An Algorithmic Framework for Synthesis of Concurrent Programs

E. Allen Emerson and Roopsha Samanta

The University of Texas at Austin

Oct 12, 2011

Let's begin with an example

Reader P_1 // Writer P_2

Let's begin with an example

Reader P_1 // Writer P_2

```
P1() {
  while(true) {
    Execute code region IDLE1;
    Execute code region TRY1;
    Execute code region CS1;
}}
```

```
P2() {
  while(true) {
    Execute code region IDLE2;
    Execute code region TRY2;
    Execute code region CS2;
}}
```

Let's begin with an example

Reader P_1 // Writer P_2

```
P1() {
  while(true) {
    Execute code region IDLE1;
    Execute code region TRY1;
    Execute code region CS1;
}}
```

```
P2() {
  while(true) {
    Execute code region IDLE2;
    Execute code region TRY2;
    Execute code region CS2;
}}
```

Mutual exclusion:
$AG(\neg(CS_1 \wedge CS_2))$.
Absence of starvation for reader P_1, provided writer P_2 remains idle:
$AG(TRY_1 \Rightarrow AF(CS_1 \vee \neg IDLE_2))$.
Absence of starvation for writer:
$AG(TRY_2 \Rightarrow AF\, CS_2)$.
Priority of writer over reader for outstanding requests to enter the critical section:
$AG((TRY_1 \wedge TRY_2) \Rightarrow A[TRY_1\, U\, CS_2])$.

Let's begin with an example

Reader P_1 // Writer P_2

```
P1() {
  while(true) {
    Execute code region IDLE1;
    Execute code region TRY1;
    Execute code region CS1;
}}
```

```
P2() {
  while(true) {
    Execute code region IDLE2;
    Execute code region TRY2;
    Execute code region CS2;
}}
```

ϕ

Let's begin with an example

Reader P_1 // Writer P_2

$$\not\models$$

$$\phi$$

Problem definition

Given unsynchronized P_1, P_2 and ϕ such that $P_1 /\!/ P_2 \not\models \phi$, automatically generate $\overline{P}_1$, $\overline{P}_2$ such that $\overline{P}_1 /\!/ \overline{P}_2 \models \phi$.

Our solution

```
main() {
    P1() // P2();
}
```

```
P1() {
   while(true) {
     Execute code region IDLE1;
     Execute code region TRY1;
     Execute code region CS1;
}}
```

```
P2() {
   while(true) {
     Execute code region IDLE2;
     Execute code region TRY2;
     Execute code region CS2;
}}
```

Our solution

```
main() {
    boolean idle1:=1, try1:=0, cs1:=0, idle2:=1, try2:=0, cs2:=0;
    lock l, condition variables cvcs1 , cvcs2 ;
     P1() // P2();
}
```

```
P1() {
   while(true) {
      Execute code region IDLE1;
      lock(l) {
         idle1, try1 := 0,1;
         notify(cvcs2);
      }
       Execute code region TRY1;
      lock(l) {
         while (!idle2)
             wait(cvcs1, l);
         try1, cs1 := 0,1;
      }
       Execute code region CS1;
      lock(l) {
         cs1, idle1 := 0,1;
         notify(cvcs2);
}}}
```

```
P2() {
   while(true) {
      Execute code region IDLE2;
      lock(l) {
         idle2, try2 := 0,1;
      }
       Execute code region TRY2;
      lock(l) {
         while (!(idle1 ∨ try1))
             wait(cvcs2, l);
         try2, cs2 := 0,1;
      }
       Execute code region CS2;
      lock(l) {
         cs2, idle2 := 0,1;
         notify(cvcs1);
}}}
```

Our solution

$$\texttt{Reader}\ \overline{P}_1\ /\!/\ \texttt{Writer}\ \overline{P}_2$$

$$\models$$

$$\phi$$

Outline

- Preliminaries
- Solution framework
- Correctness
- Conclusion

Motivation

- Problem: Shared memory concurrent programs
  - Ubiquitous
  - Hard to write
  - Harder to verify (safety, liveness)
- Proposal: Automatically synthesize synchronization code
  - Only write unsynchronized skeletons
  - Correct-by-construction synchronization code
  - No further verification needed

Motivation

- Problem: Shared memory concurrent programs
  - Ubiquitous
  - Hard to write
  - Harder to verify (safety, liveness)
- Proposal: Automatically synthesize synchronization code
  - Only write unsynchronized skeletons
  - Correct-by-construction synchronization code
  - No further verification needed

Motivation

- Problem: Shared memory concurrent programs
  - Ubiquitous
  - Hard to write
  - Harder to verify (safety, liveness)
- Proposal: Automatically synthesize synchronization code
  - Only write unsynchronized skeletons
  - Correct-by-construction synchronization code
  - No further verification needed

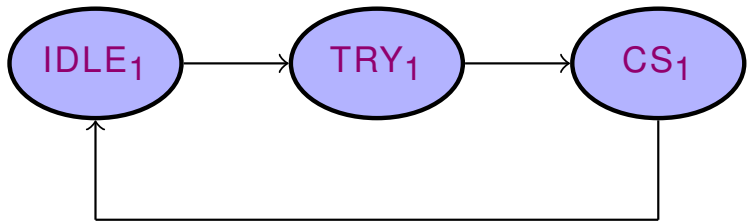
Motivation

- Problem: Shared memory concurrent programs
  - Ubiquitous
  - Hard to write
  - Harder to verify (safety, liveness)
- Proposal: Automatically synthesize synchronization code
  - Only write unsynchronized skeletons
  - Correct-by-construction synchronization code
  - No further verification needed

Groundwork

- Unsynchronized skeletons $\mapsto$ state-machines
- Code regions $\mapsto$ states (atomic propositions)
- Control-flow $\mapsto$ transition relation

```
P1() {
   while(true) {
     Execute code region IDLE1;
     Execute code region TRY1;
     Execute code region CS1;
 }}
```



Groundwork

- Unsynchronized skeletons $\mapsto$ state-machines
- Code regions $\mapsto$ states (atomic propositions)
- Control-flow $\mapsto$ transition relation

- Interleaved, asynchronous computation

- CTL specification
  - Safety: $\mathsf{AG}(\neg(\mathsf{CS}_1 \wedge \mathsf{CS}_2))$
  - Liveness: $\mathsf{AG}(\mathsf{TRY}_2 \Rightarrow \mathsf{AF}\,\mathsf{CS}_2)$

Recall problem definition

Given skeletons P_1, P_2 and ϕ such that $P_1 /\!/ P_2 \not\models \phi$, automatically generate $\overline{P}_1$, $\overline{P}_2$ such that $\overline{P}_1 /\!/ \overline{P}_2 \models \phi$.

Solution framework

- Step 1: Synthesize *synchronization skeletons*, P_1^s, P_2^s
  - High-level synchronization actions
    - Guarded commands
  - $P_1^s \,/\!/\, P_2^s \models \phi$
- Step 2: Mechanically translate P_1^s, P_2^s into $\overline{P}_1$, $\overline{P}_2$
  - Low-level synchronization code
  - Correctness-preserving translation, i.e., $\overline{P}_1 \,/\!/\, \overline{P}_2 \models \phi$

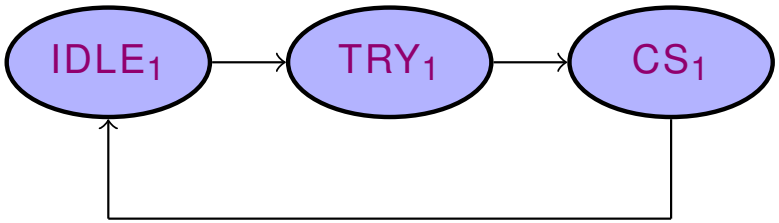
Solution framework

- Step 1: Synthesize *synchronization skeletons*, P_1^s, P_2^s
  - High-level synchronization actions
    - Guarded commands
  - $P_1^s \mathbin{/\!\!/} P_2^s \models \phi$
- Step 2: Mechanically translate P_1^s, P_2^s into $\overline{P}_1$, $\overline{P}_2$
  - Low-level synchronization code
    - Monitors (`wait` and `notify`), mutex locks
  - Correctness-preserving translation, i.e., $\overline{P}_1 \mathbin{/\!\!/} \overline{P}_2 \models \phi$

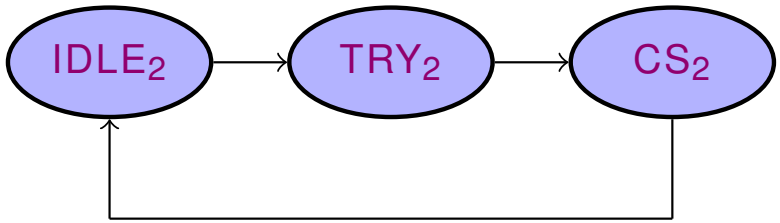
Step 1: High-level synchronization

[EmersonClarke82]

Reader P_1:



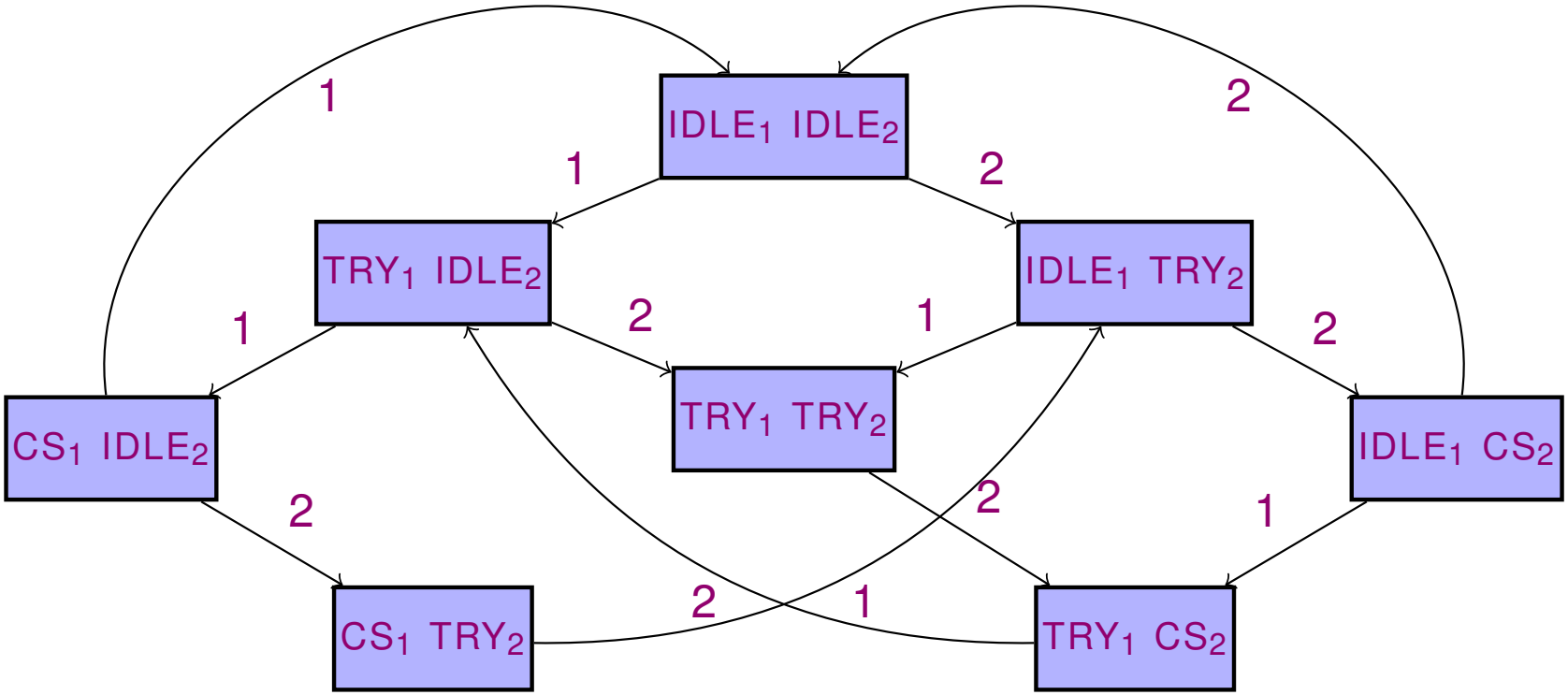
Writer P_2:



ϕ

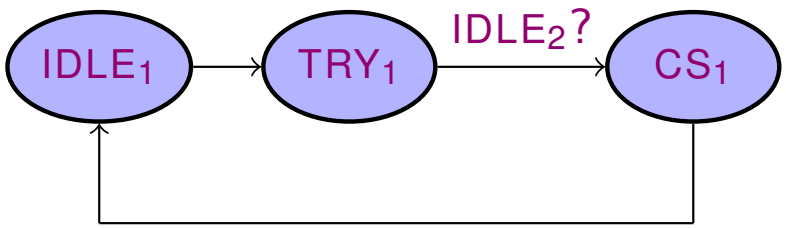
Step 1: High-level synchronization

$M \models phi$:

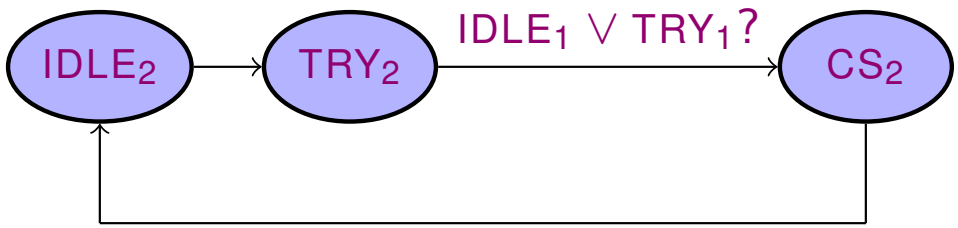


Step 1: High-level synchronization

Reader P_1^s:



Writer P_2^s:



$$P_1^s /\!/ P_2^s \models \phi$$

Step 2: Low-level synchronization

- Obtain $\overline{P}_1$, $\overline{P}_2$ from P_1^s, P_2^s
- Monitors (`wait` and `notify`), mutex locks
- $\overline{P}_1 /\!\!/ \overline{P}_2 \models \phi$

Step 2: Coarse-grained synchronization

- Declare Boolean shared variables
- Declare (*single*) lock and condition variables

```
main() {
   boolean idle1:=1, try1:=0, cs1:=0, idle2:=1, try2:=0, cs2:=0;
   lock l, condition variables cvcs1, cvcs2;
   P1c() // P2c();
}
```

Step 2: Coarse-grained synchronization

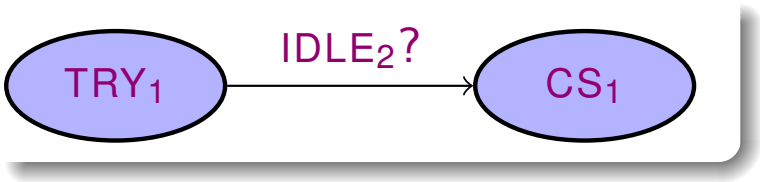
- Declare Boolean shared variables
- Declare (*single*) lock and condition variables

```
main() {
   boolean idle1:=1, try1:=0, cs1:=0, idle2:=1, try2:=0, cs2:=0;
   lock l, condition variables cvcs1, cvcs2;
   P1c() // P2c();
}
```

Step 2: Coarse-grained synchronization

- Compile each guarded command in P_1^s, P_2^s into a coarse-grained synchronization region

Step 2: Coarse-grained synchronization




```
Execute TRY1;
lock(l) {
  while (!idle2)
      wait(cv_cs1, l);
  try1, cs1 := 0, 1;
}
Execute CS1;
```

Step 2: Coarse-grained synchronization

```
main() {
    boolean idle1:=1, try1:=0, cs1:=0, idle2:=1, try2:=0, cs2:=0;
    lock l, condition variables cvcs1 , cvcs2 ;
    P1c() // P2c();
}
```

```
P1c() {
   while(true) {
      Execute code region IDLE1;
      lock(l) {
         idle1, try1 := 0,1;
         notify(cvcs2);
      }
      Execute code region TRY1;
      lock(l) {
         while (!idle2)
             wait(cvcs1 , l);
         try1, cs1 := 0,1;
      }
      Execute code region CS1;
      lock(l) {
         cs1, idle1 := 0,1;
         notify(cvcs2);
}}}
```

```
P2c() {
   while(true) {
      Execute code region IDLE2;
      lock(l) {
         idle2, try2 := 0,1;
      }
      Execute code region TRY2;
      lock(l) {
         while (!(idle1 ∨ try1))
             wait(cvcs2 , l);
         try2, cs2 := 0,1;
      }
      Execute code region CS2;
      lock(l) {
         cs2, idle2 := 0,1;
         notify(cvcs1);
}}}
```

Step 2: Fine-grained synchronization

- Declare Boolean shared variables
- Declare mutex locks, monitor locks and condition variables

```
main() {
   boolean idle1:=1, try1:=0, cs1:=0, idle2:=1, try2:=0, cs2:=0;
   lock lidle1, ltry1, lcs1, lidle2, ltry2, lcs2;
   lock lcvcs1, condition variable cvcs1;
   lock lcvcs2, condition variable cvcs2;
   P1^f() // P2^f();
}
```

Step 2: Fine-grained synchronization

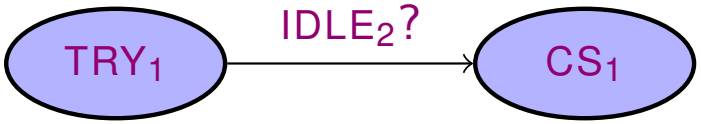
- Declare Boolean shared variables
- Declare mutex locks, monitor locks and condition variables

```
main() {
   boolean idle1:=1, try1:=0, cs1:=0, idle2:=1, try2:=0, cs2:=0;
   lock l_idle1, l_try1, l_cs1, l_idle2, l_try2, l_cs2;
   lock l_cvcs1, condition variable cv_cs1;
   lock l_cvcs2, condition variable cv_cs2;
   P1^f() // P2^f();
}
```

Step 2: Fine-grained synchronization

- Compile each guarded command in P_1^s, P_2^s into a fine-grained synchronization region

Step 2: Fine-grained synchronization




```
Execute TRY1;
lock(l_cv_cs1) {
  while (!Guard_cs1())
      wait(cv_cs1, l_cv_cs1);
}
Execute CS1;
```

```
boolean Guard_cs1() {
  lock((l_try1, l_cs1, l_idle2) {
   if (idle2) {
    try1, cs1 := 0, 1;
    return(true);
   }
   else return(false);
}}
```

Step 2: Fine-grained synchronization

```
main() {
    boolean idle1:=1, try1:=0, cs1:=0, idle2:=1, try2:=0, cs2:=0;
    lock l_idle1, l_try1, l_cs1, l_idle2, l_try2, l_cs2;
    lock l_cvcs1, condition variable cvcs1; lock l_cvcs2, condition variable cvcs2;
    P1^f() // P2^f();}
```

```
P1^c() {
   while(true) {
      Execute code region IDLE1;
      lock(l_idle1, l_try1) {
         idle1, try1 := 0,1;}
      lock(()l_cvcs2) {
         notify(cvcs2);}
      Execute code region TRY1;
      lock(l_cvcs1) {
         while (!Guardcs1())
             wait(cvcs1, l_cvcs1);}
      Execute code region CS1;
      lock(l_idle1, l_cs1) {
         cs1, idle1 := 0,1;}
      lock(()l_cvcs2) {
         notify(cvcs2);}}}
```

```
P2^c() {
   while(true) {
      Execute code region IDLE2;
      lock(l_idle2, l_try2) {
         idle2, try2 := 0,1;}
      Execute code region TRY2;
      lock(l_cvcs2) {
         while (!Guardcs2())
             wait(cvcs2, l_cvcs2);}
      Execute code region CS2;
      lock(l_idle2, l_cs2) {
         cs2, idle2 := 0,1;}
      lock(()l_cvcs1) {
         notify(cvcs1);}}}
```

Review

- What we have so far . . .
  - Fully algorithmic synthesis of synchronization
    - Algorithmic front-end for high-level synchronization
    - Algorithmic back-end for low-level synchronization
  - Coarse *and* fine-grained synchronization
- What's remaining?
  - Correctness
  - What properties can we handle?

Review

- What we have so far . . .
  - Fully algorithmic synthesis of synchronization
    - Algorithmic front-end for high-level synchronization
    - Algorithmic back-end for low-level synchronization
  - Coarse *and* fine-grained synchronization
- What's remaining?
  - Correctness
  - What properties can we handle?

Correspondence Lemmas

[Coarse-grained Correspondence]:
Given an $ACTL \setminus X$ formula ϕ, $P_1^s /\!\!/ P_2^s \models \phi \Rightarrow P_1^c /\!\!/ P_2^c \models \phi$.

[Fine-grained Correspondence]:
Given an $ACTL \setminus X$ formula ϕ, $P_1^s /\!\!/ P_2^s \models \phi \Rightarrow P_1^f /\!\!/ P_2^f \models \phi$.

Correctness

Sound and complete for consistent temporal specifications!

Contributions

- Fully algorithmic synthesis of synchronization
  - Algorithmic front-end for high-level synchronization
  - Algorithmic back-end for low-level synchronization
- Coarse and fine-grained synchronization
- Safety and liveness
- Sound and complete (for $ACTL \setminus X$)

Related Work

- Inference of high-level synchronization, guarded commands: [EC82, VYY09]
- Mapping of high-level to low-level synchronization: [DDHM01, Y-KB02]
- Lock-inference, locking granularity: [EFJM07, CCG08]
- Sketching: [S-LRBE05]
- Open systems [PR89]