# Verifying Real-World C Programs with VCC


Ernie Cohen, Markus Dahlweid,
Michał Moskal, Wolfram Schulte,
Thomas Santen, Stephan Tobies

Microsoft Research

Microsoft Innovation Center Europe

Verisoft XT

# MOTIVATION

# WHY BOTHER VERIFYING C? (1996)

**From:** owner-softverf@leopard.cs.byu.edu
**Date:** Mon, 11 Mar 1996 07:05:31 -0500 (EST)
**Subject:** Why bother verifying C? and other such questions

…The reason ***to verify C is because it is the most common language used...*** To say that a little better you will be providing more overall verification to the universe of current software by verifying C than by doing so for any other language.

That's the good part. Here's the bad part. ***By trying to verify C, you are starting something that you will likely never finish. You will almost certainly experience a sense of utter dismay*** at the number of flaws you find,...

The third thing to realize is that **you will probably learn absolutely nothing about the underlying issues of proving properties of programs** through your effort - unless of course you are not already an expert. So, let's see...

Microsoft Research

Microsoft Innovation Center Europe

Verisoft XT

# Why Bother Verifying C Today?

Modern, type-safe PLs preclude many defects that verification can catch, but:

- most system-level code is still written in C:
  - operating systems, device drivers
- operating systems stopped being single-threaded 20 years ago
- testing concurrent system-level software is very challenging
- a failure of system-level functionality can have a high impact
  - on many applications
  - on many systems

# Real-World Code to Verify: Windows Hypervisor

## Hyper-V

- virtualization platform for x64 architecture
- scalable, reliable, highly available

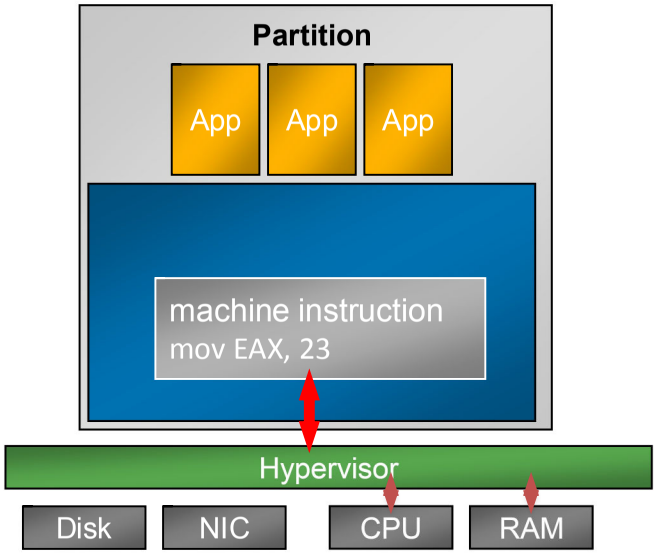
## Windows Hypervisor

- core component of Hyper-V
- thin layer of software between hardware and OS
- allows multiple operating systems to run, unmodified, on a host computer at the same time
- simple partitioning functionality
- maintains strong isolation between partitions

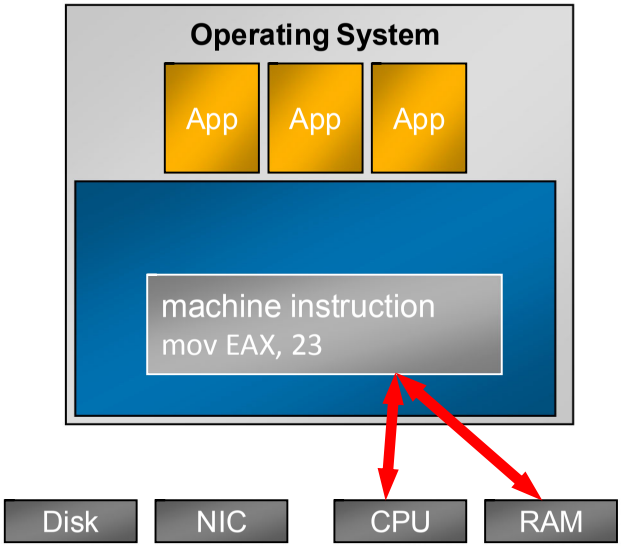
# HV Correctness: Simulation

A partition cannot distinguish (with some exceptions) whether a machine instruction is executed

a) through the HV OR b) directly on a processor




≈




Microsoft Research

Microsoft Innovation Center Europe

Verisoft XT

# Hypervisor Implementation

- ca. 100.000 lines of C, 5000 lines of assembler
- Concurrency
  - spin locks, r/w locks, rundowns, turnstiles
  - lock-free accesses to volatile data and hardware
  - access “protocol” involves complex interactions between data structures
- Access to hardware registers (memory management, virtualization support)

# Challenges for Verification of Concurrent C

1. **Memory model** that is adequate and efficient to reason about
2. **Modular reasoning** about concurrent code
3. **Invariants** for (large and complex) C data structures
4. Huge verification conditions to be proven **automatically**
5. "Live" specifications that **evolve with the code**

Microsoft Research

Microsoft Innovation Center Europe

Verisoft XT

# Hypervisor Verification (2007 – 2010)

Partners:

- European Microsoft Innovation Center
- Microsoft Research
- Microsoft Corporate
- Universität des Saarlandes


co-funded by the German Ministry of Education and Research

http://www.verisoftxt.de

Microsoft Research

Microsoft Innovation Center Europe

Verisoft XT

# GENERAL APPROACH

Microsoft Research

Microsoft Innovation Center Europe

Verisoft XT

# The Microsoft Verifying C Compiler (VCC)

- Source Language
  - ANSI C +
  - Design-by-Contract Annotations +
  - Ghost state +
  - Theories +
  - Metadata Annotations
- Program Logic
  - Dijkstra's weakest preconditions
- Automatic Verification
  - verification condition generation (VCG)
  - automatic theorem proving (SMT)


Microsoft Research

Microsoft Innovation Center Europe

Verisoft XT

# Contracts / Modular Verification

```
int foo(int x)
  requires(x > 5)        // precond
  ensures(result > x)    // postcond
{
…
}
```

```
void bar(int y; int *z)
  writes(z)              // framing
  requires(y > 7)
  maintains(*z > 7)      // invariant
{
   *z = foo(y);
   assert(*z > 7);
}
```

- function contracts: pre-/postconditions, framing
- modularity: **`bar`** only knows contract (but not code) of **`foo`**

advantages:

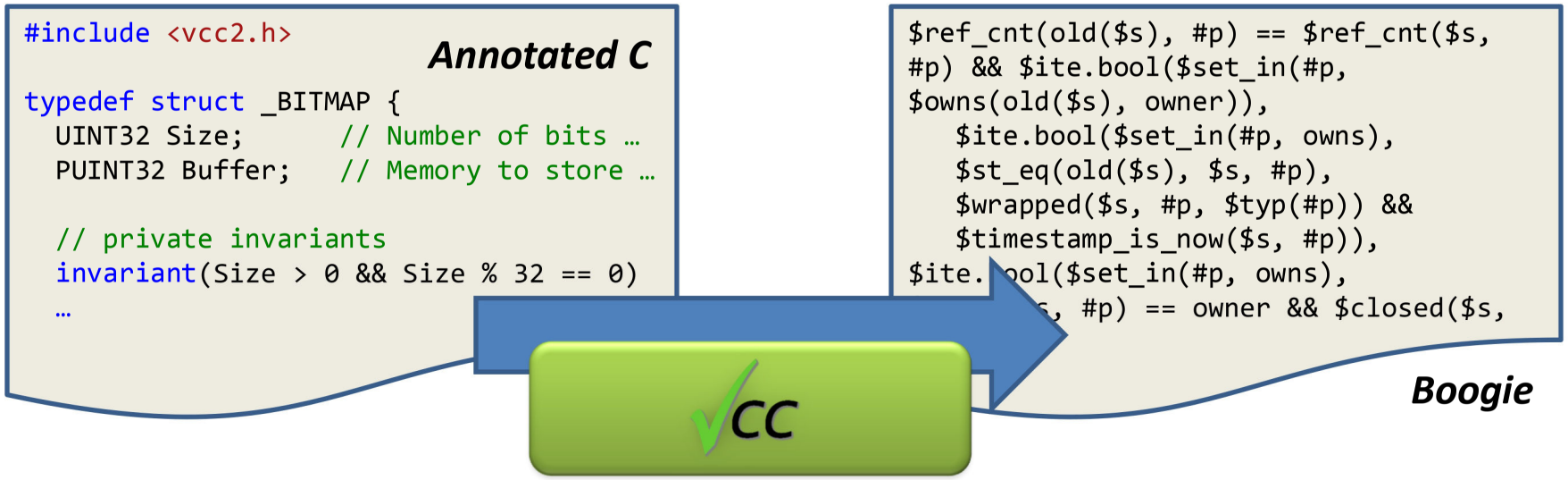
- modular verification: one function at a time
- no unfolding of code: scales to large applications

Microsoft Research

Microsoft Innovation Center Europe

Verisoft XT

# Tool Chain



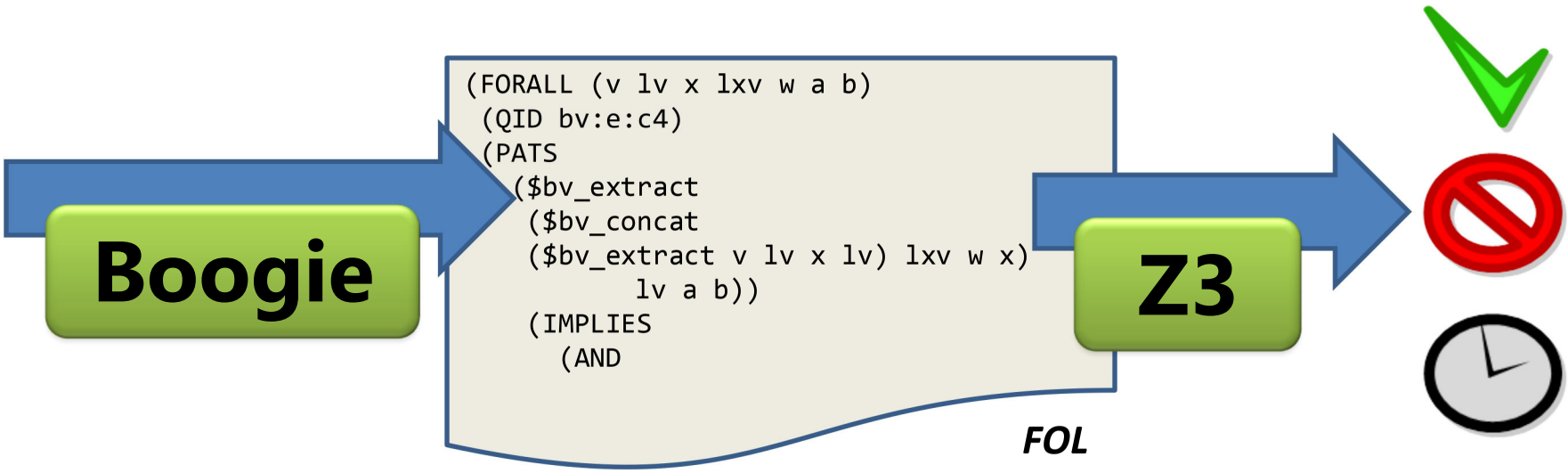

- Boogie
  - Verification Condition Generator
  - Minimal imperative control flow and types on top of first order logic
  - Microsoft Research (Rustan Leino)

Microsoft Research

Microsoft Innovation Center Europe

Verisoft XT

# Tool Chain




- Z3
  - State-of-the-art automatic SMT Solver
  - SMT (Satisfiability Modulo Theories):
    - Integer and fixed-length bit-vector arithmetic
    - Arrays, algebraic data types
  - Microsoft Research (L. de Moura, N. Bjørner)

Microsoft Research

Microsoft Innovation Center Europe

Verisoft XT

# MODELING C MEMORY

Microsoft Research

Microsoft Innovation Center Europe

Verisoft XT

# C Memory Model

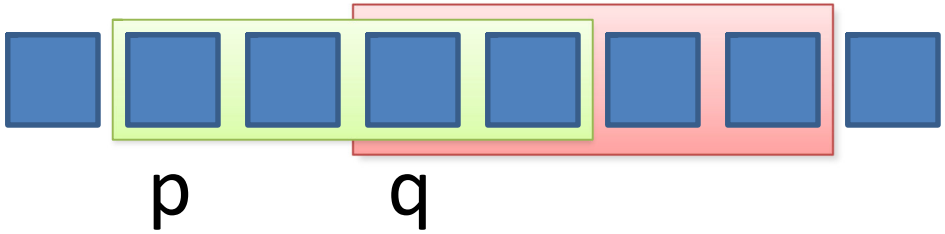
- Stack and ‘heap’ memory is organized into disjoint, fixed-size allocation regions
  - Heap is really only chunks of memory obtained via a system call
- Types ‘suggest’ how to interpret regions
- Allocation status is tracked per region
- Type status is not tracked at all

# ALIASING AND PARTIAL OVERLAP

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

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

When modeling memory as array of bytes, neither function would verify.



# VCC-1 : Disjoint Regions

In VCC-1 this was solved by:

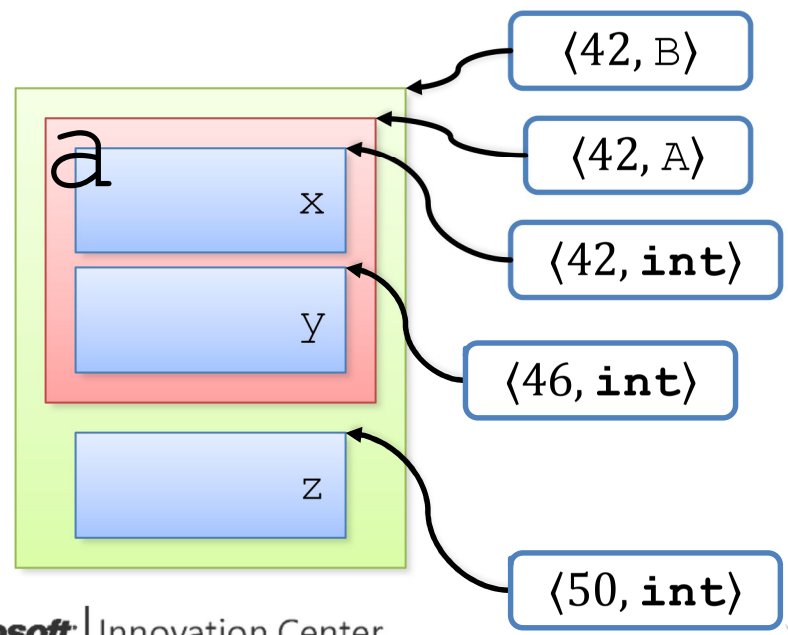
```
void bar(int *p, int *q)
  requires(!overlaps(region(p, 4), region(q, 4)))
{
  *p = 12;
  *q = 42;
  assert(*p == 12);
}
```

- high **annotation overhead**, esp. in invariants
- high prover cost: disjointness proofs is something the prover does **all the time** ( and then times out)

# VCC-2 : Typed objects

- 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;
};
```



# Object Invariants

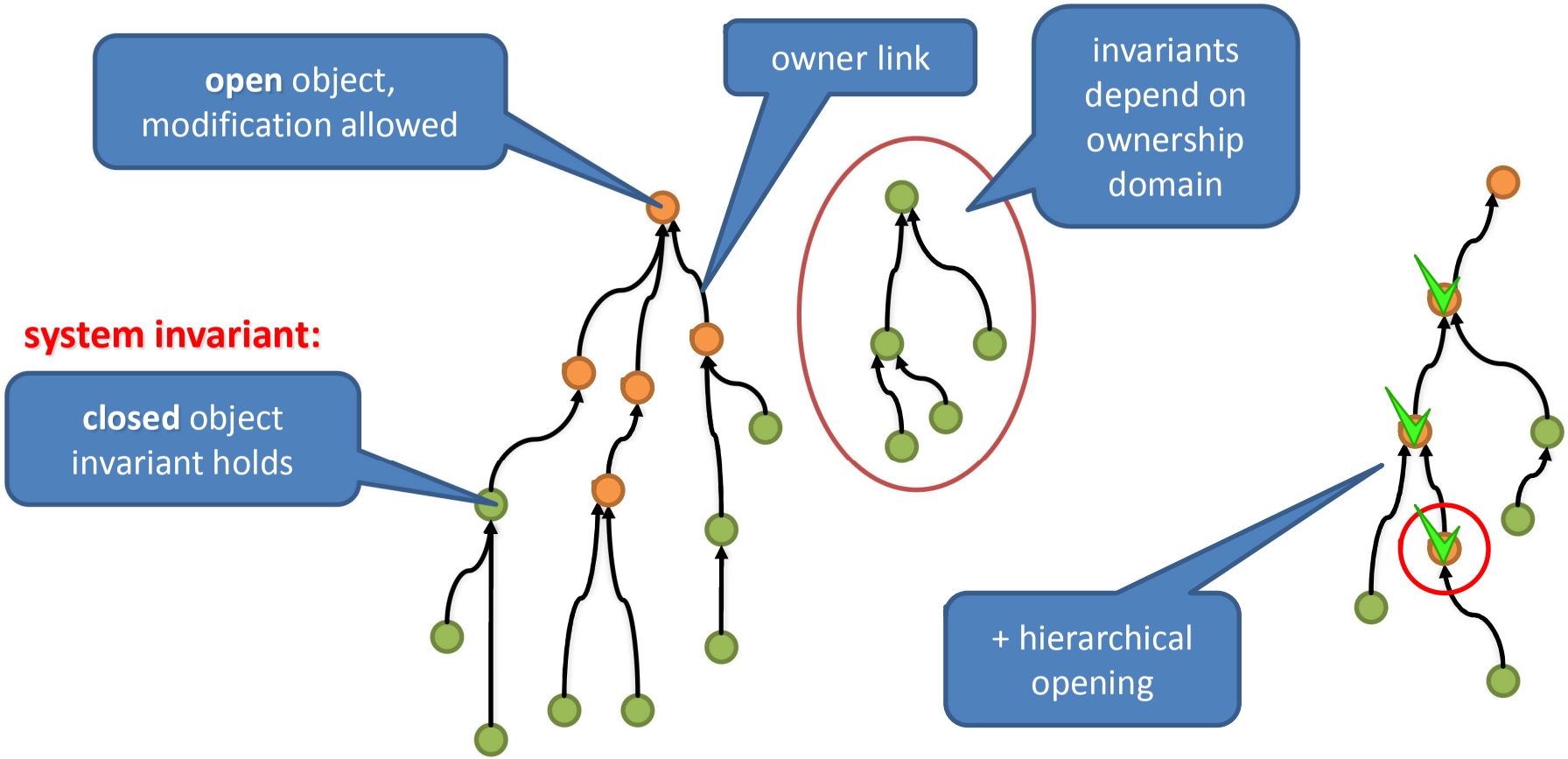
- associated with struct or union types
- hold for "closed" objects (cf. status)
- simplify contracts

  no need to mention invariants in pre/postcondition

```
struct S
{
  int a;
  int b;
  invariant(a>0 && b>a)
}
```

critical for major hypervisor data structures

# SEQUENTIAL ACCESS: OWNERSHIP

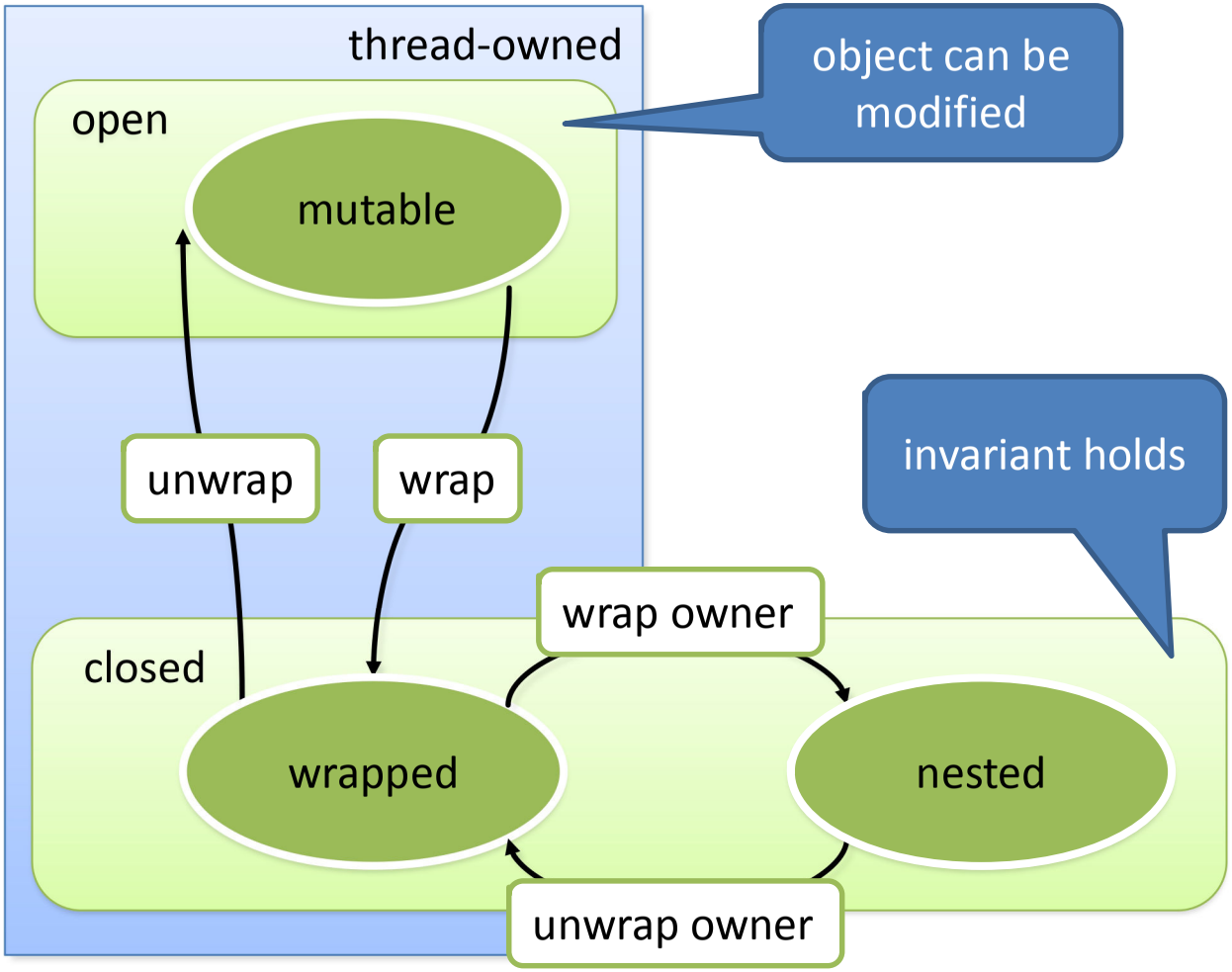



Microsoft Research

Microsoft Innovation Center Europe

Verisoft XT

# Sequential Object Life-Cycle




Microsoft Research

Microsoft Innovation Center Europe

Verisoft XT

# DEMO

Microsoft Research

Microsoft Innovation Center Europe

Verisoft XT

# Demo Screenshots

VccDemo - Microsoft Visual Studio (Administrator)

File Edit View Project Build Debug Tools Test Analyze Window Verify Help

Debug | Win32 | Map.mem

BitMap.h*

_BITMAP

```
#pragma once

#include <vcc2.h>
#include "spec.h"

// |-------------------------------------------------------|
// | 010...                                                |
// |-------------------------------------------------------|
//   <--                    Size                       -->

typedef struct _BITMAP {
  unsigned int Size;      // Number of bits in bit map
  unsigned int *Buffer;   // Memory to store the bit map

  #pragma region contract

  // private invariants
  invariant(Size > 0 && Size % 32 == 0)
  invariant(keeps(as_array(Buffer, Size / 32)))

  // public abstraction
  spec(bool BM[unsigned int];) // unsigned int --> {true, false}
  invariant(forall(unsigned int i; i < Size ==> BM[i] == ToBm32(Buffer[i/32])[i%32]))
  // ToBm32: unsigned int -> ( {0..31} -> {true, false} )

  #pragma endregion
} BITMAP;
```

- Check admissibility of '_BITMAP'
- Verify file 'BitMap.h'
- Custom verify file 'BitMap.h'
- More VCC Commands
- Call Browser
- Go To Definition
- Go To Declaration
- Find All References
- Breakpoint
- Run To Cursor
- Cut
- Copy
- Paste
- Outlining

Ready | Ln 14 | Col 57 | Ch 57 | INS

Microsoft Research

Microsoft Innovation Center Europe

Verisoft XT

# Demo Screenshots

VccDemo - Microsoft Visual Studio (Administrator)

File Edit View Project Build Debug Tools Test Analyze Window Verify Help

Debug | Win32 | Map.mem

BitMap.h*

(Global Scope)

```
void InitializeBitMap (BITMAP *BitMap, unsigned int * BitMapBuffer, unsigned int Size)
  #pragma region contract
  writes(extent(BitMap), as_array(BitMapBuffer, Size / 32))
  requires(wrapped(as_array(BitMapBuffer, Size / 32)))
  requires(Size > 0 && Size % 32 == 0)
  ensures(wrapped(BitMap))
  ensures(BitMap->Size == Size)
  ensures(forall(unsigned int i; i < Size ==> BitMap->BM[i] == false))
  #pragma endregion
  ;

void SetBit (BITMAP *BitMap, unsigned int BitNumber)
  #pragma region contract
  writes(BitMap)
  maintains(wrapped(BitMap))
  requires(BitNumber < BitMap->Size)
  ensures(forall(unsigned int i; BitMap->BM[i] == (i == BitNumber ? true : old(BitMap->BM[i]))))
  ensures(unchanged(BitMap->Size))
  #pragma endregion
  ;

ispure unsigned __int8 TestBit (BITMAP *BitMap, unsigned int BitNumber)
  #pragma region contract
  reads(&BitMap->BM)
  requires(wrapped(BitMap))
  requires(0 < BitNumber && BitNumber < BitMap->Size)
  ensures(result == BitMap->BM[BitNumber])
  #pragma endregion
  ;
```

Ready | Ln 53 | Col 1 | Ch 1 | INS

Microsoft Research

Microsoft Innovation Center Europe

Verisoft XT

# Demo Screens

VccDemo - Microsoft Visual Studio (Administrator)

File Edit View Project Build Debug Tools Test Analyze Window Verify Help

Debug | Win32 | Map.mem

BitMap.c | BitMap.h

(Global Scope) | InitializeBitMap(BITMAP * BitMap, unsigned int * BitMapBuffer, unsigned int Size)

```
void InitializeBitMap (BITMAP *BitMap, unsigned int *BitMapBuffer, unsigned int Size)
{
  BitMap->Size = Size;
  BitMap->Buffer = BitMapBuffer;
  memzero(BitMapBuffer, Size/32);

  speconly(BitMap->BM = lambda(unsigned int i; i < Size; false);)
  wrap(BitMap);
}

void SetBit (BITMAP *BitMap, unsigned int BitNumber)
{
  expose(BitMap) {
  expose(as_array(BitMap->Buffer, BitMap->Size /32)) {
    BitMap->Buffer[BitNumber/32] |= 1 << BitNumber % 32;
    speconly(BitMap->BM = lambda(unsigned int i; true; i == BitNumber ? true : BitMap->BM[i]);)
  }}
```

Output

Show output from: Verification

```
set PATH=C:\Program Files (x86)\Microsoft Visual Studio 9.0\Common7\IDE;C:\Program Files (x86)\Microsoft Visual S
executing vcc in directory: c:\Users\stobies\verisoft\FELT\Vcc2\Vcc2Tests\Demo
C:\Program Files (x86)\Microsoft Research\VerifiedC\binaries\vcc2.exe /p:/D_UNICODE;/DUNICODE;/I"C:\Program Files
vcc : Verification of _BITMAP#adm succeeded. [0.89]
vcc : Verification of InitializeBitMap succeeded. [0.05]
vcc : Verification of SetBit succeeded. [0.36]
vcc : Verification of TestBit#reads succeeded. [0.02]
Time: 2.04s total, 0.73s compiler, 0.00s Boogie, 1.31s method verification.
=============== VCC ends ================
```

Output | Error List | Pending Changes

Ready | Ln 9 | Col 1 | Ch 1 | INS

Microsoft Research

Microsoft Innovation Center Europe

Verisoft XT

# CONCURRENCY

Microsoft Research

Microsoft Innovation Center Europe

Verisoft XT

# CONCURRENT ACCESS: *VOLATILE*

- sequential approach not suitable for **lock-free** concurrency (locks, rundowns,…)
- atomic operation on *volatile* data subject to **two-state invariants**
- requires objects to be closed
- once concurrent data structures are verified, they guard sequential access to data – **no built-in concurrency primitives** needed

# DEMO

Microsoft Research

Microsoft Innovation Center Europe

Verisoft XT

# Demo Screenshots

VccDemo - Microsoft Visual Studio (Administrator)

File Edit View Project Build Debug Tools Test Analyze Window Verify Help

Debug | Win32 | Map.mem

SpinLock.h* | BitMap.c | BitMap.h

(Global Scope)

```
#include <vcc2.h>
#include <stdlib.h>

#define SIMPLE_SPIN_LOCKS

typedef struct vcc(claimable) vcc(volatile_owns) _SPIN_LOCK
{
  volatile int Lock;

  spec(obj_t protected_obj;)
  invariant(!Lock ==> set_in(protected_obj, owns(this)))
} SPIN_LOCK;


void InitializeSpinLock(SPIN_LOCK * SpinLock spec(obj_t obj))
  writes(span(SpinLock), obj)
  requires(wrapped(obj))
  ensures(wrapped(SpinLock))
  ensures(SpinLock->protected_obj == obj)
  ;
```

Ready | Ln 25 | Col 1 | Ch 1 | INS

Microsoft Research

Microsoft Innovation Center Europe

Verisoft XT

# Demo Screenshots

VccDemo - Microsoft Visual Studio (Administrator)

File Edit View Project Build Debug Tools Test Analyze Window Verify Help

Debug | Win32 | Map.mem

SpinLock.c | SpinLock.h* | BitMap.c | BitMap.h

(Global Scope) | Acquire(SPIN_LOCK * SpinLock)

```
void InitializeSpinLock(SPIN_LOCK *SpinLock spec(obj_t obj))
{
  SpinLock->Lock = 0;
  speconly(SpinLock->protected_obj = obj;)
  set_owns(SpinLock,SET(obj));
  wrap(SpinLock);
}

#ifdef SIMPLE_SPIN_LOCKS

void Acquire(SPIN_LOCK *SpinLock)
{
  int stop;
  do  {
    atomic(SpinLock) {
      stop = (__interlockedcompareexchange(&SpinLock->Lock, 1, 0) == 0);
      speconly(if (stop) giveup_closed_owner(SpinLock->protected_obj, SpinLock);)
    }
  } while (!stop);
}

void Release(SPIN_LOCK *SpinLock)
{
  atomic(SpinLock) {
    SpinLock->Lock = 0;
    set_closed_owner(SpinLock->protected_obj, SpinLock);
  }
}
```

Ready | Ln 16 | Col 2 | Ch 2 | INS

Microsoft Research

Microsoft Innovation Center Europe

Verisoft XT

# Keeping Locks Closed

1. Handle
   - references lock
   - can be **owned by arbitrary** threads or objects
   - (that do not own the lock, thus cannot open it)
2. Invariant on lock:
   - lock stays closed while closed handles exist
3. Owner of a handle can use the lock

# Concurrent access: Claims

- First-class objects, reference-counted handles to other objects
- Can claim (prevent from opening) zero or more objects
- Can state additional property, much like an invariant
- Allow for combining of invariants
- Everything is an object, even formulas.

# DEMO

Microsoft Research

Microsoft Innovation Center Europe

Verisoft XT

# Demo Screenshots

VccDemo - Microsoft Visual Studio (Administrator)

File Edit View Project Build Debug Tools Test Analyze Window Verify Help

Debug | Win32 | Map.mem

SpinLock.c | SpinLock.h* | BitMap.c | BitMap.h

(Global Scope)

```
#else

#define _claimable_ vcc(claimable)

void Acquire(SPIN_LOCK *SpinLock claimp(access_claim))
  always(access_claim, closed(SpinLock))
  ensures(wrapped(SpinLock->protected_obj))
  ensures(is_fresh(SpinLock->protected_obj))
  ;

void Release(SPIN_LOCK *SpinLock claimp(access_claim))
  writes(SpinLock->protected_obj)
  always(access_claim, closed(SpinLock))
  requires(access_claim != SpinLock->protected_obj)
  requires(wrapped(SpinLock->protected_obj))
  ;

#endif
```

Ready | Ln 70 | Col 7 | Ch 7 | INS

Microsoft Research

Microsoft Innovation Center Europe

Verisoft XT

# Demo Screenshots

VccDemo - Microsoft Visual Studio (Administrator)

File Edit View Project Build Debug Tools Test Analyze Window Verify Help

Debug | Win32 | Map.mem

SpinLock.c | SpinLock.h* | BitMap.c | BitMap.h

(Global Scope) | Acquire(SPIN_LOCK * SpinLock)

```
#else

void Acquire(SPIN_LOCK *SpinLock claimp(access_claim))
{
  int stop;
  do {
    atomic(access_claim, SpinLock) {
      stop = (__interlockedcompareexchange(&SpinLock->Lock, 1, 0) == 0);
      speconly(if (stop) giveup_closed_owner(SpinLock->protected_obj, SpinLock);)
    }
  } while (!stop);
}

void Release(SPIN_LOCK *SpinLock claimp(access_claim))
{
  atomic(access_claim, SpinLock) {
    SpinLock->Lock = 0;
    set_closed_owner(SpinLock->protected_obj, SpinLock);
  }
}

#endif
```

Ready | Ln 16 | Col 2 | Ch 2 | INS

Microsoft Research

Microsoft Innovation Center Europe

Verisoft XT

# VERIFICATION ENGINEERING

Microsoft Research

Microsoft Innovation Center Europe

Verisoft XT

# IMPLICIT VS. EXPLICIT KNOWLEDGE

- For example Hypervisor struct for partitions:
  - 70 fields (mostly of structured type)
  - 7 locks, 4 rundowns
  - Access protocols are entirely implicit or in comments:

```
//
// This lock protects the Vp table and should be
// acquired for read to lookup entries and
// for write to add/remove entries
//
```

- Explicit knowledge required for verification
- Additional structure in the form of groups

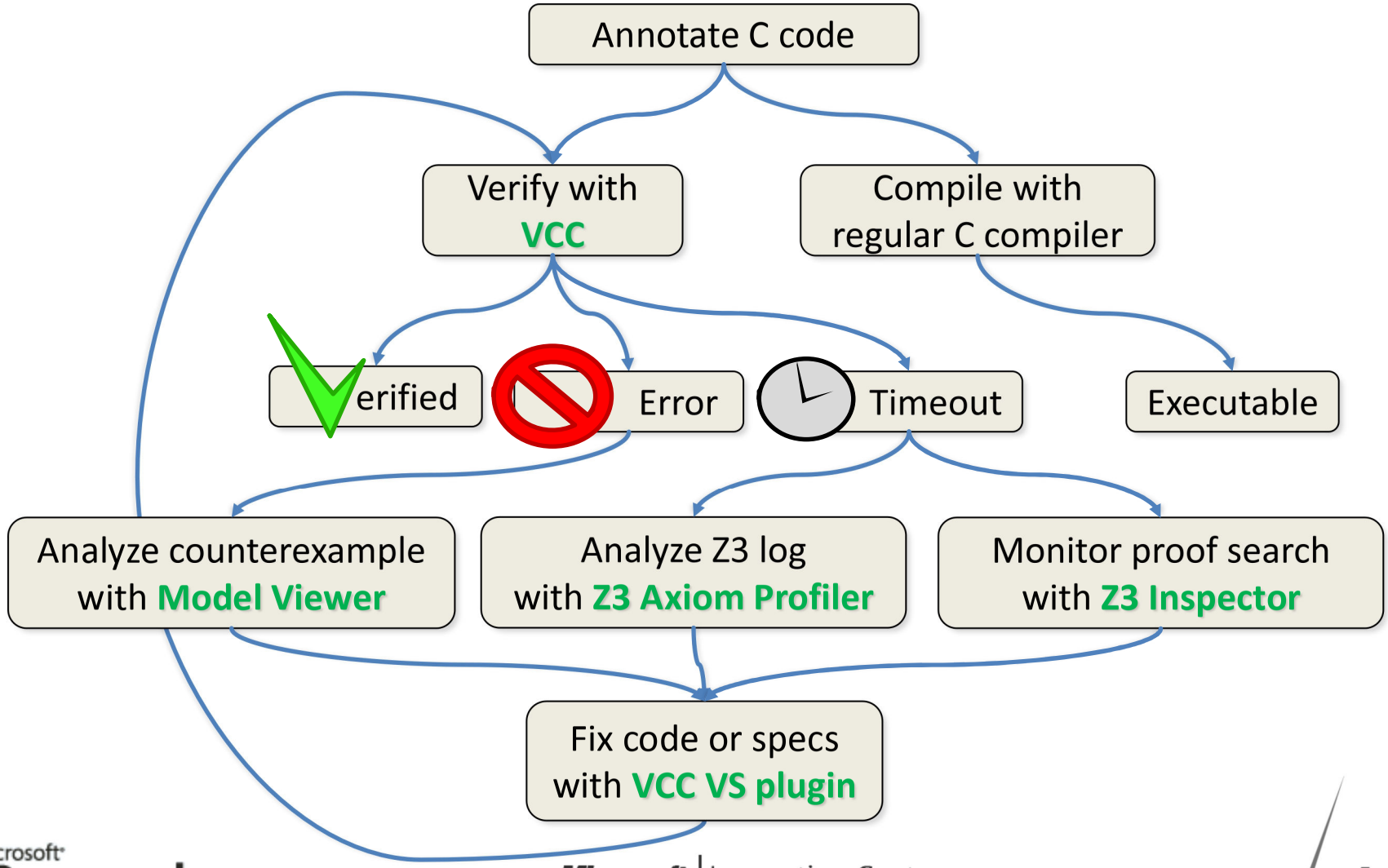
# Methodology is not Enough

**Most verification attempts fail!**

A practically useful verification tool supports

- **concise specification** of relevant properties
  - methodology and annotation language
- **meaningful feedback** for failed proof attempts
  - error model mapped back to source code
  - profile of proof search when hitting resource constraints
  - live monitoring of prover work for long running proofs
- **acceptable turnaround times** for verify&fix-cycles

# VCC WORKFLOW




Microsoft Research

Microsoft Innovation Center Europe

Verisoft XT

# Performance, Performance, Performance

Experience from the Hyper-V verification

- **successful** verifications:
  - 0.5 – 500 sec
  - all time max: 50,000 sec
  - typical: 10 – 20 sec
- failing proof attempts **often take much longer** than the finally successful verification
- acceptable time for **interactive** work: < 30 sec

# SUMMARY / OUTLOOK

Microsoft Research

Microsoft Innovation Center Europe

Verisoft XT

# VCC TODAY

- **Modular verification**
  - data: invariant admissibility
  - functions: contracts, framing
- **Low-level C**
  - Bit fields, unions, machine arithmetic, lock-free algorithms, hw access
- **Concurrency**
  - two-state invariants, claims, volatility, atomicity
- **Usability**
  - integration in Visual Studio
  - Model Viewer, Z3 Axiom Profiler, Z3 Inspector
- **Scales** to Hyper-V verification

http://research.microsoft.com/projects/vcc/

Microsoft Research

Microsoft Innovation Center Europe

Verisoft XT