOW



Overview

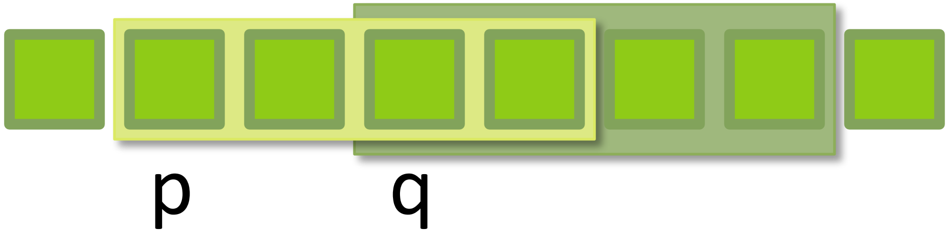
- naive modeling of flat C memory means annotation and prover overhead
  - force a typed memory/object model
- information hiding, layering, scalability
  - Spec#-style ownership
  - + flexible invariants spanning ownership domains
- modular reasoning about concurrency
  - two-state invariants

Partial overlap

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

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

When modeling memory as array of bytes, those functions wouldn't verify.



VCC-1: REGIONS

In VCC-1 you needed:

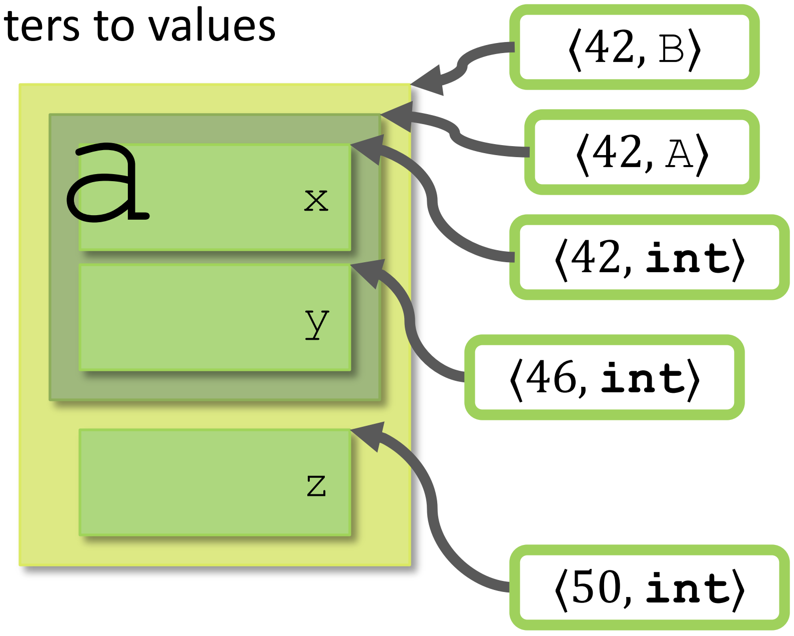
```
void bar(int *p, int *q)
     requires(!overlaps(region(p, 4), region(q, 4)))
{
     *p = 12;
     *q = 42;
     assert(*p == 12);
}
```

- high annotation overhead, esp. in invariants
- high prover cost: disjointness proofs is something the prover does all the time

Typed memory

- 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

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



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

Disjointness with embedding and path

$$\textbf{struct}\ \tau\ \{\ \dots\ \tau'\ f;\ \dots\ \}$$

$$\forall \sigma, r.\ typed(\sigma, \langle r, \tau\rangle) \Rightarrow$$

$$dot(\langle r, \tau\rangle, f) = \langle r + o, \tau'\rangle \wedge$$

$$typed\big(\sigma, dot(\langle r, \tau\rangle, f)\big) \wedge$$

$$emb\big(\sigma, dot(\langle r, \tau\rangle, f)\big) = p \wedge$$

$$path\big(\sigma, dot(\langle r, \tau\rangle, f)\big) = f$$

if you compute **field adress**

(within a **typed** object)

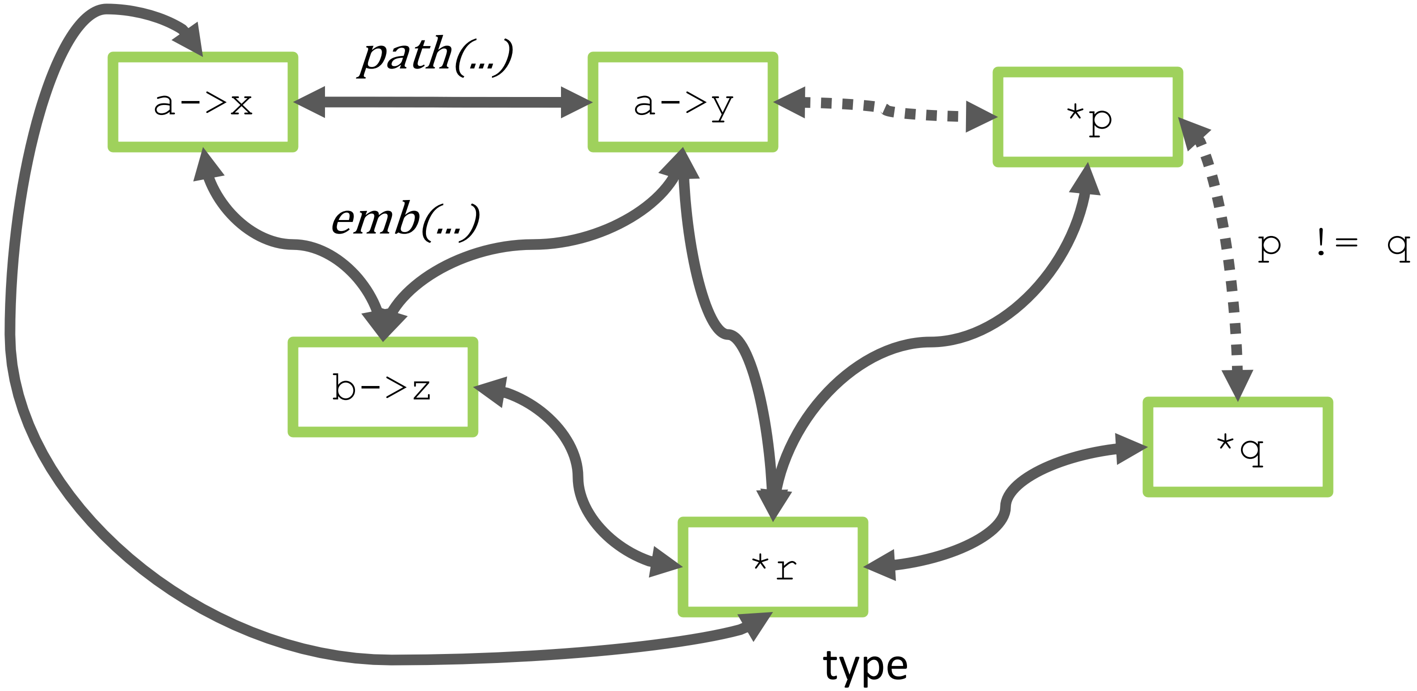
the field is **typed**

the field is **embedded** in the object (unique!)

the **only** way to get to that location is through the field

Writes commute by ...

```
int *p, *q;
short *r;
struct A { int x, y; } *a;
struct B { int z; } *b;
```



Bitfields and flat unions

```
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

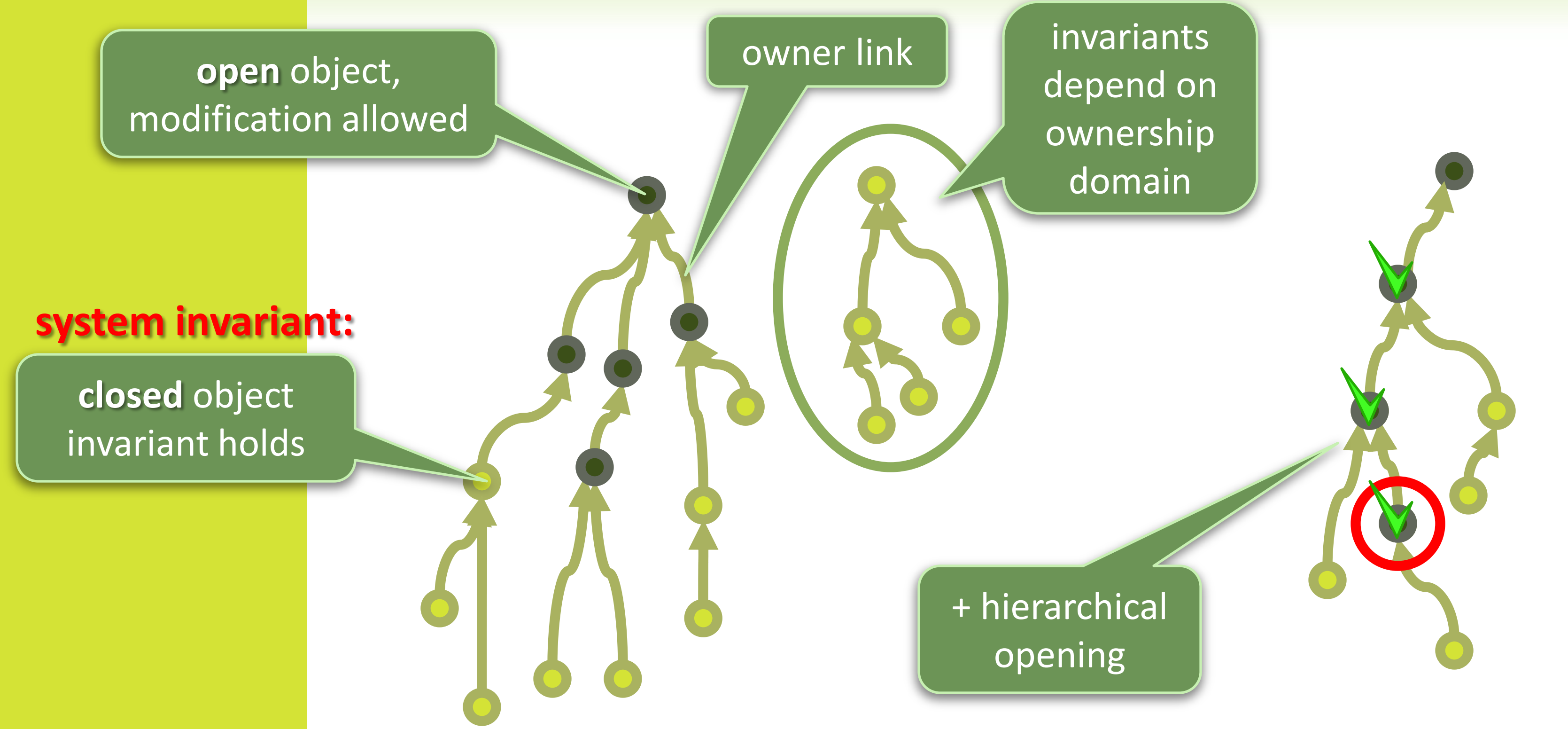
Typed memory: summary

- 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

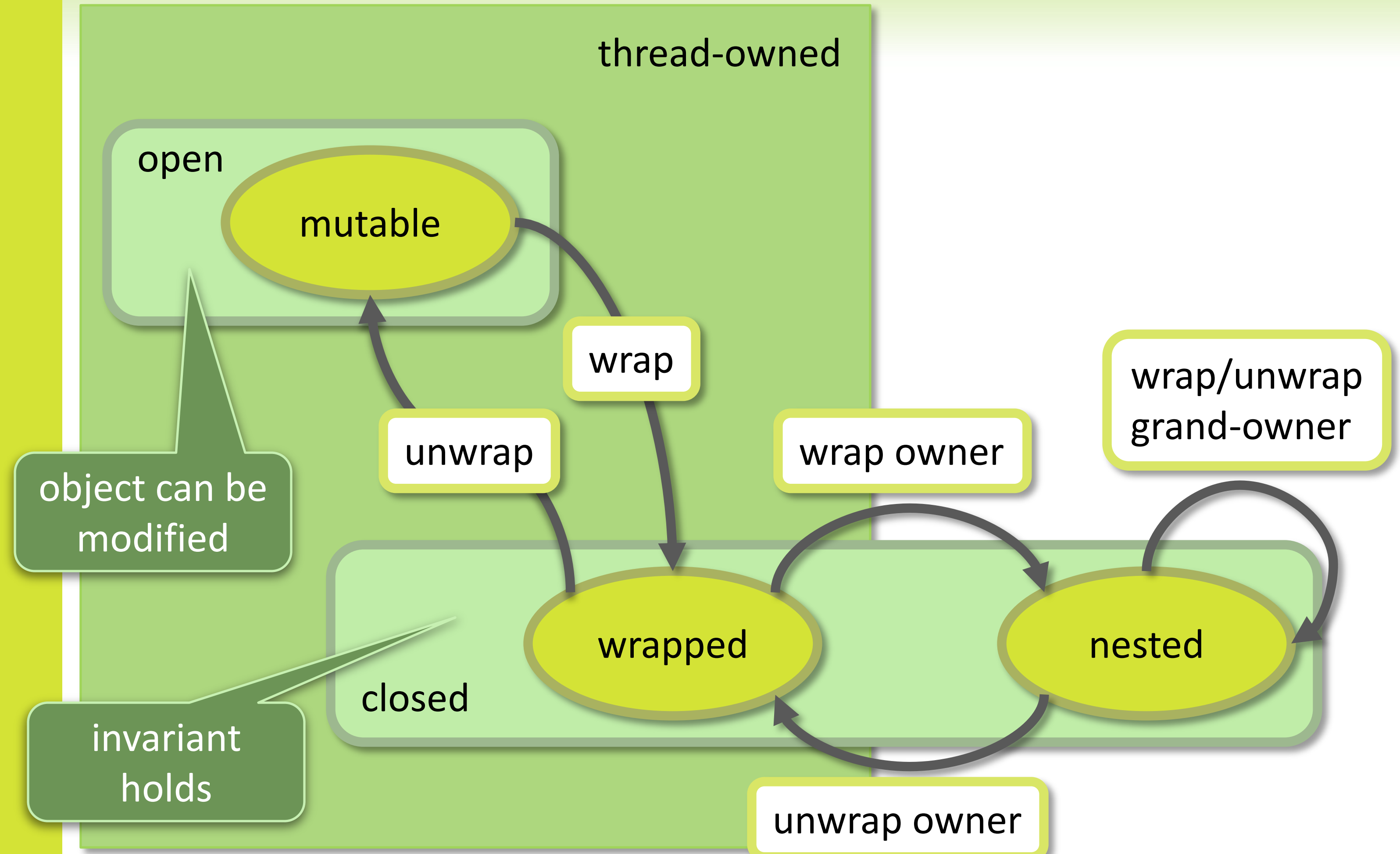
Verification methodology

- 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



Sequential object life-cycle



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...

Symbol table example

Invariants of syntax tree nodes depend on the symbol table, but they cannot **all** own it!

```
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)
};
```

typical for concurrent objects

But in reality they only depend on the symbol table **growing**, which is guaranteed by symbol table's **two-state invariant**.

Admissibility

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

generate proof obligation

By admissibility we know that all other invariants are also preserved

System invariants

Two-state invariants are OK across system transitions:

$$\forall \sigma_0, \sigma_1 . \sigma_0 \rhd \sigma_1 \Rightarrow$$
$$\forall o . \sigma_0(o, \text{closed}) \vee \sigma_1(o, \text{closed}) \Rightarrow$$
$$inv(\sigma_0, \sigma_1, o) \wedge$$
$$\forall f . \neg volatile(f) \Rightarrow \sigma_0(o, f) = \sigma_1(o, f)$$

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

$$\forall \sigma, o, c . \sigma(o, \text{closed}) \wedge c \in \sigma(o, \text{owns}) \Rightarrow$$
$$\sigma(c, \text{closed}) \wedge \sigma(c, \text{owner}) = o$$

Sequential admissibility

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**

Handles

```
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

Verified locks,
rundowns,
concurrent stacks,
sequential lists...

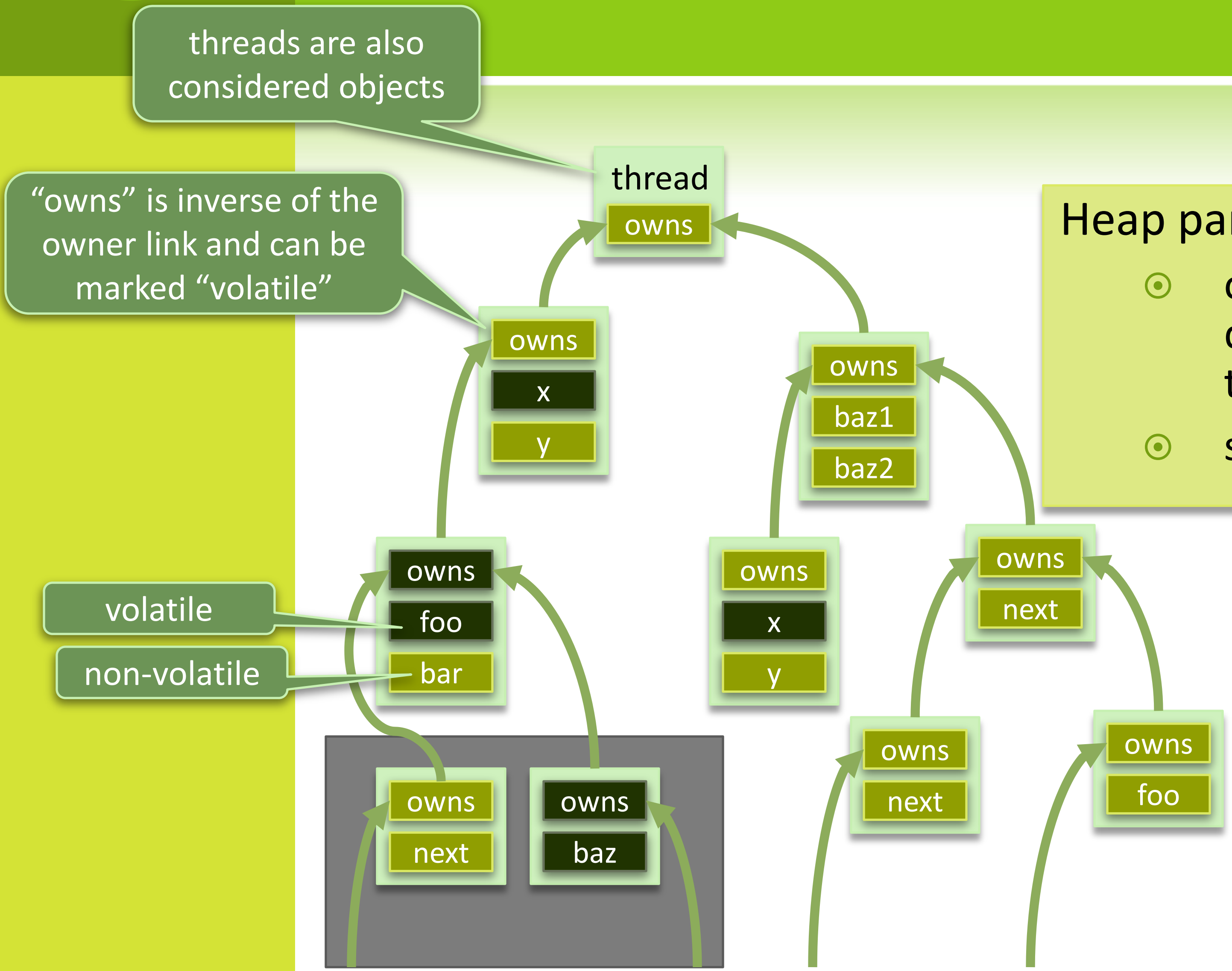
```
struct LOCK {
  volatile int locked;
  spec( obj_t obj; )
  invariant( locked == 0 ==> obj->owner == this )
};

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)

Heap partitioning



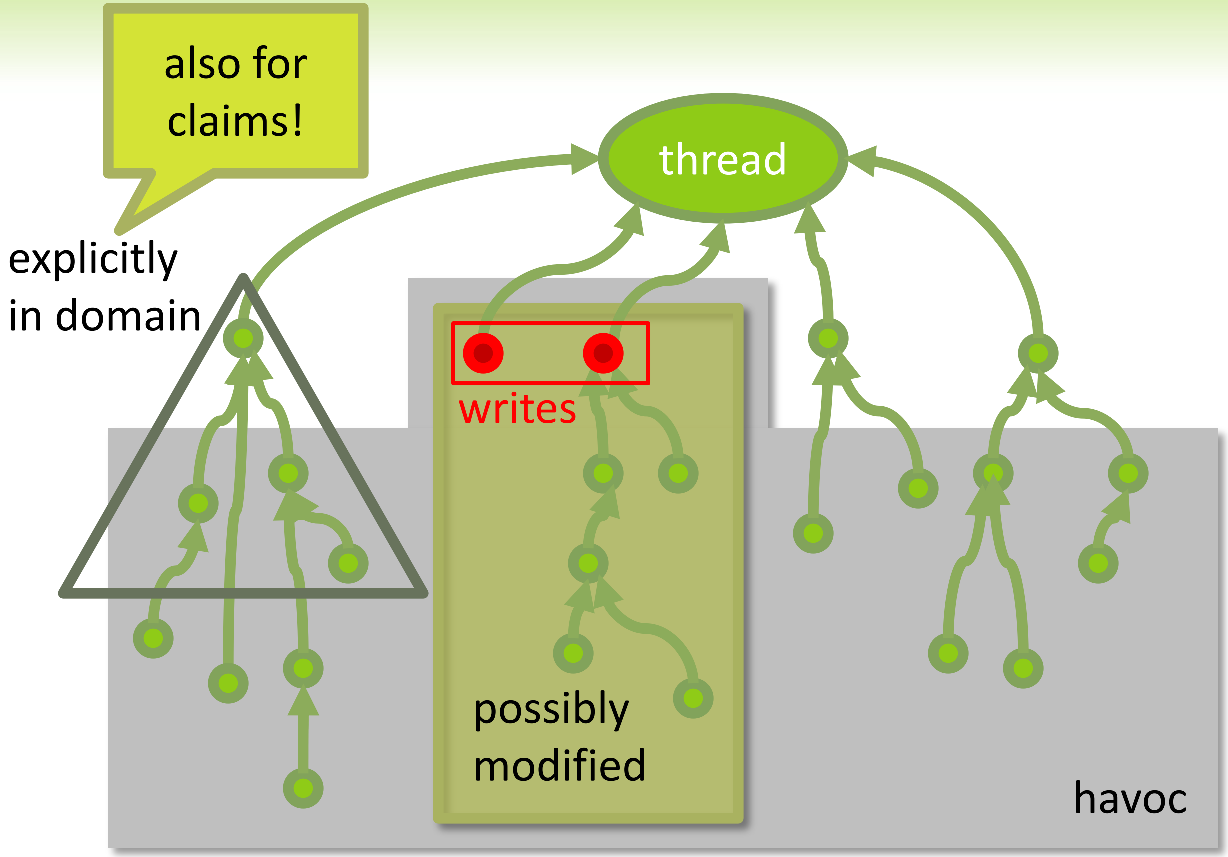
Heap partitioned into:

- ownership domains of threads
- shared state

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



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 <=> flat memory
  - thread schedules <=> logical VCC threads
- **annotation** overhead
- **performance!**
  - VC splitting, distribution
  - axiomatization fine tuning, maybe decision procedures

The end

- questions?