Local Verification of Global Invariants in Concurrent Programs

http://d3s.mff.cuni.cz

Department of Distributed and Dependable Systems

D3S

Reading Seminar

David Hauzar

CHARLES UNIVERSITY IN PRAGUE

faculty of mathematics and physics

Reffered paper:

- Ernie Cohen, Michal Moskal, Wolfram Schulte, Stephan Tobies. Local Verification of Global Invariants in Concurrent Programs. Springer Verlag 2010
  - Microsoft research

1. MOTIVATION

Motivation

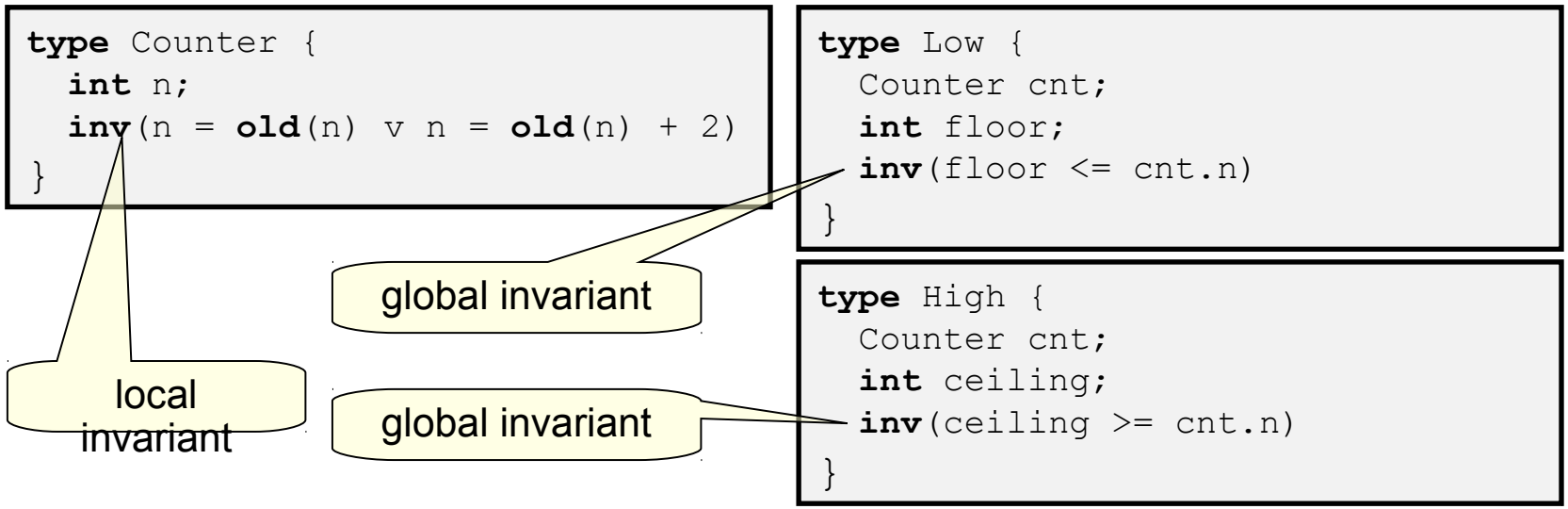
- Annotate code with invariants, check whether invariants hold
- VCC tool built for verification of the Microsoft Hyper-V hypervisor
  - about 100K lines of C code
  - number of concurrency control mechanisms and algorithms
- It was possible to express all required specifications
- It is efficient
  - turnaround times below 2 hours
  - memory limit never exceedes 1GB

Overview

- Admissibility condition on two-state invariants
  - updates can be checked locally
  - verifier can be built upon an established condition generator and theorem prover using first order logic
- Encoding higher level constructs including validity, ownership and handles
- Verifying concurrent programs
  - thread local data used for disabling undesirable interferences
  - claims used to workaround limitations of first-order logic

Invariants

- Focus on *two state invariants*
  - connections between current and previous state referred as `old`
- *Local invariants*
  - no references to other objects
- *Global invariants*
  - it is generally necessary to check all invariants on update of any object
  - does not scale for large systems



Locally checked invariants (LCI)

- Global invariants that can be checked locally
  - *admissible invariants*
  - check only invariant of updated object
  - based on first order logic
    - verifier can be built upon an established condition generator and theorem prover

```
type Counter {
  int n;
  inv(n = old(n) v n = old(n) + 2)
}
```

```
type Low {
  Counter cnt;
  int floor;
  inv(floor <= cnt.n)
}
```

```
type High {
  Counter cnt;
  int ceiling;
  inv(ceiling >= cnt.n)
}
```

global invariant that can be checked locally

global invariant that can not be checked locally

2. ADMISSIBILITY CRITERION AND CHECKING USING LCI

LCI – definitions (1)

- An action is *safe* iff it satisfies invariants of all objects
- An action is *legal* iff it preserves invariants of updated objects

```
Low low;
low.cnt = 1; low.cnt.floor = 0;
low.cnt.n = low.cnt.n  - 1;   // not legal and not safe
low.cnt.n = low.cnt.n  + 2;   // legal and safe

High high;
high.cnt = 1; high.cnt.ceiling = 2;
high.cnt.n = high.cnt.n + 2;  // legal but not safe
```

```
type Low {
  Counter cnt;
  int floor;
  inv(floor <= cnt.n)
}
```

```
type Counter {
  int n;
  inv(n = old(n) v n = old(n) + 2)
}
```

```
type High {
  Counter cnt;
  int ceiling;
  inv(ceiling >= cnt.n)
}
```

LCI – definitions (2)

- A state is *safe* iff the action that goes from that state to itself is safe
  - the action is stuttering action, it does nothing
- An invariant is reflexive iff an action $<h_0, h>$ satisfies the invariant implies that action <h, h> also satisfies the invariant
  - invariants of type `n > old(n)` are not reflexive
  - all invariants are reflexive and all satisfied implies the state is safe

```
Low low;
low.cnt = 1; low.cnt.floor = 0;
High high;
high.cnt = 1; high.cnt.ceiling = 2;
// safe state

high.cnt = high.cnt+2; // legal but not safe action
// unsafe state

high.cnt = 0;          // not legal, not safe action
// safe state
```

```
type Low {
  Counter cnt;
  int floor;
  inv(floor <= cnt.n)
}
```

```
type High {
  Counter cnt;
  int ceiling;
  inv(ceiling >= cnt.n)
}
```

```
type Counter {
  int n;
  inv(n = old(n) v n = old(n) + 2)
}
```

LCI – definitions (3)

- An invariant is stable iff it cannot be broken by a legal action
- An invariant is admissible iff it is stable and reflexive

```
type Counter {
  int n;
  inv(n = old(n) v n = old(n) + 2)
}
```

```
type Low {
  Counter cnt;
  int floor;
  inv(floor <= cnt.n)
}
```

admissible invariant

```
type High {
  Counter cnt;
  int ceiling;
  inv(ceiling >= cnt.n)
}
```

inadmissible invariant

Checking LCI

- An action is safe iff satisfies the invariant of every object
- A state is safe iff the action that goes from that state to itself is safe
- An invariant is reflexive iff an action $<h_0, h>$ satisfies the invariant implies that the state h is safe
  - invariants of type `n > old(n)` are not reflexive
- An action is legal iff it preserves invariants of updated objects
- An invariant is stable iff it cannot be broken by legal actions
- An invariant is admissible iff it is stable and reflexive

To get safety of actions it suffices to prove that:

- initial state is safe
- actions are legal
  - depends only on invariants of objects that are updated
- invariants are stable
  - depends only on invariants of data reference from the current invariant
- invariants are reflexive

Expressibility of LCI

- any invariant can be made admissible
  - objects that the invariant depends on have to explicitly reference it

```
type Counter {
  int n;
  inv(n = old(n) v n = old(n) + 2)
}
```

```
type High {
  Counter cnt;
  int ceiling;
  inv(ceiling >= cnt.n)
}
```

```
type Counter2 {
  int n;
  object b;
  inv(n = old(n) v n = old(n) + 2)
  inv(b == old(b))
  inv(n == old(n) v inv(b))
}
```

```
type High2 {
  Counter2 cnt;
  int ceiling;
  inv(cnt.b = this)
  inv(ceiling >= cnt.n)
}
```

Department of Distributed and Dependable Systems D3S

3. ENCODING HIGHER-LEVEL METHODOLOGIES USING LCI

Encoding higher-level methodologies

- LCI is easy to check, expressible
- Too much low level, high annotation overhead
- Encode higher-level constructs using LCI
  - syntactic sugar to encode effective abstractions and abbreviations
  - **Validity**
  - **Ownership**
  - **Handles**

Ghost state and ghost code

- Method
  - apply syntactic transformation to every user-defined type
  - adds additional fields and invariants
  - weakens user-defined invariant ψ
- Ghost state
  - used to encode higher level constructs
  - aids reasoning
- Ghost code
  - code that references ghost states
  - cannot update non-ghost states
  - have to be always terminating
- Ghost state and ghost code can be erased without effecting behavior of the program on non-ghost state.

Validity

- Initial state as well as all following states must be safe
- Invariants sometimes do not hold
  - prior to initialization and after destruction of objects

How to temporary disable invariant?

- introduce new field `valid`

```
type t {
  ghost bool valid;
  inv((old(valid) v valid) → Ψ)
}
```

Validity (cont)

- With introduction of validity, invariant of type Low is not inadmissible
  - if `cnt.valid = false`, counter can go down
- It is necessary to assure, that `low.cnt` is valid

```
type Counter {
  int n;
  inv(n = old(n) v n = old(n) + 2)
}
```

```
type t {
  ghost bool valid;
  inv((old(valid) v valid) → Ψ)
}
```

```
type Low {
  Counter cnt;
  int floor;
  inv(floor <= cnt.n)
}
```

Ownership

- each object has unique owner
- if the type of owner is different from `Thread`, object must be valid
- owner of object can be changed

```
type t {
  ghost bool valid;
  inv((old(valid) v valid) → Ψ)

  ghost OwnerCtrl ctrl;
  inv(unchg(ctrl))
  inv(ctrl.subject = this)
  inv(unchg(valid) v inv(ctrl))
}
```

```
ghost type OwnerCtrl {
  object owner, subject;
  inv(unchg(subject))
  inv(unchg(owner) v inv(owner))
  inv(unchg(owner) v inv(old(owner)))
  inv(unchg(subject.valid) v inv(owner))
  inv(type(owner) = Thread v
subject.valid)
  inv(subject.ctrl = this)
}
```

low is owner of its counter cnt

Counter cnt cannot be shared!

```
type Low {
  Counter cnt;
  int floor;
  inv(floor <= cnt.n)
  inv(cnt.ctrl.owner = this)
}
```

Handles

- multiple clients can each own valid handles on the shared object
  - each handle guarantees validity of the shared object

```
ghost type OwnerCtrl {
  object owner, subject;
  inv(unchg(subject))
  inv(unchg(owner) v inv(owner))
  inv(unchg(owner) v inv(old(owner)))
  inv(unchg(subject.valid) v inv(owner))
  inv(type(owner) = Thread v subject.valid)
  inv(subject.ctrl = this)

  set<Handle h;
  inv(unchg(handles) v inv(owner))
  inv(for all(Handle h;
   h in old(handles) && h not in handles
      → ¬h.valid))
  inv(handles = {} v subject.valid)
}
```

```
ghost type Handle {
  object obj;
  inv(unchg(obj) &&
    this in obj.cnt.handles &&
    obj.valid)
}
```

```
type Low2 {
  Counter cnt;
  int floor;
  inv(floor <= cnt.n)
  ghost Handle cntH;
  inv(cntH.ctrl.owner = this &&
    cntH.obj = cnt)
}
```

Department of Distributed and Dependable Systems D3S

4. VERIFYING CONCURRENT PROGRAMS

Verifying concurrent programs

„How prove that all actions of a procedure in concurrent program are legal?“

- procedures are sequences of atomic actions performed by a single thread (denoted by **me**)
  - possible interferences of safe actions of other threads in between
- Verifying procedure
  - check legality of all its actions assuming safety of interfering actions
  - without looking at the code of other procedures
- Example:
  - suppose that `c` remains valid throughout execution of `inc`
  - suppose that theorem prover can automatically infer all properties of data

```
void incr(Counter c) {
  < a := c.n; >
  < if (c.n = a) c.n := a+2; >
  < b := c.n; assert(a<b); >
}
```

1. Check the legality of a := c.n.
2. Simulate arbitrary, but safe, interference by other actions by assigning safe value to the heap.
3. Check the legality of second action performed on new heap.

…

Local variables remain unchanged between actions of the current thread and c.n can only grow implies that the assertion holds.

Thread local data

"How force `c` to remain valid throughout execution of `inc`?"

```
void incr(Counter c) {
  < a := c.n; >
  < if (c.n = a) c.n := a+2; >
  < b := c.n; assert(a<b); >
}
```

Field o.f is thread local data of thread t iff the invariant of t can admissibly prevent any other threads from changing it.

- How make field f of object o thread local providing that owner of o is thread?
  - **inv**(**unchg**(f) v **inv**(ctr.owner))
- Field o.f is thread local to thread t. How does admissible invariant that prevents o.f from changing by other threads look like?
  - **inv**(**unchg**(**this**.state) && Φ → **unchg**(o.f))

Thread local data (example)

- How force `c` to remain valid throughout execution of `incr`?

```
void incr(Counter c) {
  < a := c.n; >
  < if (c.n = a) c.n := a+2; >
  < b := c.n; assert(a<b); >
}
```

```
ghost type OwnerCtrl {
  …
  inv(unchg(subject.valid) v inv(owner))
  …
}
```

1. Make actual thread own `c`: `c.ctrl.owner = me`
   - implies that `c.valid` and `c.ctrl.owner` are thread local to `me`

1. Add (admissible) invariant that forces `c` to remain valid to `Thread`

- How to make it possible that multiple threads can execute procedure `incr` concurrently on the same `c`?
  - use Handles instead of Ownership

Claims

"Suppose that theorem prover can automatically infer all properties of data."

- e.g. "c.n can only grow"

- Preceding presumption does not hold!

```
type Counter {
  int n;
  inv(n = old(n) v n = old(n) + 2)
}
```

```
void incr(Counter c) {
  < a := c.n; >
  < if (c.n = a) c.n := a+2; >
  < b := c.n; assert(a<b); >
}
```

Claim is ghost object with invariant that represents property that theorem prover cannot infer

Claims - example

```
type Counter {
  int n;
  inv(n = old(n) v n = old(n) + 2)
}
```

```
type Low2 {
  Counter cnt;
  int floor;
  inv(floor <= cnt.n)
  ghost Handle cntH;
  inv(cntH.ctrl.owner = this &&
    cntH.obj = cnt)
  inv((unchg(floor) && unchg(cntH) && unchg(cnt))
    v inv(ctrl.owner))
}
```

```
void incr(Counter c, ghost Handle h, ghost Low2 cl)
  requires(h.obj = c && h.ctrl.owner = me &&
h.valid)
  requires(¬cl.valid && cl.ctrl.owner = me) {

  < a := c.n; ghost
    { cl.cnt = c; cl.cntH = h; h.ctrl.owner := cl;
      cl.floor := a; cl.valid := true;}
  >
  < if (c.n = a) c.n := a+2;
    ghost { cl.floor := a + 1; }>
  < b := c.n; assert(a<b); >
}
```

Make fields of Low2 thread local to owning thread.

Make fields of Low2 local to actual thrad

Make the field c.valid local to actual thread

Initialize the Low2 claim cl with lower bound for c

a+1 is new lower bound for c

Follows from cl's invariant – fields of cl are thread local

5. EVALUATION

Evaluation

- Verification of the Microsoft Hyper-V hypervisor
  - about 100K lines of C code
  - number of concurrency control mechanisms and algorithms
- It was possible to express all required specifications
- Turnaround times below 2 hours
- For most problems, a memory limit of 200MB suffices
  - memory limit never exceeds 1GB
- High annotation overhead
  - one line of annotation per line of code
- Typical work flow
  - running verification tool on the initial version of the code and specification
  - fixing either the specification or the code

Thank you for your attention

- Paper:
  - Ernie Cohen, Michal Moskal, Wolfram Schulte, Stephan Tobies. Local Verification of Global Invariants in Concurrent Programs. Springer Verlag 2010
- VCC tool
  - http://research.microsoft.com/en-us/projects/vcc/