Notes for Lecture 19

Automated and modular refinement reasoning

Per-procedure simulation, non-interference via invariants.
Preemptive vs cooperative semantics.
Linear variables.
Variable hiding.

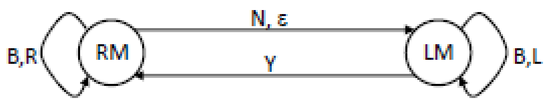
The CIVL verifier manipulates multiple operational descriptions of a program, i.e., several layers of refinement are specified and verified at once. What does this mean?

A simulation relation between a program and its abstraction is inferred from checks on each procedure, thus **decomposing** a whole-program refinement problem into per-procedure verification obligations.

For example, the mover type of AcquireLock is right.

These mover types (**R**ight, **L**eft, **B**oth and **N**on-mover) are checked by constructing verification conditions from each pair of atomic actions.

A yield sufficiency automaton (Figure 2) encodes all sequences of atomic actions (of Right, Left, Both and Non-mover types) and yields for which safety of cooperative semantics is sufficient for safety of preemptive semantics.



Each "transaction" starts with a sequence of right movers (or both movers) and ends with a sequence of left movers (or both movers). In the middle, it can have at most one non mover. Transactions must be separated by yield statements.
Commutativity reasoning may be avoided by annotating atomic action specifications with the mover type atomic and inserting a statement before every invocation of an atomic action. This can make proofs difficult.

The CIVL type system ensures that values contained in linear variables cannot be duplicated.

Each atomic action has a single-state gate predicate and a two-state transition relation. If a … gate predicate does not hold, the program fails. Compare with "fail" in Spec.

The *yield* invariant is expected to hold at yield statement (sequential correctness) and be preserved by concurrent threads (non-interference). So it's a global invariant: pc = yield → invariant.

The CIVL type checker enforces a generalization of the distinctness invariant that the permission sets corresponding to the values in available variables across all threads are mutually disjoint. ???

We can guarantee that Prog_{lo} is safe (i.e., all atomic actions will satisfy their gates when run) if the following conditions hold:

1. Prog_{hi} is safe when executed with preemptive semantics.

2. Prog_{lo} is a valid refinement of Prog_{hi}, according to the rules for refinement in CIVL. Specifically, for any atomic action A in Prog_{hi} implemented by a procedure P in Prog_{lo}, any path from entry to exit of P must contain exactly one atomic block that implements the action A; all other atomic blocks on the path must leave the global and thread-local state unchanged. Furthermore, all calls to A in Prog_{hi} are replaced by calls to P in Prog_{lo}.

3. The invariants of Prog_{lo} satisfy sequential correctness and non-interference with respect to cooperative semantics.

4. Prog_{lo} is well-typed with respect to linearity: does not duplicate any linear variables, and linear variables passed to procedure calls and atomic actions are available.

5. The atomic actions in Prog_{lo} satisfy the pairwise commutativity checks.

6. The yield statements in Prog_{lo} are sufficient, according to the yield sufficiency automaton in Figure 2.

7. Any infinite execution of Prog_{lo} must visit a yield statement infinitely often.

By themselves, conditions 1-4 guarantee that Prog_{lo} will be safe when executed with cooperative semantics. Conditions 5-7 then additionally ensure that Prog_{lo} will be safe when executed with preemptive semantics.

A key challenge for modular verification in CIVL is the checking of non-interference and commutativity. Each module M owns a set of global variables, each owned by exactly one module. Only M's procedures and actions can access M's global variables. Invariants can access anything. Note that ownership can change across refinement layers.

To perform this check, the CIVL verifier introduces the following fresh local variables in P: (1) a Boolean variable b initialized to false to track whether an atomic block along the current execution has modified a global or thread-local variable, (2) variables to capture snapshot of global and thread-local variables at the beginning of each atomic block. By updating these auxiliary variables appropriately, the refinement check is reduced to a collection of assertions introduced into the body of P at the end of atomic blocks and at the exit of P. ???

All the annotations, except those at yields, loops, and procedure boundaries, are automatically generated using the technique of verification conditions [5].

In initialization and root scanning, the mutator threads temporarily donate a fraction of their linear permissions to the GC thread.

The specification states that `Allocate` atomically adds new objects to the heap, while `ReadField` and `WriteField` read and write heap object fields. Although the GC's Mark and Sweep code constitutes most of the GC code, they are hidden in the high-level specification.

CIVL supports large proof steps, in each of which the bodies of several procedures are automatically replaced by atomic actions, thereby lowering the cost of both interaction and automation.

To verify a concurrent, shared-memory program using such tools as TLA+, one must encode the program semantics as a state-transition system and express verification goals in terms of this system. For concurrent, shared-memory software, CIVL enables reasoning on the structured, imperative multithreaded program text rather than a logic description of the program's state-transition relation.

Main points

Note that the programs being verified are small—17 to 539 loc. But GC is 2100 lines.

This isn't C, it's a "a core concurrent programming language". CIVL is designed for verification. What's the gap? It's a conservative extension of the Boogie [4] language: new language primitives for linear variables, asynchronous and parallel procedure calls, yields, atomic actions as procedure specifications, expressing refinement layers, and hiding of global variables and procedures.

Compare $Prog = (ps;\ as;\ G; \vec{T})$ (procs, atomics, Global vars, Threads) with the state in non-atomic Spec. $T = (TL;\ \vec{F});\ F = \ (P;\ L;\ s)$. T is Thread, P is Procedure, F is Frame, L is for Local.

"Refinement" here is replacing atomic actions with procs, perhaps several at once, thus making atomicity more fine-grained. It's a partial function RS from procedures to atomic actions; Prog^{hi} is obtained from Prog^{lo}. We prove safe $\text{Prog}^{\text{hi}} \rightarrow$ safe Prog^{lo}.

Pieces of the proof:

- "refinement" = simulation proofs—some sequential code simulates one step of spec. One step of code does the step of spec, the others are nops. Includes variable hiding.
- "location invariants", **non-interference** checked pairwise—each invariant vs. each extant atomic action (plus established by running thread). Note: these can refer to variables in other threads.
- "atomic specs" with mover types for procedures (which are the only atomic brackets in CIVL). This avoids the need for the labels we had in Spec.
- "linear types" to rule out aliasing—"values contained in linear variables cannot be duplicated". Used for thread-private variables, separation of memory, permissions. What is this in logic?
- "mover types" to make bigger steps ("transactions": RB* N LB*) between yields atomic without specific reasoning (**cooperative** vs. **preemptive** semantics). **Commutativity** is also checked pairwise—each procedure vs. each atomic action. Logic is only on cooperative program.
- "yield sufficiency automaton", a specialized simulation proof.
- Modularity—per-procedure reasoning for correctness, per-module reasoning for interference—modules "own" variables.

I didn't understand the bit about atomic blocks.

Explain about "rely-guarantee" reasoning. This might be Abadi and Lamport's composition.
Note the varying levels of formality—the Henzinger paper has a very different flavor.

Types vs logic??

Rely-guarantee

R,G ⊢ {P} C {Q}

IF:

(1) the initial state satisfies P, and
(2) every state change by another thread is in R,

THEN:

(1) every final state satisfies Q, and
(2) every state change by C is in G

Abadi-Lamport

Consider a system Π that is the composition of systems $\Pi_1, \ldots, \Pi_n$. We must prove that Π guarantees a property *M* under an environment assumption *E*, assuming that each Π_i satisfies a property M_i under an environment assumption E_i.

Observe that:

(1) We expect Π to guarantee M only because of the properties guaranteed by its components. Therefore, we must be able to infer that Π_i guarantees M from the assumption that each Π_i guarantees M_i.

(2) The component Π_i guarantees M_i only under the assumption that its environment satisfies E_i; and Π_i's environment consists of Π's environment together with all the other components Π_j. We must therefore be able to infer E_i from the environment assumption *E* and the component guarantees M_j.

These observations lead to the following principle.

Composition Principle: Let Π be the composition of $\Pi_1, \ldots, \Pi_n$, and let the following conditions hold.

(1) Π guarantees *M* if each component Π_i guarantees M_i.

(2) The environment assumption E_i of each component Π_i is satisfied if the environment of Π satisfies E and every Π_j satisfies M_j.

(3) Every component Π_i guarantees M_i under environment assumption E_i.

Then Π guarantees *M* under environment assumption *E*.

Types for Atomicity: Static Checking and Inference for Java

Flanagan, Freund, Lifshin, and Qadeer

Expressing our atomicity analysis as a type system in this way offers several key benefits. Type checking is modular and more scalable to large programs than model-checking or whole-program analyses. Atomicity specifications also serve as useful and verifiable documentation of a program's synchronization requirements. Moreover, the type system can be extended to uniformly handle additional locking idioms, such as locks protecting other locks, as we illustrate below.

Tech report MSR-TR-2015-8, reference 23: https://www.microsoft.com/en-us/research/publication/automated-and-modular-refinement-reasoning-for-concurrent-programs/