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);

• 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

Pre-commit

Post-commit

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: \( s_0 \), pre, 1

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: \( s_0 \), pre, 1

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: \( s_0 \), pre, 1

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: \( s_0 \), pre, 1

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: \( s_0 \), pre, 1

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: \( s_0 \), pre, 1

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: \( s_0 \), pre, 1
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:

```
<table>
<thead>
<tr>
<th>State</th>
<th>Phase</th>
<th>Tid</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>s1</td>
<td>pre</td>
<td>1</td>
</tr>
<tr>
<td>s2</td>
<td>post</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Instrumented state:

```
<table>
<thead>
<tr>
<th>State</th>
<th>Phase</th>
<th>Tid</th>
</tr>
</thead>
<tbody>
<tr>
<td>s3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>s4</td>
<td>pre</td>
<td>1</td>
</tr>
</tbody>
</table>
```

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:

```
<table>
<thead>
<tr>
<th>State</th>
<th>Phase</th>
<th>Tid</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>s1</td>
<td>pre</td>
<td>1</td>
</tr>
<tr>
<td>s2</td>
<td>post</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Instrumented state:

```
<table>
<thead>
<tr>
<th>State</th>
<th>Phase</th>
<th>Tid</th>
</tr>
</thead>
<tbody>
<tr>
<td>s3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>s4</td>
<td>pre</td>
<td>1</td>
</tr>
</tbody>
</table>
```

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
 g = 1;
while (true) skip;
T2
```

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 $O(n \cdot m^2)$
  - SLAM, ESP, BLAST implement this algorithm
- $K \geq 2$ threads
  - Undecidable!

Transaction

Lipton: any sequence $(R+B)^*; (N+\varepsilon); (L+B)^*$ is a transaction

```
S_0 \xrightarrow{R^*} \mathcal{S} \xrightarrow{N} \mathcal{S} \xrightarrow{R^*} \mathcal{S}
```

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 incr_by_2() {
    x++;
    x++;
}
void main() {
    ... x = 0;
    incr_by_2();
    ...
    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 $P = \text{Set of all (pre-state } \Rightarrow \text{ post-state) pairs obtained by invocations of } P$}

```c
int x; void main() { x → x'
  void incr_by_2() { x = 0; incr_by_2(); x → x + 2; incr_by_2(); x = 1; incr_by_2(); ...
}
void main() { x = 0; incr_by_2(); y = 2; incr_by_2(); y = 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

```c
bool available[N]; mutex m;
int get_resource() { int x = 0; L0: acquire(m); L1: while (i < N) { L2: if (available[i]) { L3: available[i] = false; L4: release(m); L5: return i; }
```
Transactional procedures

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

What if a procedure body comprises multiple transactions?

```c
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;
}
```

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) {
   if (r == 0) {
      foo(r);
   } else {
      acquire(m);
      g++;
      release(m);
   }
   return;
}

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

Summaries for foo:
\[ \langle pc, r, m, g \rangle \Rightarrow \langle pc', r', m', g' \rangle \]
\[ \langle L0, 1, 0, 0 \rangle \Rightarrow \langle L5, 1, 0, 2 \rangle \]

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