Local Verification of Global Invariants in Concurrent Programs

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

[1] European Microsoft Innovation Center, Aachen
{ernie.cohen,stephan.tobies}@microsoft.com
[2] Microsoft Research, Redmond
{michal.moskal,schulte}@microsoft.com

Abstract. We describe a practical method for reasoning about realistic concurrent programs. Our method allows global two-state invariants that restrict update of shared state. We provide simple, sufficient conditions for checking those global invariants modularly. The method has been implemented in VCC[1], an automatic, sound, modular verifier for concurrent C programs. VCC has been used to verify functional correctness of tens of thousands of lines of Microsoft's Hyper-V virtualization platform[2] and of SYSGO's embedded real-time operating system PikeOS.

1 Introduction

Verifying functional correctness of complex, low-level, shared memory, highly concurrent programs (e.g., operating system kernels) requires intricate global invariants. To scale to large systems, checking these invariants should be modular; in particular, invariance checking should obey program abstraction boundaries, so that e.g., low-level code can be checked without having to consider high-level invariants, and private low-level invariants can be changed without breaking high-level clients. However, the need for modularity conflicts with the practical need to have invariants that span multiple objects[3]; for example, high-level objects often have invariants that mention (public) fields of low-level objects.

One common approach to this problem is to impose on programs a structural discipline that syntactically restricts the form of object invariants. Such disciplines tend to work well for certain classes of programs but not for others, leading to a stream of extensions (e.g. the extension of the Spec# discipline [3,13] with friends [1], or freezing [16]). Another popular approach is based on separation logic [18] and its combination with rely-guarantee reasoning [11,19,10]. However, they too make methodological commitments that work well for certain classes of programs but less well for others (e.g., the choice to use fractional permissions

[1] VCC is available in source for academic use at `http://vcc.codeplex.com/`

[2] The Hypervisor verification is part of the Verisoft XT project supported by BMBF under grant 01IS07008.

[3] Objects mean collections of closely related data, e.g., regions of memory interpreted as structs in C.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 480–494, 2010.

rather than counting permissions [5]). Moreover, resource logics like separation logic have inherently higher complexity than ordinary state-based logics [6], and automation for separation logics (in their full generality) is far less developed than for conventional logics.

We propose a different approach that achieves modularity without going outside of ordinary logic: *locally checked invariants* (LCI). LCI assigns to each object (including threads) a two-state invariant, i.e., a predicate over pairs of states expected to hold for every pair of consecutive states in every execution.[4] Verification of a concurrent program reduces to checking that these invariants hold for every state update invoked by the program. Rather than syntactically restricting invariants, LCI imposes a *semantic* condition (i.e., a proof obligation) on the object invariants; this condition guarantees that *local* invariant checking (i.e., checking the invariants of objects actually changed in an update) suffices to prove preservation of all invariants. Because the restriction is semantic rather than syntactic, it allows for great deal of flexibility in structuring *global* invariants. In particular the kind of verification scaffolding that has to be built into other programming methodologies can be implemented syntactically in LCI at the program level, using ghost state and ordinary object invariants, allowing multiple methodologies to be used on a single program. As LCI is based on ordinary logic, it can be implemented on top of stock theorem provers.

LCI has been implemented in VCC [7], a verifier for concurrent C software that has verified the functional correctness of tens of thousands of lines of commercial concurrent C code. The use of LCI, rather than a specialized program logic, allowed VCC to be built on an established verification condition generator (Boogie [2]) and state-of-the-art theorem prover (Z3 [9]). This paper presents LCI on the example of a simple type-safe language, with a natural notion of objects. Applying LCI to C involved developing a type-safe, yet flexible, view of C memory, which is described separately [8]. Viewing states of execution, for which the invariants are evaluated, at the granularity of the hardware-atomic memory updates allowed us to verify fine-grained concurrent algorithms.

The main contributions of the paper are:

- the formulation of an *admissibility* condition for invariants, which permits the local checking of global invariants (Sect. 3),
- an encoding of common invariant disciplines as admissible invariants (Sect. 4),
- the introduction of *claims*, which are objects encapsulating stable, derived knowledge about system state, often necessary to verify concurrent algorithms (Sect. 5).

2 The Idea: Stable Invariants and Legal Actions

For simplicity of presentation, consider a state to be a heap mapping addresses to typed object values. By an *action*, we mean an ordered pair of states, representing

[4] States of execution refer to states possibly visible to other threads, e.g., in C every memory write yields a separate state. This allows for verification of fine-grained concurrent algorithms.

a transition from the first state (pre-state) to the second (post-state). Again for simplicity, we restrict consideration to actions that do not change the type of object stored at an address, i.e., the set of objects and their addresses are fixed (but this restriction is easily dropped).

Each object has a two-state invariant, determined by its type. Terms within object invariants wrapped with **old()** refer to the pre-state of a transition; terms not so wrapped refer to the post-state. An action is *safe* iff it satisfies the invariant of every object.

Here are two simple examples of object type definitions:

```
type Account {
  int val, creditLimit;
  inv(creditLimit ≤ val)
}
```

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

The invariants of these types can be read as follows: after a safe action, in every Account object, the value of the val field has to be greater or equal than the creditLimit field, and in every Counter object, the n field is unchanged or incremented by two. Note that any number of objects can be updated in a single action.

An object invariant can be interpreted on a single state by interpreting it over the *stuttering* action from that state, i.e., the action that goes from that state to itself. A state is *safe* if the stuttering action from that state is safe. We would like to be assured that actions always start from safe states, so we require program execution to start in a safe state, and require that all object invariants satisfy the following *reflexivity* property: if an action $\langle h_o, h \rangle$ satisfies the object invariant, then the stuttering action $\langle h, h \rangle$ also satisfies the invariant. If all object invariants are reflexive, then every safe action has a safe post-state. Invariants of both objects above are reflexive, whereas an invariant n > **old**(n) for the Counter, which requires an increment during each safe transition, would not be reflexive.

The invariants above refer only to fields of their object. If all invariants were like that, checking safety of an action (starting from a safe state) would require only checking the invariants of those objects updated by the action. But some invariants have to span multiple objects. Consider for example the following types, where % is the modulo operator and fields of object type are object references (like in Java or C#).

```
type ParityReading {
  Counter cnt;
  int parity;
  inv(cnt.n % 2 = parity)
}
```

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

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

An action that updates x.cnt.n, where the type of x is any of the types defined above, might break the invariant of x. However, if the action preserves the invariant of x.cnt (i.e., increments x.cnt.n by two), then it cannot break the invariant of an x of type ParityReading or Low, but it can break the invariant of an x of type High. We therefore reject the definition of the type High as *inadmissible*, whereas

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

Fig. 1. Definitions

it will allow the other type definitions. Note that the admissibility of a type can depend on the definitions of other types; e.g., ParityReading is only admissible because increments of Counter are always by two.

The essence of LCI is captured in the following (still informal) definitions. An action is *legal* iff it preserves the invariants of updated objects. A *stable* invariant is one that cannot be broken by legal actions (i.e., holds over legal actions). An *admissible* invariant is one that is stable and reflexive. It follows (by Theorem 1 below) that if all invariants are admissible, then every legal action from a safe pre-state is safe and has a safe post-state. Our methodology involves proving the admissibility of each invariant, and then proving that each action produced by the program is legal; this lets us conclude that all actions produced by the program are safe.

3 Formalization

We now formalize the insight from the last section (Fig. 1). Heaps $\mathbb{H}$ map integers (i.e., addresses) to objects, which are maps from field names to integers. The invariant function $\mathsf{inv}(h_o, h, p)$ returns true iff the action changing the state from h_o 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). E.g., the Counter's invariant $\mathsf{n} = \mathbf{old}(\mathsf{n}) \vee \mathsf{n} = \mathbf{old}(\mathsf{n}) + 2$ translates to:

$$\mathsf{inv}_{\mathsf{Counter}}(h_o, h, p) \equiv h[p][\mathsf{n}] = h_o[p][\mathsf{n}] \vee h[p][\mathsf{n}] = h_o[p][\mathsf{n}] + 2$$

Our goal is to conclude the safety of an action while only checking its legality.

We first note that if all invariants are reflexive, then safe actions result in safe states:

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

Stability is defined for type τ by $\mathsf{stable}(\tau)$ (in Fig. 1). Given a legal action ($\mathsf{legal}(h_o, h)$) that starts from a safe state ($\mathsf{safe}_1(h_o)$), we want to be able to

conclude that the action is safe, and thus (by Lemma 1) that the post-state is safe. By syntactic transformations of the definition of stable we get:

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

We call an invariant of type τ admissible ($\mathsf{adm}(\tau)$) if its reflexive and stable. This takes us to our main soundness theorem:

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

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

Proof. By combination of Lemmas 1 and 2. □

Thus, any sequence of *legal* actions starting from a safe state is a sequence of *safe* actions. Therefore, the verification system can generate proof obligations for legality, reflexivity and stability and, by Theorem 1, deduce global correctness. Reflexivity and stability depend only on invariants of data referenced from the current invariant; legality depends only on invariants of objects that are updated. As a consequence all these conditions can be checked modularly.

The proof of Theorem 1 is trivial given the carefully chosen definition of admissibility. Yet, in the following, we see that this notion is not overly restrictive and that many interesting invariants can naturally be made admissible.

Dependents and Fix-Points

We have mentioned that the invariant of High is inadmissible. However, if we actually wanted the Counter to have an external object restricting its changes, we could have done so by explicitly referring to the invariant of the High bound from the invariant of the Counter (multiple **inv**(...) clauses are syntactic sugar for their conjunction).

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

```
type High2 {
  Counter2 c;
  int ceiling;

  inv(c.b = this)
  inv(c.n ≤ ceiling)
}
```

The invariant of High2 is now admissible thanks to the back-reference b in the invariant of the Counter2. Any legal action touching n in a counter has to preserve invariant of its attached object b. The first-order translation of the Counter2's invariant that includes the reference to b's invariant is:

$$\begin{aligned}\mathsf{inv}_{\mathsf{Counter2}}(h_o, h, p) \ \equiv\ & (h[p][\mathsf{n}] = h_o[p][\mathsf{n}] \vee h[p][\mathsf{n}] = h_o[p][\mathsf{n}] + 2) \wedge \\ & h[p][\mathsf{b}] = h_o[p][\mathsf{b}] \wedge (h[p][\mathsf{n}] = h_o[p][\mathsf{n}] \vee \mathsf{inv}(h_o, h, h[p][\mathsf{b}]))\end{aligned}$$

The definition of $\mathsf{inv}(\dots)$ has become recursive. We thus define the function $\mathsf{inv}(\dots)$ as any fix-point solution of the set of type-invariant-derived equations

like the one above. To ensure existence of a fix-point solution, we restrict the use of **inv**(...) in invariants to positive polarities.[5]

4 Structuring Invariants

LCI can be viewed as a "low-level" verification mechanism: admissibility, being a semantic check, allows for making arbitrary tradeoffs as to what to check where. In particular, any invariant can be made admissible by referencing it from the invariants of objects that it depends on (as in the High2 example). This flexibility allows for encoding more sophisticated methodologies on top of LCI at the syntactic level, without jeopardizing soundness. Indeed, LCI makes it easy to soundly add (and mix) different formal methodologies within a single program.

Central to the encoding of methodologies on top of LCI is the use of *ghost state* — state added to the program to aid reasoning. An implementation should check that ghost code (code that references ghost state) does not update non-ghost state and is total (terminating); these conditions guarantee that the ghost code and state can be erased without effecting behavior of the program on non-ghost state. Ghost code is not only used to encoding methodologies; it also provides a critical tool for program annotation, allowing the programmer to communicate to the verifier important intuitions as to why the program works. For example, ghost objects can be used to encode rights and knowledge, and ghost code can be used to create, destroy, and transfer these objects during program execution.

Here, we describe how some of the methodologies used in VCC are encoded syntactically on top of LCI (summarized in Fig. 2). We apply a syntactic transformation to every user-defined type, which adds additional fields and invariants, and which also weakens the user-defined invariant ψ. For every instance of an user-defined type, we also add an instance of the helper OwnerCtrl type. We first describe disabling invariants (Sect. 4.1) and ownership trees (Sect. 4.2); these provide the Concurrent Spec# [13] ownership system. We then describe handles (Sect. 4.3), a generalized form of read permissions [5].

4.1 Validity

Theorem 1 assumes that the initial state is safe. However, most languages allow for dynamic creation (and perhaps disposal) of objects, and object's invariants usually do not hold prior to completion of initialization nor after the start of destruction. Additionally, when objects are only accessed sequentially it is convenient to temporarily disable the invariant, perform several updates on an object, and re-enable its invariant. For those reasons we introduce a valid field, defined

[5] Let L be the power set lattice of $\mathbb{H} \times \mathbb{H} \times \mathbb{Z}$, let $f : L \to L$ be a function such that $\langle h_o, h, p\rangle \in f(I)$ if and only if $\mathsf{inv}_{\mathsf{type}(p)}(h_o, h, p)$ with inv replaced with characteristic function of I. Because $\mathsf{inv}(\dots)$ only occurs positively in $\mathsf{inv}_{\mathsf{type}(p)}(\dots)$ f is monotonic, and therefore by Knaster-Tarski theorem f has a fix-point, which we use as inv. An unconditional cycle in inv definitions might yield an interpretation of **false**, which is sound but will prevent successful verification.

```
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 }
```

```
17 ghost type OwnerCtrl {
18   object owner, subject;
19   inv(unchg(subject))
20   inv(unchg(owner) ∨ inv(owner))
21   inv(unchg(owner) ∨ inv(old(owner)))
22   inv(unchg(subject.valid) ∨ inv(owner))
23   inv(type(owner) = Thread ∨ subject.valid)
24   inv(subject.ctrl = this)
25   // Handles, Sect. 4.3
26   set<Handle> handles;
27   inv(unchg(handles) ∨ inv(owner))
28   inv(∀(Handle h;
29         h ∈ old(handles) ∧ h ∉ handles
30           ⇒ ¬h.valid))
31   inv(handles = {} ∨ subject.valid)
32 }
```

Fig. 2. Every user (i.e., not OwnerCtrl) type τ with fields $\mathcal{F}$ is transformed by adding implicit ghost fields and invariants. The user invariant, ψ, is weakened. Additionally an ownership control type is added (but not transformed). **unchg**(f) is short for **old**(f)= f.

for every object type. Line 6 of Fig. 2 says that user invariants have to hold only when the object is valid in the pre- and/or the post-state. In particular, single-state user-invariants only have to hold when the object is valid. We define the valid field to be **false** in the initial state, so the initial state is safe. An object is typically made valid at the end of initialization and invalid before its destruction.

4.2 Ownership

How does the introduction of validity effect the admissibility of invariants? Consider, for example, the invariant of an object low of type Low. To make use of the "counter only goes up" property of low.cnt, low now needs to know that low.cnt is valid. In a language with explicit destruction, this can be rephrased as: what prevents somebody from destroying low.cnt? One possible solution is to designate low as the *owner* of low.cnt; informally, we can think of this meaning that low.cnt is "part of" low.

Formally, ownership is expressed by adding a ghost ctrl field to every object and making it point to an instance of ownership control type (again Fig. 2). Now the field x.ctrl.owner points to the object that owns x. This encoding allows the ownership relation to dynamically change during execution, but guarantees that each object always has a unique owner.

The invariants governing OwnerCtrl say that it always refers to the same subject (line 19), ownership transfer is allowed, but requires a check of the invariants of the old and new owner (line 20 and line 21), and only threads can own invalid objects (line 23; thread invariants will be discussed in Sect. 5.1). Note that the use of a separate ownership control object means that the invariants of the object itself do not have to be checked when its owner changes. Note also that these

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

```
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);
}
```

Fig. 3. A spin-lock. The semantics of actions (Acquire) is explained in Sect. 5.

invariants allow ownership cycles, but in practice the ownership relation defines a forest, except that the tree roots (threads) own themselves.

We could now make Low own and thus control validity of its Counter by adding the user invariant cnt.ctrl.owner = **this**, which entails cnt.valid. Such an invariant is admissible: the additional invariant on Counter from line 10 ensures that any legal action leaves cnt.ctrl unchanged, and an attempt to update cnt.ctrl.owner needs to conform with both the old and new owner's invariants (see lines 20 and 21). Thus, Low cannot loose control of its Counter.

Conversely, an invariant of the form **type**(ctrl.owner)= Low in Counter would not be admissible: the control type enforces the invariant of owners, not subjects, so changing the owner of a subject (which only involves changing the owner field in the OwnerCtrl type and not the subject itself) could break the invariant. This is intentional: synchronization mechanisms, like *spin-locks*, should be polymorphic in the type of the data that they protect, but the implementation of these mechanisms needs to manipulate ownership of the protected objects. For example, acquiring a lock transfers ownership of the lock-protected resource (rsc in Fig. 3) to the calling thread. Similarly, generic containers (e.g. lists, stacks) should be polymorphic in the type of the objects they hold.

4.3 Handles

Ownership does not provide for sharing; at most one thread can (transitively) own an object at any time, and only that thread can deduce (through ownership) that the object is valid. But sharing is often necessary. For example, the whole point of a spin-lock is that multiple threads can try to acquire it simultaneously; all of these threads must know that the lock is valid.

One way to provide sharing is with *handles*. Consider the following type (on which we apply the transformation from Fig. 2):

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

The invariant of Handle is admissible, because the invariants of OwnerCtrl prevent removal of a valid handle from the handles set (line 30) and prevent the object from being invalidated as long as there are outstanding handles (line 31). Multiple clients can each own valid handles on the shared object, and each of these handles guarantees validity of the shared object.

When multiple clients rely on an object, the object's owner typically keeps track of the clients, e.g., by maintaining a reference counter. An example of such scenario is a *reader-writer lock*. Acquiring a read lock returns to the caller a handle on the resource. Acquiring a writer lock waits for the reader count (correlated in the invariant with the cardinality of the handle set) to drop to zero and transfers the ownership of the resource from the lock to the acquiring thread. This thread can then invalidate the resource and update its fields.

A handle allows the fields of obj to be changed (subject to checking obj's invariant); the validity of obj is needed in the case that obj is owned by another thread. This allows threads to race in a controlled manner, for example by trying to acquire a lock. A handle on obj can also be viewed as a read permission on the fields f of obj for which obj.valid $\wedge$ **inv**(obj)$\Rightarrow$ **unchg**(obj.f): as long as the handle exists, these fields cannot change. This is a counting, rather than fractional, permission[6]; but unlike permissions in separation logic, handles are first-class objects, so one can create handles on handles with similar results to splitting fractional permissions (cf. [5]).

As a more concrete example consider the following modification of the type Low:

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

Adding the handle cntH on cnt makes the invariant of the Low2 admissible, while being able to share Counter objects between clients. A similar modification needs to be made to ParityReading and High2.

5 Verifying Actions

LCI treats procedures as sequences of atomic actions performed by a single thread (denoted by **me**), with possible interference of safe actions of other threads in between. To verify a procedure, one needs to check legality of all its actions, assuming only safety of interfering actions (that is without looking at the code of other procedures). For example, VCC achieves that by translating the sequence of actions into a sequential Boogie program, and verifying it using weakest preconditions calculus [2]. After translation of each action, we assert its legality

[6] Fractional permissions can also be implemented on top of LCI, but we have not found them necessary.

and assign an arbitrary value to the entire heap. We assume this new heap to represent a safe state (per Theorem 1) and that it was constructed from the old one by zero or more safe actions of other threads (which has consequences to thread-local data, see below). As an example, consider the following code, using $\langle \ldots \rangle$ to denote atomic actions:

```
1  void incr(Counter c) {
2    ⟨ a := c.n; ⟩
3    ⟨ if (c.n = a) c.n := a + 2; ⟩
4    ⟨ b := c.n; assert(a < b); ⟩
5  }
```

We first read the value of the counter into a local variable a. Then, using an atomic compare-and-swap operation (in-lined here for clarity), we increment the value of the counter, provided it did not change in between. Finally, we read the new value of the counter into b and want to statically prove $a < b$. This should be provable because the invariant of Counter ensures that the counter can only grow, and so either the compare-and-swap succeeds in storing the incremented value, or it fails because somebody else incremented it between lines 2 and 3.

We verify incr as follows. Suppose for a moment that we know c remains valid throughout execution of incr. We check the legality of the first action; because it modifies only local variables, this is trivial. Afterwards, we simulate arbitrary, but safe, interference by other actions by assigning an arbitrary safe value to the heap. We then check legality of the second action performed on that new heap and so on. Local variables, remain unchanged between actions of the current thread. Because locals do not change, and c.n can only grow, from line 2, we have $a \leq c.n$ and after line 3 even $a < c.n$. Thus, the assertion does indeed hold.

This kind of reasoning makes use of the fact that any property ψ of data on the heap is preserved, provided that it is maintained by safe actions of other threads, i.e., $\forall h_o, h.\ \mathsf{safe}(h_o, h) \wedge h_o[\mathbf{me}][\mathsf{state}] = h[\mathbf{me}][\mathsf{state}] \wedge \psi(h_o) \Rightarrow \psi(h)$, where $h[\mathbf{me}][\mathsf{state}]$ is used to encode the local state of the current thread. However, we cannot expect all such ψ's to be automatically inferred by the theorem prover as needed. In the following, we deal with situations where the inference of a suitable ψ is beyond the capabilities of the theorem prover. Preservation of some common properties, like behavior of thread-local data, can be proven as lemmas once and for all (see Sect. 5.1) and then be built into the verifier. Other properties need to be taken care of by the user (Sect. 5.2), using LCI's existing machinery.

5.1 Thread-Local Data

We now come back to the fact that we need validity of a Counter c throughout the execution of incr(c): if some other thread would invalidate c while incr(c) is executing, c.n could potentially decrease and the assertion from line 4 can no longer be proven. Simply making the invariant of some other object o guarantee c.valid, does not help, as it would only defer the validity of c to the validity of o. The natural place to start validity deduction is **me**, i.e., the Thread object representing the current thread of execution.

The field o.f is *thread-local* data of thread t iff the invariant of t can admissibly prevent any other threads from changing it, e.g., by including a formula like **unchg**(**this**.state) $\wedge\ \phi \Rightarrow$ **unchg**(o.f), where ϕ encodes additional conditions, like o.ctrl.owner = **this**. The thread invariants are defined to be conjunctions of all such admissible formulas. Thus, they prevent any interference they possibly can. We approximate that by assuming **inv**(**me**) to be true and leaving other thread invariants unspecified when verifying actions. Hence it is sound to assume that thread-local data does not change between actions of the current thread.

In particular an object invariant **unchg**(f) $\vee$ **inv**(ctrl.owner) makes the field f local data of the owning thread (provided that ctrl.owner is a thread). Thus the valid and owner fields are both local to the owning thread. Additionally, per line 15 of Fig. 2, all fields of invalid objects are local to the owning thread, too.

As a consequence, if **me** owned c, that would be enough to verify incr(c): the fields c.ctrl.owner and c.valid would be thread-local. But clearly, incr(c) is designed so that multiple threads can execute it concurrently on the same c. So instead we reuse the Handle type: **me** owns a (initially valid) handle h, therefore h.ctrl.owner, h.valid, and h.obj are thread-local, and **inv**(h) implies c.valid.

5.2 Claims

Verification of incr(), in addition to a handle, relied on a lemma that the assertion a $\leq$ c.n is preserved by legal updates. To persist such a property ψ between LCI actions one can use a *claim* — an object with invariant ψ. The stability of the invariant (checked by an LCI verifier) implies the lemma. In case of incr() the existing type Low2 can be used as a claim. A claim is ghost, as it is only a verification device, and so we can insert updates of its multiple fields in the middle of the physical atomic actions. In the code below, we pass a pre-allocated, thread-local Low2 claim object and, for simplicity, ignore its creation. We also add an invariant making fields of Low2 thread-local.

```
type Low2 { // ...
  inv((unchg(floor) ∧ unchg(cntH) ∧ unchg(cnt)) ∨ 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); ⟩
}
```

After the first read, we initialize the Low2 claim cl with a as a lower bound for c according to its invariant. The next action first tries to increment the counter, and regardless of the result increments the floor field of the claim. This is a legal action: if we incremented the counter, then a + 2 is a new lower bound. If we did

not, $\mathsf{c.n} \neq \mathsf{a}$ must hold and Low2's invariant (which holds just before the action) additionally entails $\mathsf{c.n} \geq \mathsf{a}$, and thus $\mathsf{a} + 1$ is a lower bound for the counter. Correctness of last action simply follows from cl's invariant as we have made fields of the claim thread-local by requiring changes to satisfy **inv(me)**. Thus, a claim is used to derive a inductive property of an invariant.

Claims are often needed when verifying programs using LCI. For this reason, VCC provides a syntax for creating a new claim type and instance, along with the required handles, in a single statement. The claim object also captures the values of locals at the point where it is created. Using this syntactic sugar, the first action in our program could be written without introduction of the type Low2 as:

$\langle\ \mathsf{a} := \mathsf{c.n};\ \textbf{ghost}\ \{\ \mathsf{s} := \textbf{claim}(\mathsf{h},\ \mathsf{c.n} \geq \mathsf{a})\ \}\ \rangle$

VCC claims are still first-class objects, so it is possible to own them, pass them around, and store them in other objects. In particular, it allows for proving a lemma in one context and using it in another.

6 Evaluation and Related Work

The LCI verification methodology was developed for and along with the VCC verification tool. Our goal for VCC was the sound verification of functional correctness of industrial-strength concurrent C code, driven by program-level annotations on the code base. Because we were after a scalable, industrial verification process driven by software engineers (rather than verification engineers), we avoided interactive proof checking and logics that are difficult to automate efficiently (e.g., higher-order logics and separation logics). Ghost data and code was used to workaround limitations of first-order logic, e.g., the reachability relation was expressed with a ghost field holding the set of reachable nodes, which needed to be updated explicitly, using ghost code. We believe that developers are well equipped to write substantial amounts of code manipulating ghost state (to manipulate ownership, claims, as well as abstractions like the reachability above), and we know that such techniques scale to complex problems.

The driving application for VCC development was the verification of the Microsoft Hyper-V hypervisor. The hypervisor (about 100K lines of C code) sits directly on multi-processor x64 hardware, and provides a number of virtual multi-processor x64 machines (with some additional instructions). The hypervisor is highly optimized for multi-core hardware, so in addition to typical OS components (e.g., scheduler and memory allocator) it contains a number of custom concurrency control mechanisms and algorithms, mostly using fine-grained concurrency control. For the last two years, the Hyper-V verification project has focused on annotating the existing code base with invariants and function contracts and checking these annotations, to prove that the Hyper-V simulates (a model of) the actual x64 hardware. So far, about 1/3 of the code base has been annotated. VCC is also used to verify PikeOS, an embedded real-time OS [4].

LCI has proven to be powerful enough to express all required specifications, ranging from low-level details, like concurrency primitives, to higher-level abstractions like virtual machine partitions. LCI was relatively easy to implement in VCC; however specifications need to be carefully formulated to stay off of unfruitful, non-terminating, or very time-consuming proof paths. Still, it was possible to massage the specifications to the point where the Z3 theorem prover was accepting them with certain robustness (i.e., small changes in specifications would not make it fail).

The expressivity of LCI is beyond that of other verification tools and there are no commonly accepted benchmarks in this area, so we cannot easily compare run-times. To give you an idea of the performance that we achieve, here are the times consumed by the verification of the Hyper-V sources on a single 2.33 GHz Intel Xeon core:

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

Actual turnaround times on our 8-core machine (multiple admissibility checks and function verifications can be parallelized trivially) is well below 2 hours. For most problems, a memory limit of 200MB suffices, while a handful of problems require more memory but never exceeding 1GB.

The typical VCC work flow starts by running VCC on the initial version of the code and specification. In case of a verification error either the specification or the code needs to be fixed, the decision is up to the VCC's user, who is aided by a model viewer, showing the counter-example found by the theorem-prover. The user then runs VCC again and the process repeats, typically multiple times. From this point of view the most important performance characteristic is the time it takes to verify a single function.

Many of the ideas of VCC are rooted in the Spec# project [3]. However, unlike Spec#, VCC is based on a tiny core, namely LCI, allowing users to extend the methodology at will. In fact, Spec#'s sequential methodology, comprising reps, peers, visibility, and observers, can be expressed in LCI. Also our thread-ownership discipline follows Concurrent Spec# [13]. The difference is that in LCI one can also verify lock *implementations* (see Fig. 3 and [12]), instead of treating locks as primitives. Finally, VCC also adopted Spec#'s framing mechanism: procedures list the objects' ownership domains that they are allowed to write.

An alternative promising approach to addressing the modular verification problem of concurrent programs is taken by the separation logic [18] community. Concurrent separation logic [17] divides the heap into a thread-local and a shared part. The shared part is governed by a single-state invariant. Extensions like SAGL [11] and RGSep [19] use two-state rely/guarantee predicates for the shared part. Finally, LRG [10] introduces the separating conjunction over two-state predicates. All these approaches require specialized resource logics, whereas our goal has been to stick to first-order logic to be able to leverage existing high-performance automated theorem provers.

The Chalice program verifier [15] also uses a resource logic, but it reasons about permissions at the syntactic level, and pushes the remaining proof obligations to the theorem prover. While restricted at the permission level, the specifications can be stronger in their classical parts.

A current disadvantage of LCI is its high annotation overhead: Spec#, some separation logic tools, and Chalice all have much lower overhead in cases they are expressive enough. Part of this is intentional; we intended to provide a powerful "low-level" program verifier, and to later introduce syntactic sugar as we learned what kinds of abstractions and abbreviations were most effective. In VCC, which works at slightly higher level of abstraction than the bare-bone LCI presented in this paper, we have found it to be in the order of one line of annotation per line of code. As a comparison, the impressive seL4 verification project [14], where similar properties were verified (but in a much smaller code base of sequential code), is reporting 2:1 overhead for specifications and 10:1 overhead for the proofs (in LCI the proofs are automatic, modulo the aforementioned massaging).

7 Conclusion

LCI is a modular verification methodology for concurrent programs. It introduces semantic admissibility condition on two-state invariants, which guarantees that updates can be checked locally. LCI has been proven to scale in practice and be expressive enough for industrial program verification.

Acknowledgements. Thanks to VCC users and people who provided the infrastructure: Artem Alekhin, Eyad Alkassar, Mike Barnett, Nikolaj Bjørner, Holger Blasum, Sebastian Bogan, Sascha Böhme, Matko Botinĉan, Vladimir Boyarinov, Markus Dahlweid, Ulan Degenbaev, Lieven Desmet, Sebastian Fillinger, Mark A. Hillebrand, Tom In der Rieden, Bruno Langenstein, Dirk Leinenbach, K. Rustan M. Leino, Wolfgang Manousek, Stefan Maus, Leonardo de Moura, Andreas Nonnengart, Steven Obua, Wolfgang Paul, Hristo Pentchev, Elena Petrova, Thomas Santen, Norbert Schirmer, Sabine Schmaltz, Peter-Michael Seidel, Andrey Shadrin, Alexandra Tsyban, Sergey Tverdyshev, Herman Venter, and Burkhart Wolff.

References

1. Barnett, M., Naumann, D.A.: Friends need a bit more: Maintaining invariants over shared state. In: Kozenand, D., Shankland, C. (eds.) MPC 2004. LNCS, vol. 3125, pp. 54–84. Springer, Heidelberg (2004)
2. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
3. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology 3(6), 27–56 (2004)

4. Baumann, C., Beckert, B., Blasum, H., Bormer, T.: Better avionics software reliability by code verification – A glance at code verification methodology in the Verisoft XT project. In: Embedded World 2009 Conference, Nuremberg, Germany, March 2009. Franzis Verlag (to appear, 2009)
5. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. SIGPLAN Not. 40(1), 259–270 (2005)
6. Calcagno, C., Yang, H., O'Hearn, P.W.: Computability and complexity results for a spatial assertion language for data structures. In: APLAS, pp. 289–300 (2001)
7. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Urban, C. (ed.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009) (invited paper)
8. Cohen, E., Moskal, M., Schulte, W., Tobies, S.: A precise yet efficient memory model for C. In: Workshop on Systems Software Verification (SSV 2009). Electr. Notes Theor. Comput. Sci., vol. 254, pp. 85–103. Elsevier Science B.V., Amsterdam (2009)
9. de Moura, L., Bjorner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
10. Feng, X.: Local rely-guarantee reasoning. In: Shao, Z., Pierce, B.C. (eds.) POPL, pp. 315–327. ACM, New York (2009)
11. Feng, X., Ferreira, R., Shao, Z.: On the relationship between concurrent separation logic and assume-guarantee reasoning. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 173–188. Springer, Heidelberg (2007)
12. Hillebrand, M.A., Leinenbach, D.C.: Formal verification of a reader-writer lock implementation in C. In: Workshop on Systems Software Verification (SSV 2009). Electr. Notes Theor. Comput. Sci., vol. 254, pp. 123–141. Elsevier Science B.V., Amsterdam (2009)
13. Jacobs, B., Smans, J., Piessens, F., Schulte, W.: A simple sequential reasoning approach for sound modular verification of mainstream multithreaded programs. Electr. Notes Theor. Comput. Sci. 174(9), 23–47 (2007)
14. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: POPL, pp. 207–220. ACM, New York (2009)
15. Leino, K.R.M., Müller, P.: A basis for verifying multi-threaded programs. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 378–393. Springer, Heidelberg (2009)
16. Leino, K.R.M., Müller, P., Wallenburg, A.: Flexible immutability with frozen objects. In: Shankar, N., Woodcock, J. (eds.) VSTTE 2008. LNCS, vol. 5295, pp. 192–208. Springer, Heidelberg (2008)
17. O'Hearn, P.W.: Resources, concurrency, and local reasoning. Theor. Comput. Sci. 375(1-3), 271–307 (2007)
18. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS, pp. 55–74. IEEE Computer Society, Los Alamitos (2002)
19. Vafeiadis, V., Parkinson, M. J.: A marriage of rely/Guarantee and separation logic. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 256–271. Springer, Heidelberg (2007)