VCC: A Practical System for Verifying Concurrent C

Ernie Cohen[1], Markus Dahlweid[2], Mark Hillebrand[3], Dirk Leinenbach[3], Michal Moskal[2], Thomas Santen[2], Wolfram Schulte[4] and Stephan Tobies[2]

[1]Microsoft Corporation, Redmond, WA, USA
[2]European Microsoft Innovation Center, Aachen, Germany
[3]German Research Center for Artificial Intelligence (DFKI), Saarbrucken, Germany
[4]Microsoft Research, Redmond, WA, USA

May 6, 2013

Pavol Bielik

Introduction

- VCC stands for Verifying C Compiler
- deductive verifier for concurrent C code
- performs static modular analysis and sound verification of functional properties of low-level concurrent C code
- VCC translates annotated C code into BoogiePL
  - Boogie translates BoogiePL into verification conditions
  - Z3 solves them or gives couterexamples

Motivation

- driven by the verification of the Microsoft Hyper-V hypervisor
- the hypervisor turns a single real multi-processor x64 machine into a number of virtual multiprocessor x64 machines
- own concurrency control primitives, complex data structures and dozens of tricky optimizations
- written in no verification in mind (100KLOC of C, 5KLOC of assembly)
- performance was of a great importance

Function Contracts

Specification (Contract) consisting of four kinds of clauses

- preconditions: _(*requires* P)
- postconditions: _(*ensures* Q)
- writes clause: _(*writes* S)
- termination: _(*decreases* T)

Modular – only looks at function specification

Function Contracts

```
int min(int a, int b)
_(requires \true)
_(ensures \result <= a && \result <= b)
_(ensures \result == a || \result == b)
{
  if (a <= b)
  return a;
  else return b;
}
```

Pure functions

```
_(pure) int min(int a, int b) ...
```

- no side effects on programs state
- not allowed to allocate memory
- can only write to local variables
- can be called within VCC annotations

However:

- empty writes clause $\not\Rightarrow$ pure

Pure Ghost functions

- used only in specification
- along with other annotations removed before compilation by preprocessor

```
_(def \bool sorted(int *arr, unsigned len) {
  return \forall unsigned i, j;
    i <= j && j < len ==> arr[i] <= arr[j];
})

void sort(int *arr, unsigned len)
_(writes \array_range(arr, len))
_(ensures sorted(arr, len))
```

Type Invariants

- object invariants can be associated with compound types (structs and unions)
- support for both single and two-state predicates

```
#define SSTR_MAXLEN 100
typedef struct SafeString {
  unsigned len;
  char content[SSTR_MAXLEN + 1];
  _(invariant \this->len <= SSTR_MAXLEN)
  _(invariant content[len] == '\0')
} SafeString;
```

Type Invariants

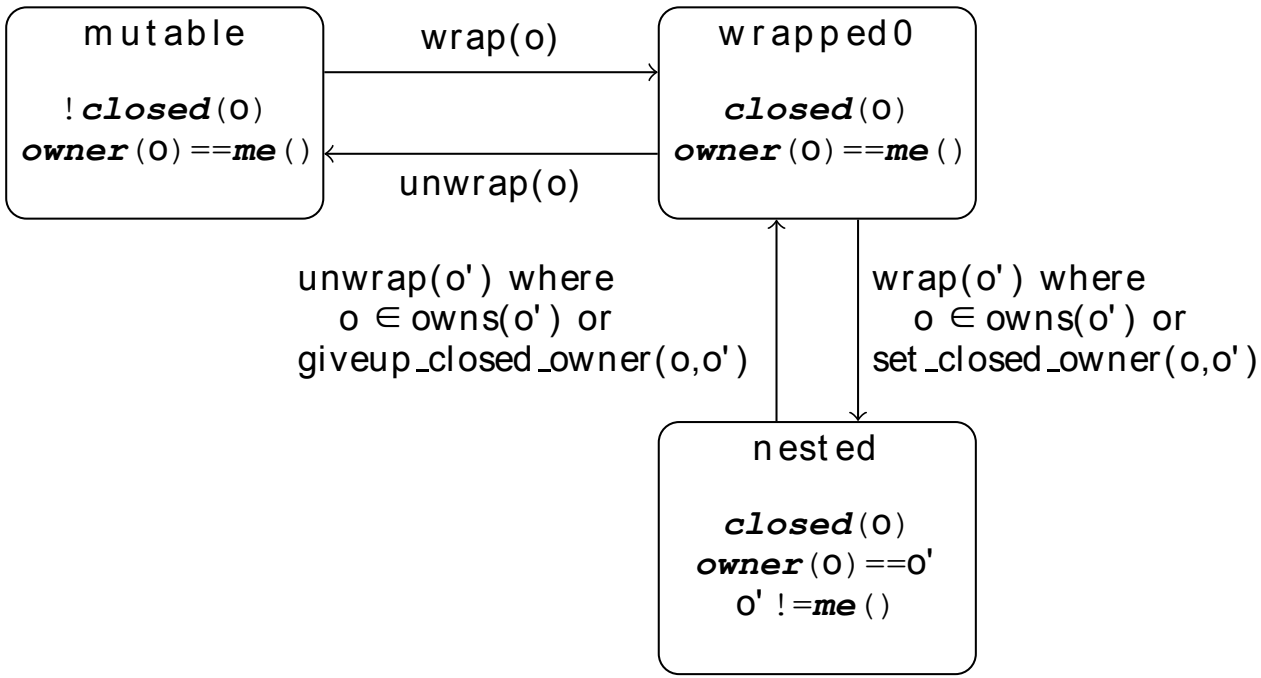
- object invariants cannot hold at all times, e.g.:
  - initialization
  - destruction
  - updates
- an object can be in two states controlled by *closed* ghost field
  - **closed** – invariant holds but non-volatile fields can not be changed
  - **open** – invariant can not be assumed, but fields can be changed
- closedness is manipulated using *wrap* and *unwrap* helper methods
- type invariants are coupled with ownership

Ownership

- ownership is expressed by adding a ghost field to every object and making it point to object owner
- the roots of trees in the ownership forest are objects representing threads of execution.
  - threads are always closed, and own themselves
- the set of objects directly or transitively owned by an object is called the ownership domain of that object

Ownership

- type invariants are coupled with ownership
- if an object is owned by a thread, only that thread can change its (nonvolatile) fields (and then only if the object is open), wrap or unwrap it, or change its owner to another object



Wrapping & Unwrapping Example

```
1  void sstr_append(struct SafeString *s, char c)
2    _(maintains \wrapped(s))
3    _(requires s->len < SSTR_MAXLEN)
4    _(ensures s->len == \old(s->len) + 1)
5    _(ensures s->content[\old(s->len)] == c)
6    ...
7    _(writes s)
8  {
9    _(unwrap s)
10   s->content[s->len++] = c;
11   s->content[s->len] = '\0';
12   _(wrap s)
13 }
```

Verifying Concurrent Programs

- coarse-grained concurrency - ownership based
- fine-grained concurrency - atomic actions on volatile fields

Volatile fields

C meaning:

- value can be changed from "outside"
- prevents compiler from storing value in registers

VCC meaning:

- field can be modified while the object is closed, as long as the update occurs inside an explicit atomic action that preserves the object invariant
- thread forgets the values of these fields when it makes an impure function call and just before an atomic action

Spin-lock

- ownership used to control access to shared resource

```
_(volatile_owns) struct Lock {
volatile int locked;
_(ghost \object protected_obj;)
_(invariant locked == 0 ==> \mine(protected_obj))
};
```

Spin-lock - Initialization

```
void InitializeLock(struct Lock *l
                    _(ghost \object obj))
_(requires \wrapped(obj))
_(ensures \wrapped(l) && l->protected_obj == obj )
_(ensures \nested(obj))
_(writes \span(l), obj)
{
  l->locked = 0;
  _(ghost {
    l->protected_obj = obj;
    l->\owns = {obj};
    _(wrap l)
  })
}
```

Spin-lock - Acquire

```
void Acquire(struct Lock *l)
_(maintains \wrapped(l))
_(ensures l->locked == 1)
_(ensures \wrapped(l->protected_obj) && \fresh(l->protected_obj))
{
  int stop = 0;
  do {
    _(atomic l) {
      stop = InterlockedCompareExchange(&l->locked, 1, 0) == 0;
      _(ghost if (stop) l->\owns -= l->protected_obj)
    }
  } while (!stop);
}
```

Spin-lock - Release

```
void Release(struct Lock *l)
_(maintains \wrapped(l))
_(requires l->locked == 1)
_(requires \wrapped(l->protected_obj))
_(ensures l->locked == 0
_(ensures \nested(l->protected_obj))
_(writes l->protected_obj)
{
  _(atomic l) {
    l->locked = 0;
    _(ghost l->\owns += l->protected_obj)
  }
}
```

Spin-lock - Problems

- acquire and release require that the lock is closed

```
void Acquire(struct Lock *l)
_(requires \wrapped(l)) ...
```

- definition of wrapped:

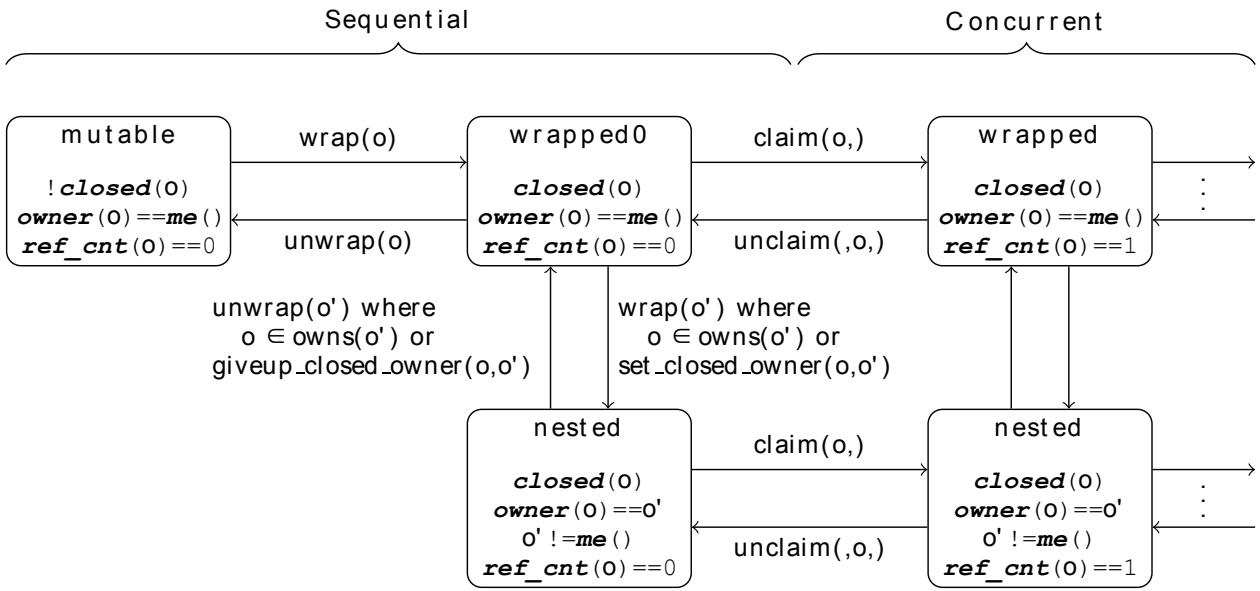
```
\bool \wrapped(\object o)
_(ensures \result <==> o->\owner == \me && o->\closed)
```

- $o \rightarrow \backslash owner == \backslash me$ is satisfiable by only single thread

Claims

```
_(ghost typedef struct {
  \ptrset claimed;
  _(invariant \forall \object o; o \in claimed ==> o->\closed)
} \claim; )
```

- Ownership meta-states updated:



Spin-lock revisited

```
void Release(struct Lock *l _(ghost \claim c))
_(maintains \wrapped(c) && \claims_object(c, l))
_(requires l->locked == 1 )
_(requires \wrapped(l->protected_obj))
_(ensures l->locked == 0 )
_(ensures \nested(l->protected_obj))
_(writes l->protected_obj)
{
  _(atomic c, l) {
    l->locked = 0;
    _(ghost l->\owns += l->protected_obj)
  }
}
```

Invariant contraints

- what part of state are invariants allowed to mention
- how to make sure that updates dont break out of scope invariants?
- VCC allows invariants to mention arbitrary parts of the state, but requires them to be admissible
- VCC checks that no object invariant can be broken by invariant-preserving changes to other objects

```
struct Counter {
int n;
_(invariant n = old(n) ||
            n = old(n) + 2)
};
```

```
struct Low {
Counter cnt;
int floor;
_(invariant floor <= cnt.n)
};

struct High {
Counter cnt;
int ceiling;
_(invariant cnt.n <= ceiling)
};
```

Locally checked invariant

- VCC enforces that all invariants are **reflexive**.

$$refl(\tau) \equiv \forall p, h_o, h.type(p) = \tau \wedge inv_\tau(h_o, h, p) \Rightarrow inv_\tau(h, h, p)$$

- an action is **legal** iff it preserves the invariants of updated objects

$$legal(h_o, h) \equiv safe(h_o) \Rightarrow \forall p.h_o[p] = h[p] \vee inv(h_o, h, p)$$

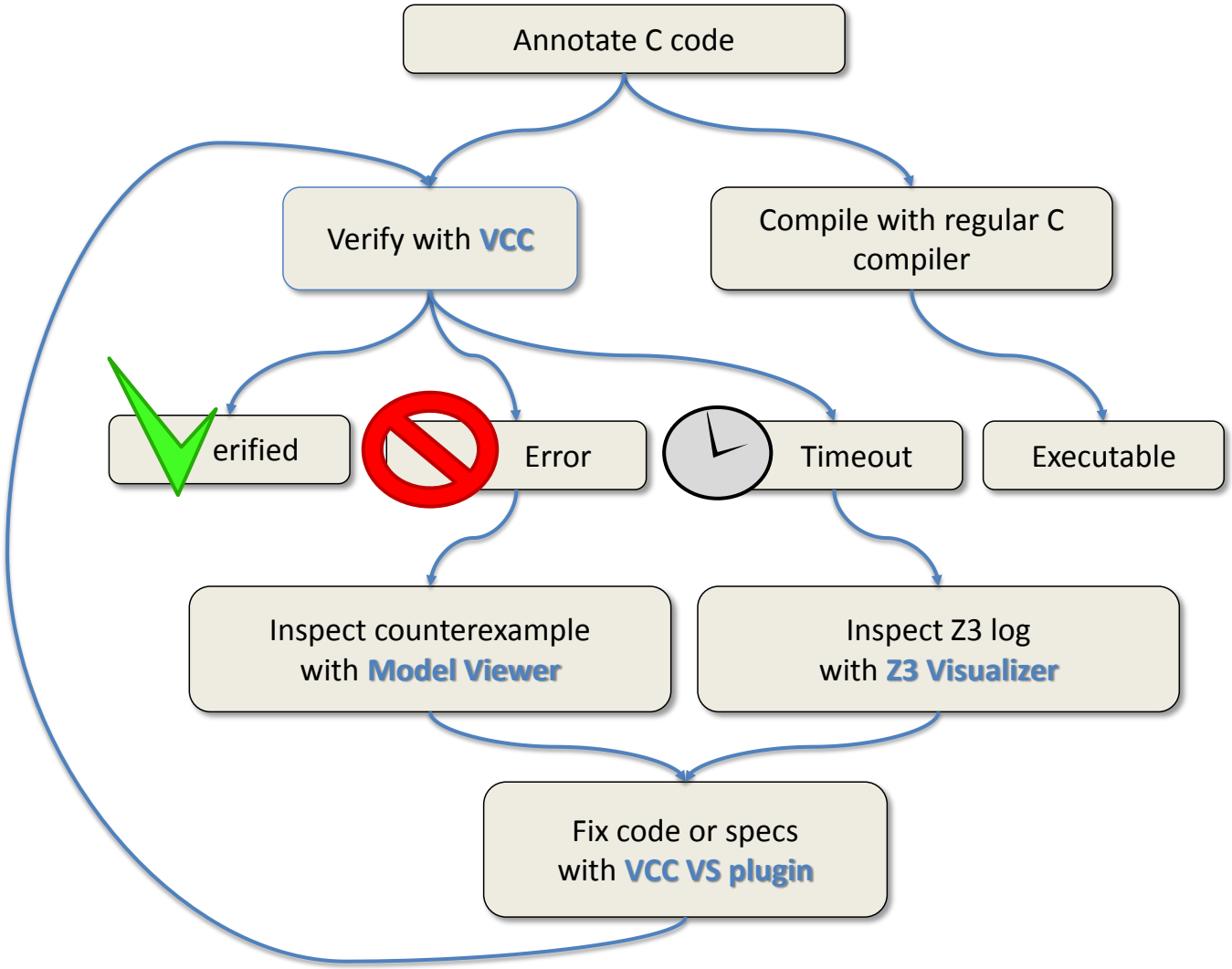
- a **stable** invariant is one that cannot be broken by legal actions

$$stable(\tau) \equiv \forall p, h_o h.type(p) = \tau \wedge safe(h_o) \wedge legal(h_o, h) \Rightarrow inv_\tau(h_o, h, p)$$

- An **admissible** invariant is one that is stable and reflexive

$$adm(\tau) \equiv stable(\tau) \wedge refl(\tau)$$

VCC Workflow



source: http://research.microsoft.com/en-us/projects/vcc/

Microsoft Hyper-V hypervisor Experience

- implementation of the SPT algorithm contains $\approx$ 700 lines of C code
- $\approx$ 4000 lines of the annotations
- overall proof time is $\approx$ 18 hours on one core of 2GHz Intel Core 2 Duo machine
- most functions in 0.5 to 500 seconds with an average of $\approx$ 25 seconds
- estimated person effort is $\approx$ 1.5 person-years, including VCC learning period

Conclusion

- VCC follows largely the design of Spec#
- expressive enough for industrial program verification
- lot's of helper methods
- up to the user to guarantee that access annotated as \atomic is indeed atomic
- assumes sequential consistency

Future work

- incorporate x86 memory model
- annotation overhead
- performance

References


Ernie Cohen, Markus Dahlweid, Mark Hillebrand, et. al.,
"VCC: A practical system for verifying concurrent C,"
Artificial Intelligence, vol. 5674, no. 1753, pp. 23–42, 2009.


Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies,
"Local Verification of Global Invariants in Concurrent Programs,"
Innovation, vol. 6174, pp. 480–494, 2010.


Ernie Cohen and Mark A Hillebrand,
"The VCC Manual & Verifying C Programs : A VCC Tutorial,"
Tech. Rep., 2012.


Ernie Cohen, Michał Moskal, Stephan Tobies, and Wolfram Schulte,
"A precise yet efficient memory model for C,"
Electron. Notes Theor. Comput. Sci., vol. 254, pp. 85–103, Oct. 2009.