ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Local Verification of Global Invariants in Concurrent Programs

Ernie Cohen[1], Michal Moskal[2], Wolfram Schulte[2], Stephan Tobies[1]

[1] European Microsoft Innovation Center, Aachen
[2] Microsoft Research, Redmond

Presentation: Florian Besser

Overview

- Motivation
- Requirements
- Promises
- Definitions, Lemmas and Theorems
- Recursive Invariants / Ghost State
- Verify Procedure / Claims
- Conclusion

Motivation

- Invariant checking can be problematic in concurrent programs.
- Two extreme cases:
  - High concurrency / efficiency needed.
  - Invariants spanning multiple objects.
- Locking all involved objects an inefficient way, especially at run-time.
- When proving (partial) correctness, proving an invariant that only spans the current class is simpler.

Motivation – existing “Solutions”

- Spec#
  - Every object is either valid or mutable.
  - Uses **expose** block to make object mutable.
  - Checks invariants after the **expose** block.
  - To accommodate invariants spanning multiple objects, an **expose**(*this*) must also recursively expose all owners of *this*. Later, all invariants of the exposed objects have to be re-checked.
- Separation logic / Rely-Guarantee
  - High complexity, not as far developed as conventional ways.

Motivation – the LCI Solution

- A system so that locally checking invariants is enough to satisfy global invariants.
- Then **prove** the invariants.

- Uses **actions** which can update any number of objects.
- After each **action**, invariants must hold.

- Think of actions as feature applications in Eiffel.
- (Eiffel checks invariants at run-time, while LCI does so without running the code.)

LCI Requirements

- Definition of an action:

  A pair of states, representing a transmission from pre- to post-state, denoted $< h_0 , h >$

- Actions are **safe** iff they satisfy every invariant of every object.
- Actions are **legal** iff they satisfy every invariant of every updated object.

- A state is **safe** iff < h, h > is **safe**.
- Must start in a **safe** state (later dropped).

LCI Requirements

- Invariants must be **reflexive**: If $< h_0 , h >$ satisfies the invariant, so must $< h, h >$
- Invariants must be **stable**: They can't be broken by **legal** actions.
- Invariants that are both **stable** and **reflexive** are called **admissible**.

LCI Promises

- If all invariants are **admissible** then every **legal** action (from a **safe** pre-state) is **safe** and has a **safe** post-state.
- In short: The program is partially correct (it might not terminate)

- How is that promise achieved:
  - **Prove** admissibility of every invariant.
  - **Prove** that every action produced by the program is legal.

A simple Example

```
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(cnt.n ≤ ceiling)
}
```

- Invariant of **High** is **not admissible**!
- A **legal** action on **Counter** cnt can break the invariant of **High**.

Definitions

A condensed recap of the last few slides:

$$
\begin{array}{llll}
 & & \mathsf{type} & : \mathbb{Z} \to \mathbb{T} \\
\text{fields} & \mathbb{F} \equiv \{\mathsf{f}_0, \mathsf{f}_1, \dots\} & \mathsf{inv}_\tau & : \mathbb{H} \times \mathbb{H} \times \mathbb{Z} \to \mathbb{B} \quad \text{for } \tau \in \mathbb{T} \\
\text{types} & \mathbb{T} \equiv \{\mathsf{t}_0, \mathsf{t}_1, \dots\} & \mathsf{inv}(h_o, h, p) & \equiv \mathsf{inv}_{\mathsf{type}(p)}(h_o, h, p) \\
\text{integers} & \mathbb{Z} \equiv \{0, 1, -1, \dots\} & \mathsf{inv}_1(h, p) & \equiv \mathsf{inv}(h, h, p) \\
\text{heaps} & \mathbb{H} \equiv \mathbb{Z} \to (\mathbb{F} \to \mathbb{Z}) & \mathsf{legal}(h_o, h) & \equiv \mathsf{safe}_1(h_o) \Rightarrow \forall p.\ h_o[p] = h[p] \vee \mathsf{inv}(h_o, h, p) \\
\text{Booleans} & \mathbb{B} \equiv \{\textbf{true}, \textbf{false}\} & \mathsf{safe}(h_o, h) & \equiv \forall p.\ \mathsf{inv}(h_o, h, p) \\
 & & \mathsf{safe}_1(h) & \equiv \forall p.\ \mathsf{inv}_1(h, p)
\end{array}
$$

$$\mathsf{stable}(\tau) \equiv \forall p, h_o, h.\ \mathsf{type}(p) = \tau \wedge \mathsf{safe}_1(h_o) \wedge \mathsf{legal}(h_o, h) \Rightarrow \mathsf{inv}_\tau(h_o, h, p)$$

$$\mathsf{refl}(\tau) \equiv \forall p, h_o, h.\ \mathsf{type}(p) = \tau \wedge \mathsf{inv}_\tau(h_o, h, p) \Rightarrow \mathsf{inv}_\tau(h, h, p)$$

$$\mathsf{adm}(\tau) \equiv \mathsf{stable}(\tau) \wedge \mathsf{refl}(\tau)$$

Heaps **H** map integers (i.e., addresses) to objects, which are maps from field names to integers. The invariant function inv(h_0 , h, p) returns true iff the action changing the state from h_0 to h satisfies the invariant of (the object referenced by) p. For simplicity, the type of an object at a given address (given by the type function) is fixed. The inv function is constructed from type-specific invariants (inv_τ).

Lemmas and Theorems

A condensed recap of the last few slides:

Lemma 1. $(\forall \tau.\, \mathsf{refl}(\tau)) \Rightarrow \mathsf{safe}(h_o, h) \Rightarrow \mathsf{safe}_1(h)$

Lemma 2. $(\forall \tau.\, \mathsf{stable}(\tau)) \Rightarrow (\mathsf{safe}_1(h_o) \wedge \mathsf{legal}(h_o, h) \Rightarrow \mathsf{safe}(h_o, h))$

Theorem 1. *Let all types be admissible. Then, for a sequence of heaps* $h_0, h_1, \ldots$:

$$\mathsf{safe}_1(h_0) \wedge (\forall i.\, \mathsf{legal}(h_i, h_{i+1})) \Rightarrow (\forall i.\, \mathsf{safe}_1(h_i) \wedge \mathsf{safe}(h_i, h_{i+1}))$$

LCI extended – recursive Invariants

```
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.n ≤ ceiling)
}
```

- Invariant of **High2** is now admissible!
- An action on **Counter2** must fulfill the invariant of **Counter2** and as such also the invariant of **High2**.
- When checking invariant admissibility use fixpoint iteration.

LCI extended – Ghost State

- Ghost code can be removed without affecting functionality.
- Every instance of a user-defined object gets a ghost OwnerCtrl object attached.

- Problem: The initial state must be safe, but objects being created or destroyed might not satisfy this.
- Solution: Introduce **ghost bool valid** as a field for every object. **Valid** implies invariant. **Valid** is initially false. Set **valid** to true after creation, set it to false before destruction.

LCI extended – Ghost State

- Problem: An object is shared, what if someone destroys it?
- Solution: Ownership. Every object must have a unique owner. Transfer of ownership is possible, but requires an invariant check of both the old and the new owner.

- Problem: Ownership does not allow an object to be shared.
- Solution: Handles. Multiple clients can have handles on an object, and the handle's invariant guarantees the object's validity. The object owner keeps track of the handles, and can for example implement a simple read-write lock with a single **ghost int**.

LCI extended – Definitions

A condensed recap of the last few slides:

```
1  type τ {
2    F
3
4    // Validity, Sect. 4.1
5    ghost bool valid;
6    inv((old(valid) ∨ valid) ⇒ ψ)
7
8    // Ownership, Sect. 4.2
9    ghost OwnerCtrl ctrl;
10   inv(unchg(ctrl))
11   inv(ctrl.subject = this)
12   inv(unchg(valid) ∨ inv(ctrl))
13   // for every f ∈ F
14   inv(¬valid ⇒
15     unchg(f) ∨ inv(ctrl.owner))
16 }
```

```
16 ghost type OwnerCtrl {
17   object owner, subject;
18   inv(unchg(subject))
19   inv(unchg(owner) ∨ inv(owner))
20   inv(unchg(owner) ∨ inv(old(owner)))
21   inv(unchg(subject.valid) ∨ inv(owner))
22   inv(type(owner) = Thread ∨ subject.valid)
23   inv(subject.ctrl = this)
24   // Handles, Sect. 4.3
25   set<Handle> handles;
26   inv(unchg(handles) ∨ inv(owner))
27   inv(∀(Handle h;
28         h ∈ old(handles) ∧ h ∉ handles
29           ⇒ ¬h.valid))
30   inv(handles = {} ∨ subject.valid)
31 }
```

LCI extended – Lock Example

```
type Lock {
  bool locked;
  ghost object rsc;
  inv(unchg(rsc))
  inv(¬locked ⇒
    rsc.ctrl.owner = this)
}

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

```
void Acquire(Lock l, ghost Handle h)
  requires(h.ctrl.owner = me ∧ h.valid ∧ h.obj = l)
  ensures(l.rsc.ctrl.owner = me)
{
  do ⟨ prev := l.locked;
       if (¬prev) {
         l.locked := true;
         ghost { l.rsc.ctrl.owner := me; } } ⟩
  while (prev);
}
```

LCI in depth – Verify Procedure

- Definition of a procedure: *Some actions chained together, where one can assume that no other thread will interfere with the current object(s).*

```
void incr ( Counter c) {
        < a := c.n; >
              // Someone could change c.n, so we get a ≤ c.n
        < if c.n = a then c.n := a + 2 end >
              // Now we know a < c.n
        < b := c.n; assert ( a < b ); >
}
```

- Okay for humans, but sadly the verifier has problems with this.

LCI in depth – Claims

- Definition of thread-local data: *A field o.f is thread-local iff the current thread can prevent any other thread from changing it.*
- Definition of a claim: *Claims are objects with invariant v. The stability of v implies a lemma.*

- Claims are always ghost, and only used for verification.
- The verification if incr() relied on a lemma that a ≤ c.n is preserved by legal actions – this needs a claim.

- Together, thread-local data and claims allow verification.

LCI in depth – Verify Procedure

```
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)) ∨
           inv(ctrl.owner))
}
```

LCI in depth – Verify Procedure

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**; } >

// The ghost command essentially sets up cl.

< **if** (c.n = a) **then** c.n := a + 2; **end ghost** { cl.floor := a + 1; } >

// The ghost command is legal. If c.n = a then a+2 is the new floor, otherwise cl's invariant holds with a ≤ c.n before the action. Thus, a+1 is a valid new lower bound.

< b := c.n; assert(a < b); >

}

Conclusion – what LCI achieved

- Formulated an admissibility condition for invariants, which permits the local checking of global invariants.
- Guide to transform common invariants to admissible invariants (using ghost state, etc.).
- The introduction of claims, which are often necessary to verify concurrent algorithms.

Verification of the Hyper-V sources

	# of checks	min	max	avg	median
admissibility	152	0.5s	50s	15s	13.6s
functions	367	0.4s	2581s	50.5s	12.8s

Conclusion – Problems remaining

- High overhead: LCI was incorporated into VCC, then used on Hyper-V. After 2 years, one third of the 100k lines were annotated.
  - Assuming only one person working on it, working 235 days a year, this translates to **72 LOC annotated a day**.
- Since LCI is only available as part of VCC, and VCC uses a different syntax, trying it out is somewhat more complicated.
- Link: http://rise4fun.com/vcc

Questions

Backup – Benefits of LCI vs. other Methodologies

- Since the methodology is modular, it allows verification of software pieces, and does not require complete annotation of the code.
- In this special case, verification only requires to check the used objects (which scales nicely, btw).
- The methodology is also very general, and thus flexible. It allows other methodologies to be built upon itself.
- For concurrency, the flexibility is even more essential, since it allows verification of fine-grained algorithms, that might not be possible (or very hard) in different methodologies.

Backup – Verify Admissibility (Requirements Recap)

- Invariants must be **reflexive**: If $< h_0 , h >$ satisfies the invariant, so must $< h, h >$
- Invariants must be **stable**: They can't be broken by **legal** actions.
- Invariants that are both **stable** and **reflexive** are called **admissible**.

Backup – Verify Admissibility Stability for Invariant of Low

forall h1, h2: **heap**, o: **Low** ::

(h2[o, floor] <= h2[h2[o, cnt], n] ||
// either a legal update of o

h2[h2[o, cnt], n] == h1[h2[o, cnt], n] ||

h2[h2[o, cnt], n] == h1[h2[o, cnt], n] + 2 ||
// or a legal update of o.cnt

(h1[o, cnt] == h2[o.cnt] && h2[h2[o, cnt], n] == h1[h2[o, cnt], n]) // or an update to another object

==> h2[o, floor] <= h2[h2[o, cnt], n]
// the invariant is preserved

Backup – Verify Admissibility Reflexivity for Invariant of Low:

forall h1, h2: **heap**, o: **Low** ::

h2[o, floor] <= h2[h2[o, cnt], n] **==>**
// if a transition to h2 is safe

h2[o, floor] <= h2[h2[o, cnt], n]
// then h2 is safe

(This one is trivial, cause the invariant is single-state, so h1 does not event appear.)