Compositionality Entails Sequentializability

Pranav Garg, P. Madhusudan
University of Illinois at Urbana-Champaign

What is Sequentializability ?

$\boxed{C} \longrightarrow \boxed{S}$

Conc. program	Seq. program
N shared vars	O(N) shared vars
L local vars/thread	O(L) local vars/thread

- Sequentialization requires that the size of sequential program is of the same order as the size of concurrent program.
  - The constant in O(N) and O(L) must be independent of #threads.
- For non-recursive concurrent programs, global simulation is always possible to obtain a sequential program.
  - Leads to a state-space explosion (L^n where *n* is #threads)
  - Is not a sequentialization.
- Recursive programs are even harder.

Why are sequentializations appealing ?

Practically

- Allows the use of analysis tools developed for sequential programs to be used directly for concurrent programs.
  - Deductive verification
  - Symbolic model checking
  - Predicate abstraction
  - Symbolic test case generation

Theoretically

- Intriguing
  - When are concurrent programs sequentializable?

Earlier Results

- Analysis of concurrent programs under a bounded number of context-rounds (no thread creation).
  - Lal and Reps [CAV 2008].
    - Used for bounded model checking of concurrent programs.
    - STORM [Lahiri, Qadeer and Rakamaric] and Poirot from Microsoft.
  - LaTorre, Madhusudan and Parlato [CAV 2009]
    - Shown to be more efficient for explicit model checking with state caching.
  - LaTorre, Madhusudan and Parlato [Unpublished]
    - Parameterized programs with unbounded number of threads but a bounded number of rounds.
- Analysis of concurrent programs with dynamic thread creation under a delay bound.
  - Emmi, Qadeer and Rakamaric [POPL 2011]

Compositionality entails sequentializability

Compositional semantics of a concurrent program with respect to a set of auxiliary variables can be sequentialized.

- It generalizes the prior sequentializations known for the under-approximate context-bounded analysis.
  - Bounding the number of context switches makes them amenable to compositional reasoning.
- Compositional semantics: Over-approximation
  - Can be used to “prove” concurrent programs correct.

Overview

- Jones style rely-guarantee proofs
- Compositional semantics for concurrent program
- Main theorem, intuition behind the sequentialization.
- Experimental results.
- Conclusion.

Rely-Guarantee Proofs

Hoare style method for proving concurrent program.

$$P \models (pre, post, rely, guar)$$

- *pre*, *post* are unary predicates defining subsets of states.
- *rely*, *guar* are binary relations defining transformations to the shared state.

Parallel compositional rule:

$$\frac{\begin{array}{cc} guar_1 \Rightarrow rely_2, & guar_2 \Rightarrow rely_1, \\ P \models (pre, post_1, rely_1, guar_1), & Q \models (pre, post_2, rely_2, guar_2) \\ rely \Rightarrow rely_1,\ rely \Rightarrow rely_2, & (guar_1 \vee guar_2) \Rightarrow guar \end{array}}{P || Q \models (pre, post_1 \wedge post_2, rely, guar)}$$

Auxiliary variables

- Parallel compositional rule in itself is not complete.

- Example:

```
        pre:  x = 0

      P1        ||      P2
atomic {        ||  atomic {
  x := x + 1;   ||    x := x + 1;
}               ||  }

        post:  x = 2
```

- Impossible to come up with consistent *rely-guar* conditions which are strong enough to prove the *post*.

Auxiliary variables

pre: $x = 0 \wedge pc_1 = 0 \wedge pc_2 = 0$

```
      P1            ||        P2

 atomic {           ||   atomic {
   x := x + 1;      ||     x := x + 1;
   pc1 := 1;        ||     pc2 := 1;
 }                  ||   }
```

Auxiliary variables

$post$: $x = 2$

- Since auxiliary variables are not read, the semantics of the concurrent program remains unchanged.

- Auxiliary variables are the local variables which are required to be exposed to the environment to prove the concurrent program compositionally.

Compositional Semantics wrt. Auxiliary variables

$P = P_1 \parallel P_2$

- L_1, L_2 : the set of local variables of the individual threads.
- S: the set of shared variables.
- $A \subseteq L_1 \cup L_2$: the set of auxiliary variables.
- δ_1, δ_2 : local (binary) transition relations of P1 and P2

Compositional Semantics is defined by four sets:

$$R_1 \subseteq (Val_{L_1} \times Val_S \times Val_{A \cap L_2}),$$
$$R_2 \subseteq (Val_{L_2} \times Val_S \times Val_{A \cap L_1}),$$
$$Guar_1, Guar_2 \subseteq (Val_S \times Val_A \times Val_S \times Val_A),$$

R_i : Reachable state in P_i

$Guar_i$: Guarantee that P_i promises.

Compositional Semantics wrt. Auxiliary variables

a) Initialization:

- R_i contains the set $\{(l_i, s, t) \mid l_i \cup s \cup t \in Init \downarrow (L_i \cup A \cup S)\}$.

b) Transitions of P_1: If $(l_1, s, t) \in R_1$ and $\delta_1(l_1, s, l_1', s')$ holds, then

- **Local update:** $(l_1', s', t) \in R_1$.
- **Update to guarantee:** $(s, l_1 \downarrow A \cup t, s', l_1' \downarrow A \cup t) \in Guar_1$.

c) Transitions of P_2: Similar to (**b**)

d) Interference:

- If $(l_1, s, t) \in R_1$ and $(s, l_1 \downarrow A \ \cup \ t, s', t') \in Guar_2$, then $(l_1, s', t' \downarrow L_2) \in R_1$.
- If $(l_2, s, t) \in R_2$ and $(s, l_2 \downarrow A \ \cup \ t, s', t') \in Guar_1$, then $(l_2, s', t' \downarrow L_1) \in R_2$.

$Reach_A = \{(l_1, s, l_2) \mid (l_1, s, l_2 \downarrow A) \in R_1 \ and \ (l_2, s, l_1 \downarrow A) \in R_2\}$

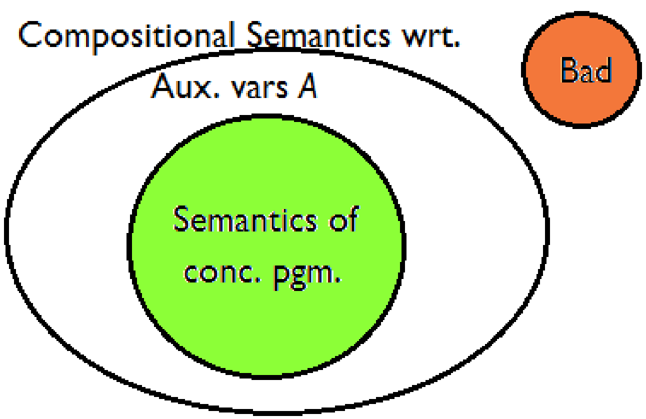
Compositional Semantics wrt. Auxiliary variables

- Existence of a Jones style rely-guarantee proof →

  Compositional semantics of the concurrent program with respect to the corresponding auxiliary variables is correct.

- Note:

  - Compositional semantics is an over-approximation of the standard operational semantics of the concurrent programs.



The main result

- Given a concurrent program *C* with auxiliary variables *A*, we can build a sequential program $S_{C,A}$ such that
  - The compositional semantics of the concurrent program with respect to the auxiliary variables *A* is correct iff $S_{C,A}$ is correct.
  - At any point, the scope of $S_{C,A}$ keeps a constant number of copies of the variables of *C*.

Intuition behind the sequentialization

- Let there be methods G_1 and G_2 which semantically capture the guarantees $Guar_1$ and $Guar_2$ respectively.

$$G_i(\hat{s},\hat{a}) = \{(s,a) \mid (\hat{s},\hat{a},s,a) \in Guar_i\}$$

Once we have G_2, we can compute the reachable states R_1 according to:

```
while(*) {
  if (*) then
    <<simulate a transition of P1>>
  else
    (s,a) := G2(s,a);
  fi
}
```

- Track the local state L_1, shared state and the auxiliary state of the second thread
- On two successive calls to G_2, there is no preservation of the local state of P_2.

Intuition behind the sequentialization

The method G_2 (and similarly G_1) can be implemented as:

```
G2(s#, a#) {
   <<initialize variables of P2>>
   while(*) {
       if (*) then
          <<simulate a transition of P2>>
       else
          (s,a) := G1(s,a);
       fi
   }
   assume (local and shared state is consistent with s#, a#);
   while(*) {
         <<simulate a transition of P2>>
   }
   return (s,a);
}
```

Code for R_2

An example sequentialization

```
decl int x, pc1, pc2;
decl int x*, pc1*, pc2*;
decl bool z, term;
main begin
    G_1();
    return
end
```

```
I_i: if(term = true) then
      return fi
    if(!z & *) then
      x', pc1', pc2' :=
           x*, pc1*, pc2*;
      G_{3-i}();
      z, term:=false, false;
      x*, pc1*, pc2* :=
           x', pc1', pc2'
    fi
    if(x = x* & pc1 = pc1* &
      pc2 = pc2* & *) then
      z := true
    fi
    if(z & *) then
      term := true; return
    fi
```

```
G_1() begin
  z, term:=false, false;
  x*, pc1*, pc2* :=
           x, pc1, pc2
  x, pc1, pc2 := 0, 0, 0;
  main_1();
  assume(term = true);
  return
end

main_1() begin
  decl int x', pc1', pc2';
   I_1
  x := x + 1;
  pc1 := 1;
   I_1
  assert(spec_1);
   I_1
  return
end
```

```
G_2() begin
  z, term:=false, false;
  x*, pc1*, pc2*:=
           x, pc1, pc2
  x, pc1, pc2:=0, 0, 0;
  main_2();
  assume(term = true);
  return
end

main_2() begin
  decl int x', pc1', pc2';
   I_2
  x := x + 2;
  pc2 := 1;
   I_2
  assert(spec_2);
   I_1
  return
end
```

Applications of the theorem

- Deductive verification
- Predicate abstraction

Deductive Verification

Concurrent program *C* with Auxiliary variables *A*	➔	Sequentialization $S_{C,A}$
Jones style Rely/guarantee annotation	➔	Hoare-style pre-post conditions & assertions
- *rely/guar*	→	summaries of G_1 and G_2
- *pre/post*	→	pre/post conditions
- assertions	→	assertions
- loop invariants	→	loop invariants

- Use deductive verification tools (like Boogie/Z3) to verify the correctness of $S_{C,A}$ and hence the correctness of the rely/guarantee proof of *C*.

Deductive Verification

Experimental results

	Concurrent pgm			Sequential pgm		
Programs	#Threads	#LOC	#Lines of annotations	#LOC	#Lines of annotations	Time
X++	2	38	5	113	6	8s
Lock	2	50	9	184	10	122s
Peterson	2	52	35	232	36	145s
Bakery	2	55	8	147	13	18s
ArrayIndexSearch	2	74	17	222	21	126s
GCD	2	78	23	279	29	869s
Bluetooth	unbdd	69	20	276	55	107s

- Manually wrote Jones-style rely/guarantee annotations
- Boogie for Hoare-style verification of the corresponding sequential program.
- Experiments run on Intel dual-core with 1.6 GHz and 1Gb RAM.

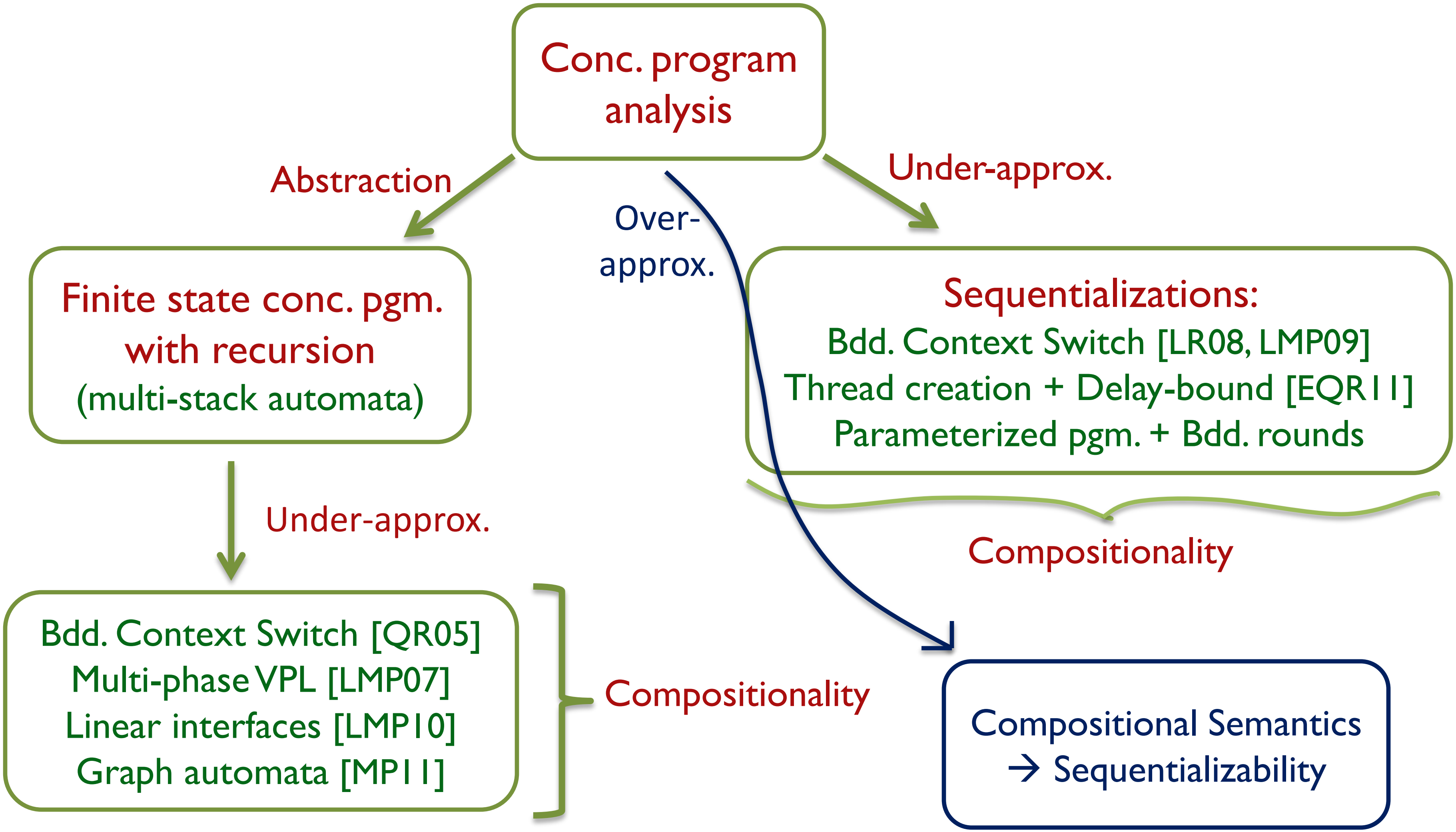
Predicate Abstraction

Concurrent program C + auxiliary variables A determined manually/heuristically ➔ Sequential program $S_{C,A}$

- Use a predicate abstraction tool (like SLAM) to prove $S_{C,A}$ and hence C correct.
  - The procedure is sound but incomplete.
  - If $S_{C,A}$ cannot be proved correct, cannot deduce anything (more auxiliary variables may be needed)
- We get a semi-automatic predicate abstraction tool for concurrent programs.
  - Determining auxiliary variables can be automated

    [Cohen-Namjoshi, Gupta-Popeea-Rybalchenko]
- Used for proving programs: X++, Lock, Bakery and Peterson.

Broader context of the result



Conclusion

- Compositionality entails sequentializability.
- Can be used to prove concurrent programs correct.
- Our result could potentially have a number of applications:
  - Deductive verification
  - Predicate abstraction
  - Symbolic testing
  - Static analysis