Interactive symbolic execution of concurrent programs in a theorem prover

EPFL

Alexandre Pinazza, EPFL, Switzerland
Supervised by Clément Pit-Claudel and Thomas Bourgeat

Systems and Formalisms Laboratory

VCA

Why is the following concurrent system correct?

User program

```
while (true) {
  int n = *counter_ptr;
  while (*flag_ptr ≠ 0) {}
  // Begining of exclusive access
  *data_ptr = n;
  // End of exclusive access
  *flag_ptr = 1;
  *counter_ptr = n + 1;
}
```

Model of Network Card

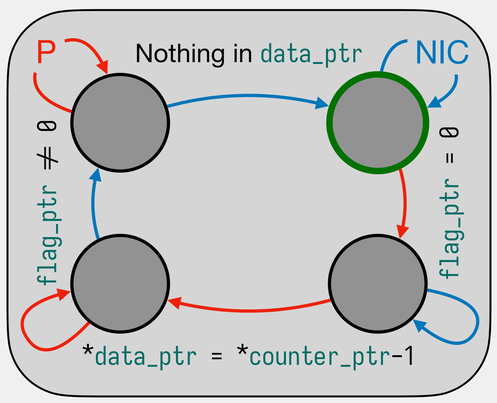
```
while (true) {
  while (*flag_ptr == 0) {}
  // Begining of exclusive access
  int n = *data_ptr;
  // End of exclusive access
  *flag_ptr = 0;
  EMIT(n);
}
```

Initial memory state: `*counter_ptr = 0`, `*flag_ptr = 0`

Spec: The network card (NIC) `EMIT`s the numbers 0,1,2,…

Typical paper proof:

"It is obvious that the two programs *only* race on `flag_ptr`. So, the system can be reduced to the following:"



Naïve Machine Check Proof:

We consider all interleavings. The resulting invariant of the system is the following:

Observation Machine checked concurrency proofs are more complex than paper ones.

Dynamic-ownership annotations enable simple concurrent-program proofs through optimistic symbolic execution

Dynamic-ownership annotations

Our paper proof relies on the insight that all accesses to `data_ptr` and `counter_ptr` are race-free.

```
TAKE(data_ptr);
int n = *data_ptr;
UNTAKE(data_ptr);
```

⟹ We have added ownership annotations to our language to encode those insights in the code: `TAKE(data_ptr)` `UNTAKE(data_ptr)`

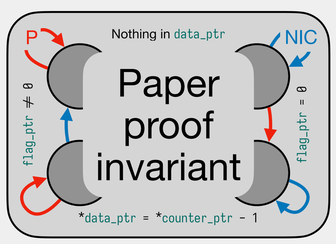
Simple concurrent-program proofs

Using dynamic-ownership annotations and optimistic symbolic execution, we only need to reason about the interleaving of accesses to `flag_ptr`. We can thus reuse the *paper proof invariant* which has 4 states instead of 152!

```
(* key user provided proofs! *)
Lemma I_is_invariant: ∀ (s₁ s₂: state) (t₁ t₂: trace),
  I s₁ t₁ → run_OSE s₁ s₂ t₂ → I s₂ (t₁ ++ t₂).
∵
  inversion 1;
  (* 4 cases, one for each state of the invariant *)
  run_optimistic_symbex;
  (* 8 cases, one for each transition *)
  eauto.
■

Lemma annotations_correct: ∀ s t,
  I s t → state_ok s.
∵ inversion 1; eauto using set_solver. ■

Theorem goal: ∀ s t, run s₀ s t → P t.
∵ eauto using I_implies_P, optimistic_symbex_sound,
    I_is_invariant, annotations_correct. ■
```



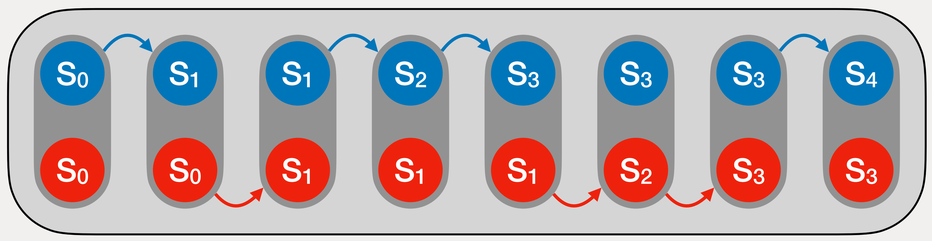
Annotation-correctness side conditions

Proved once and for all

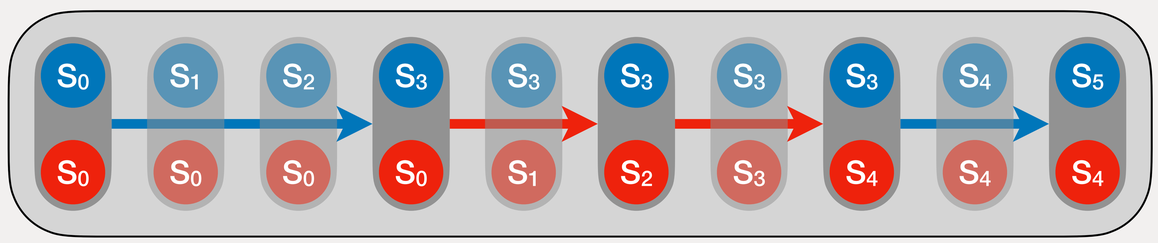
Optimistic symbolic execution

Using the dynamic-ownership annotations, we optimistically execute programs until they access a global memory location or terminate. This generates side conditions to verify the annotations.

Naïve symbolic execution:



Execution with our optimistic symbolic executer:



\+ **Annotation-correctness side conditions**

Results and future work

We have proved correctness of systems that use locks implemented using ***Compare-and-Swap*** or ***Peterson's algorithm*** using the *same invariant as the paper proofs*. We plan to add *modularity* reasoning using **contextual-refinement** and *automatic invariant generation*.