An Annotation Assistant for Interactive Debugging of Programs with Common Synchronization Idioms

Tayfun Elmas, Ali Sezgin, Serdar Tasiran

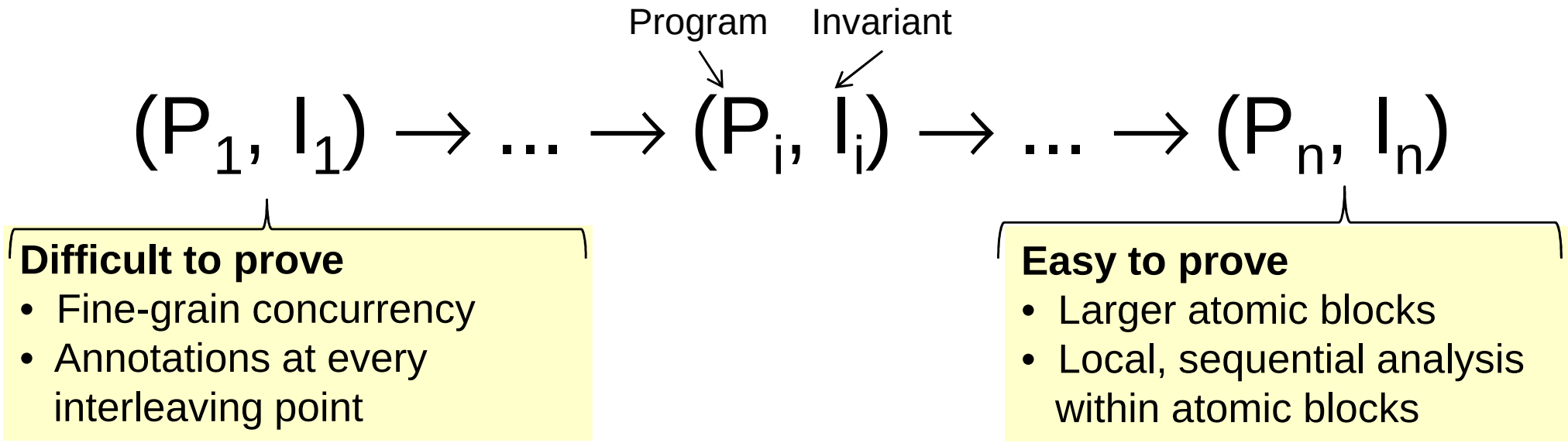
Koç University, İstanbul, Turkey

Shaz Qadeer

Microsoft Research, Redmond, WA

PADTAD 2009

The QED method [POPL'09]



- Central idea: Atomicity as proof tool
- Proof strategy: Transform program by enlarging atomic blocks
- P_1 certified correct by analyzing P_n
  - Soundness: Starting from I_n , P_n satisfies all assertions
    → P_1 satisfies all assertions

Synchronization idiom

- Well-known pattern to restrict the amount of concurrency
  - Mutual-exclusion, reentrant locks, readers/writer lock, events
  - Implementations must guarantee the pattern
    - Library or custom implementation

- Example: Readers/writer lock
  - Two kinds of critical sections: Readers, writers
  - At any time: Multiple readers or only one writer

What we propose

- Application of tool to, for example, readers/writer lock
  - Identify the code implementing idiom
  - Annotate critical sections protected by idiom
  - Show atomicity of critical regions
  - Determine errors about use of idiom
    - Conflicting actions due to missing synchronization
    - Code disobeying the pattern of readers/writer lock
    - Incorrect implementation of idiom by the program

Example


```
T1  x := 0  ;  v := 0

   T2              T3                    T4                          T5
Update(1)  ||  Update(2)  ||  (x1,v1) := Read()  ||  (x2,v2) := Read()

T1  assert (v1 == v2) ==> (x1 == x2)
```

```
Update(k)
----------
x := k;
v := v + 1;
```

```
Read()
----------
return (x, v);
```

Readers/writer lock


```
T1  x := 0  ;  v := 0

   T2              T3                    T4                          T5
Update(1)  ||  Update(2)  ||  (x1,v1) := Read()  ||  (x2,v2) := Read()

T1  assert (v1 == v2) ==> (x1 == x2)
```

```
Update(k)
----------
AcqWrite();
x := k;
lv := v;
v := lv + 1;
RelWrite();
```

```
Read()
----------
AcqRead();
lx := x;
lv := v;
RelRead();
return (lx, lv);
```

Annotations


```
w == 0          : Write lock free
w == tid        : Write lock held by current thread
r[tid] == true  : Read lock held by current thread
r[tid] == false : Read lock not held by current thread
```

```
Update(k)
-----------
< AcqWrite(); w := tid; >
< assert w == tid; x := k; >
< assert w == tid; lv := v; >
< assert w == tid; v := lv + 1; >
< assert w == tid; RelWrite(); w := 0; >
```

```
Read()
-----------
< AcqRead(); w := 0; r[tid] := true; >
< assert w == 0; lx := x; >
< assert w == 0; lv := v; >
< assert w == 0; RelRead(); r[tid] := false; >
return (lx, lv);
```

Conflict check

Annotations indicate conflicting actions not enabled from the same state.

Update by T3 (tid_3)
< assert w == tid_3; x := k_3; >

Read by T4 (tid_4)
< assert w == 0; lx_4 := x; >

Thread id cannot be 0: **tid_3 != 0**

Update by T2 (tid_2)
< assert w == tid_2; x := k_2; >

Update by T3 (tid_3)
< assert w == tid_3; x := k_3; >

Run by diffierent threads: **tid_2 != tid_3**

Verifying annotation


```
Update(k)
-----------
atomic {
  AcqWrite(); w := tid;
 ✓assert w == tid; x := k;
 ✓assert w == tid; lv := v;
 ✓assert w == tid; v := lv + 1;
 ✓assert w == tid; RelWrite(); w := 0;
}
Read()
-----------
atomic {
  AcqRead(); w := 0; r[tid] := true;
✓assert w == 0; lx := x;
✓assert w == 0; lv := v;
✓assert w == 0; RelRead(); r[tid] := false;
  return (lx, lv);
}
```

Insufficient synchronization


```
Update(k)
-----------
AcqWrite();
x := k;
lv := v;
v := lv + 1;
RelWrite();
```

```
Read()
-----------
lx := x;
lv := v;
return (lx, lv);
```

⟶ Read region not protected by lock

- Symptom: **`Read`** not annotated
  - **`Update`** and **`Read`** conflict: Atomicity computation fails
  - Tool shows the conflicting lines, missing annotation

```
Update(k)
-----------
< AcqWrite(); w := tid; >
< assert w == tid; x := k; >
< assert w == tid; lv := v; >
< assert w == tid; v := lv + 1; >
< assert w == tid; RelWrite(); w := 0; >
```

Conflict on **x** !

```
Read()
-----------
lx := x;
lv := v;
return (lx, lv);
```

Incorrect use of idiom


```
Update(k)
-----------
x := k;
lv := v;
v := lv + 1;
RelWrite();
```


Missing acquire before release

- Symptom: Tool checks if idiom is used correctly before annotating
  - Warning about missing acquire

Incorrect implementation


```
AcqWrite()
-----------
atomic {
  await (reads == 0);
  write := true;
}
```

Should have been: **(reads == 0 && write == false)**

- Symptom: **Update** not annotated
  - **Update** and **Read** conflict: Atomicity computation fails
  - Tool shows the conflicting lines, missing annotation

```
Update(k)                          Read()
-----------                        -----------
AcqWrite();                        < AcqRead(); w := 0; r[tid] := true; >
x := k;          Conflict on x !   < assert w == 0; lx := x; >
lv := v;                           < assert w == 0; lv := v; >
v := lv + 1;                       < assert w == 0; RelRead(); r[tid]:=false; >
RelWrite();                        return (lx, lv);
```

Specifying idiom

- Idiom template: Abstract description of idiom using auxiliary variables
  - Independent of implementation of idiom

- Synchronization state
  - **`w: int`**
    - **`w == 0`** : Write lock free
    - **`w == tid`** : Write lock held by **`tid`**
  - **`r: int -> bool`**
    - **`r[tid] == false`**: Read lock not held by **`tid`**
    - **`r[tid] == true`** : Read lock held by **`tid`**

- Invariant: **`(exists t!=0. w == t) ==> (forall u. !r[u])`**
- Atomic operations:
  - Acquire write: **`w == 0 --> w == tid`**
  - Release write: **`w == tid -->  w == 0`**
  - Acquire read: **`!r[tid] --> r[tid]`**
  - Release read: **`r[tid] --> r[tid]`**
  - Non-synch actions: **`w`** and **`r`** remain unchanged

Implementing readers/writer lock


```
Globals: reads: int, write: bool

AcqWrite()
-----------
atomic {
  await (reads == 0 && write == false);
  write := true;
}

RelWrite()
-----------
atomic {
  write := false;
}

AcqRead()
-----------
atomic {
  await (write == false);
  reads := reads + 1;
}

RelRead()
-----------
atomic {
  reads := reads - 1;
}
```

Connecting specification to implementation

- Identifying code implementing readers/writer lock
  - Idiom formulas
  - P_R: States where reader lock is held
    - **`reads > 0 && write == false`**
  - P_W: States where writer lock is held
    - **`reads == 0 && write == true`**
- Associating abstract description with implementation
  - Invariant:

    1. P_R ==> !P_W

    2. !P_W <==> (w == 0)

    3. P_R ==> (exists t. r[t])
  - Transition: How actions of the program modifies **`r`** and **`w`**

Conclusion

- Manually annotating program is difficult, error-prone
  - Too weak: Insufficient to show non-interference
  - Too strong: Adds false non-interference
- Automated annotations: Precise and reflects idiom's semantics
  - Useful hints on the use of idiom
  - Idiom template: Generic handling of idiom independent of implementations
- Evidence from the literature
  - Idiom implementations, programs using idioms
- Supported idioms:
  - Mutual-exclusion, reentrant locks, readers/writer lock, events
- Future work: Barriers, fork/join parallelism