Part II: Atomicity for Software Model Checking

```
Class Account {
   int balance;
   static int MIN = 0, MAX = 100;

   bool synchronized deposit(int n) {
      int t = balance + n;
      if (t > MAX) return false;
      balance = t;
      assert(MIN ≤ balance ≤ MAX);
      return true;
   }

   bool synchronized withdraw(int n) {
      int t = balance – n;
      if (t < MIN) return false;
      balance = t;
      assert(MIN ≤ balance ≤ MAX);
      return true;
   }
}
```

```
Account a = new Account();
Account b = new Account();
async a.deposit(5);
async b.withdraw(10);
```

Zing

- Model checker for concurrent software
  - assertions, deadlocks
- Rich input language
  - Procedures, dynamic object creation, dynamic thread creation
  - Shared-memory (via globals and channels)
  - Message-passing (via channels)
- Joint work with Tony Andrews, Jakob Rehof, and Sriram Rajamani
- http://www.research.microsoft.com/zing

Analysis of concurrent programs is difficult (1)

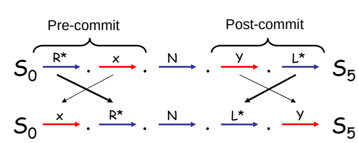
- Finite-data single-procedure program
  - n lines
  - m states for global data variables
- 1 thread
  - n * m states
- K threads
  - $(n)^K$ * m states

The theory of movers (Lipton 75)

- R: right movers
  - lock acquire
- L: left movers
  - lock release
- B: both right + left movers
  - variable access holding lock
- N: non-movers
  - access unprotected variable

Transaction

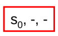
Lipton: any sequence (R|B)*; [N] ; (L|B)* is a transaction



Other threads need not be scheduled in the middle of a transaction

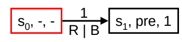
Algorithm:
1. Schedule all threads in the initial state.
2. For each state s discovered by executing a thread t:
   - If s is inside a transaction, schedule only thread t from s.
   - Otherwise, schedule all threads from s.

Instrumented state: State, Phase, Tid



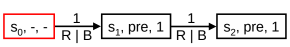
Algorithm:
1. Schedule all threads in the initial state.
2. For each state s discovered by executing a thread t:
   - If s is inside a transaction, schedule only thread t from s.
   - Otherwise, schedule all threads from s.

Instrumented state: State, Phase, Tid



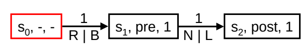
Algorithm:
1. Schedule all threads in the initial state.
2. For each state s discovered by executing a thread t:
   - If s is inside a transaction, schedule only thread t from s.
   - Otherwise, schedule all threads from s.

Instrumented state: State, Phase, Tid



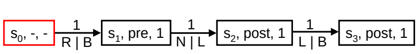
Algorithm:
1. Schedule all threads in the initial state.
2. For each state s discovered by executing a thread t:
   - If s is inside a transaction, schedule only thread t from s.
   - Otherwise, schedule all threads from s.

Instrumented state: State, Phase, Tid



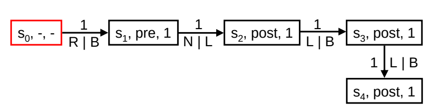
Algorithm:
1. Schedule all threads in the initial state.
2. For each state s discovered by executing a thread t:
   - If s is inside a transaction, schedule only thread t from s.
   - Otherwise, schedule all threads from s.

Instrumented state: State, Phase, Tid



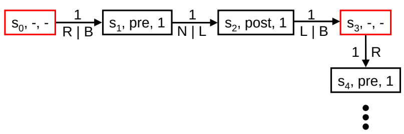
Algorithm:
1. Schedule all threads in the initial state.
2. For each state s discovered by executing a thread t:
   - If s is inside a transaction, schedule only thread t from s.
   - Otherwise, schedule all threads from s.

Instrumented state: State, Phase, Tid



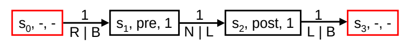
Algorithm:
1. Schedule all threads in the initial state.
2. For each state s discovered by executing a thread t:
   - If s is inside a transaction, schedule only thread t from s.
   - Otherwise, schedule all threads from s.

Instrumented state: State, Phase, Tid



Algorithm:
1. Schedule all threads in the initial state.
2. For each state s discovered by executing a thread t:
   - If s is inside a transaction, schedule only thread t from s.
   - Otherwise, schedule all threads from s.

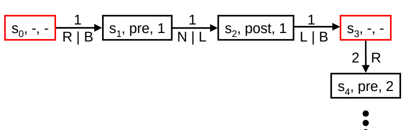
Instrumented state: State, Phase, Tid



Schedule thread 2

Algorithm:
1. Schedule all threads in the initial state.
2. For each state s discovered by executing a thread t:
   - If s is inside a transaction, schedule only thread t from s.
   - Otherwise, schedule all threads from s.

Instrumented state: State, Phase, Tid




```
Class Account {
   int balance;
   static int MIN = 0, MAX = 100;

   bool synchronized deposit(int n) {
      int t = balance + n;
      if (t > MAX) return false;
      balance = t;
      assert(MIN ≤ balance ≤ MAX);
      return true;
   }

   bool synchronized withdraw(int n) {
      int t = balance – n;
      if (t < MIN) return false;
      balance = t;
      assert(MIN ≤ balance ≤ MAX);
      return true;
   }
}
```

```
Account a = new Account();
Account b = new Account();
async a.deposit(5);
async b.withdraw(10);
```

Execution of a.deposit(5) is a transaction.

Execution of b.withdraw(10) is a transaction.

ZING explores two interleavings only!

Unsoundness problem

What if a transaction does not terminate?

Initially g == 0;

T1	T2

g = 1;
while (true)
  skip;
	assert(g == 0);

Thread 2 never gets scheduled here!

ZING: Work in progress

- Algorithm for sound transaction-based model checking
- Inferring mover information for accesses to the heap and globals

Related work

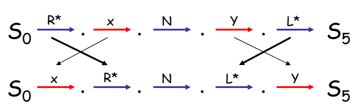
- Partial-order reduction
  - stubborn sets (Valmari 91), ample sets (Peled 96), sleep sets (Godefroid 96)
  - used mostly for message-passing systems (no shared memory)
- Bogor model checker (Dwyer et al. 04)
  - applied classic partial-order reduction to shared-memory Java programs
- Transaction-based reduction (Stoller-Cohen 03)

Analysis of concurrent programs is difficult (2)

- Finite-data program with procedures
  - n lines
  - m states for global data variables
- 1 thread
  - Infinite number of states
  - Can still decide assertions in $O(n * m^3)$
  - SLAM, ESP, BLAST implement this algorithm
- $K \geq 2$ threads
  - Undecidable!

Transaction

Lipton: any sequence (R+B)*; (N+ε) ; (L+B)* is a transaction



Other threads need not be scheduled in the middle of a transaction

⇒ Transactions may be summarized

Summarization for sequential programs

- Procedure summarization (Sharir-Pnueli 81, Reps-Horwitz-Sagiv 95) is the key to efficiency

```
int x;                      void main() {
                              …
void incr_by_2() {            x = 0;
    x++;                      incr_by_2();
    x++;                      …
}                             x = 0;
                              incr_by_2();
                              …
                            }
```

- Bebop, ESP, Moped, MC, Prefix, …

Assertion checking for sequential programs

- Boolean program with:
  - *g* = number of global vars
  - *m* = max. number of local vars in any scope
  - k = size of the CFG of the program
- Complexity is $O(k \times 2^{O(g+m)})$, linear in the size of CFG
- Summarization enables termination in the presence of recursion

Assertion checking for concurrent programs

Ramalingam 00:

There is no algorithm for assertion checking of concurrent boolean programs, even with only two threads.

Our contribution

- Precise semi-algorithm for verifying properties of concurrent programs
  - based on model checking
  - procedure summarization for efficiency
- Termination for a large class of concurrent programs with recursion and shared variables
- Generalization of precise interprocedural dataflow analysis for sequential programs

What is a summary in sequential programs?

- Summary of a procedure P = Set of all (pre-state → post-state) pairs obtained by invocations of P

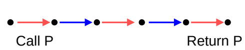
```
int x;

void incr_by_2() {
    x++;
    x++;
}
```

```
void main() {
   …
   x = 0;
   incr_by_2();
   …
   x = 0;
   incr_by_2();
   …
   x = 1;
   incr_by_2();
   …
}
```

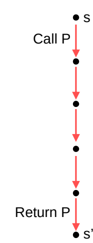
x	→	x'
0	→	2
1	→	3

What is a summary in concurrent programs?

- Unarticulated so far
- Naïve extension of summaries for sequential programs do not work



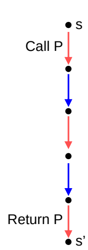
Attempt 1



Advantage: summary computable as in a sequential program

Disadvantage: summary not usable for executions with interference from other threads

Attempt 2



Advantage: Captures all executions

Disadvantage: s and s' must comprise full program state
- summaries are complicated
- do not offer much reuse

If a procedure body is a single transaction, summarize as in a sequential program

```
bool available[N];
mutex m;

int getResource() {
    int i = 0;
L0:   acquire(m);
L1:   while (i < N) {
L2:     if (available[i]) {
L3:       available[i] = false;
L4:       release(m);
L5:       return i;
        }
L6:     i++;
      }
L7:   release(m);
L8:   return i;
}
```

Choose N = 2

Summaries:

⟨ m, (a[0],a[1]) ⟩ → ⟨ i', m', (a[0]',a[1]') ⟩

⟨ 0, (0, 0) ⟩ → ⟨ 2, 0, (0,0) ⟩
⟨ 0, (0, 1) ⟩ → ⟨ 1, 0, (0,0) ⟩
⟨ 0, (1, 0) ⟩ → ⟨ 0, 0, (0,0) ⟩
⟨ 0, (1, 1) ⟩ → ⟨ 0, 0, (0,1) ⟩

Transactional procedures

- In the Atomizer benchmarks (Flanagan-Freund 04), a majority of procedures are transactional

What if a procedure body comprises multiple transactions?

```
bool available[N];
mutex m[N];

int getResource() {
    int i = 0;
L0:   while (i < N) {
L1:     acquire(m[i]);
L2:     if (available[i]) {
L3:       available[i] = false;
L4:       release(m[i]);
L5:       return i;
        } else {
L6:       release(m[i]);
        }
L7:     i++;
      }
L8:   return i;
}
```

Choose N = 2

Summaries:

⟨ pc,i,(m[0],m[1]),(a[0],a[1]) ⟩ → ⟨ pc',i',(m[0]',m[1]'),(a[0]',a[1]') ⟩

⟨ L0, 0, (0,*), (0,*) ⟩ → ⟨ L1, 1, (0,*), (0,*) ⟩

⟨ L0, 0, (0,*), (1,*) ⟩ → ⟨ L5, 0, (0,*), (0,*) ⟩

⟨ L1, 1, (*,0), (*,0) ⟩ → ⟨ L8, 2, (*,0), (*,0) ⟩

⟨ L1, 1, (*,0), (*,1) ⟩ → ⟨ L5, 1, (*,0), (*,0) ⟩

What if a transaction
1. starts in caller and ends in callee?
2. starts in callee and ends in caller?

```
int x;
mutex m;

void foo() {                void bar() {
  acquire(m);
  x++;          1
                              release(m);
  bar();
                2             acquire(m);
  x--;
  release(m);                }
}
```

What if a transaction
1. starts in caller and ends in callee?
2. starts in callee and ends in caller?

```
int x;
mutex m;

void foo() {                void bar() {
  acquire(m);
  x++;          1
                              release(m);
  bar();
                2             acquire(m);
  x--;
  release(m);                }
}
```

Solution:
1. Split the summary into pieces
2. Annotate each piece to indicate whether transaction continues past it

Two-level model checking

- Top level performs state exploration
- Bottom level performs summarization
- Top level uses summaries to explore reduced set of interleavings
  - Maintains a stack for each thread
  - Pushes a stack frame if annotated summary edge ends in a call
  - Pops a stack frame if annotated summary edge ends in a return

Termination

- Theorem:
  - If all recursive functions are transactional, then our algorithm terminates.
  - The algorithm reports an error iff there is an error in the program.

Concurrency + recursion

```
int g = 0;
mutex m;

void foo(int r) {
L0:  if (r == 0) {
L1:     foo(r);
     } else {
L2:     acquire(m);
L3:     g++;
L4:     release(m);
     }
L5:  return;
}

void main() {
    int q =
    choose({0,1});
M0:  foo(q);
M1:  acquire(m)
M2:  assert(g >= 1);
M3:  release(m);
M4:  return;
}

Prog = main() || main()
```

Summaries for foo:

⟨ pc,r,m,g ⟩ → ⟨ pc',r',m',g' ⟩

⟨ L0,1,0,0 ⟩ → ⟨ L5,1,0,1 ⟩

⟨ L0,1,0,1 ⟩ → ⟨ L5,1,0,2 ⟩

Sequential programs

- For a sequential program, the whole execution is a transaction
- Algorithm behaves exactly like classic interprocedural dataflow analysis

Related work

- Summarizing sequential programs
  - Sharir-Pnueli 81, Reps-Horwitz-Sagiv 95, Ball-Rajamani 00, Esparza-Schwoon 01
- Concurrency+Procedures
  - Duesterwald-Soffa 91, Dwyer-Clarke 94, Alur-Grosu 00, Esparza-Podelski 00, Bouajjani-Esparza-Touili 02