Algorithmic Verification of Concurrent Programs

Shaz Qadeer
Microsoft Research
Reliable concurrent software?

• Correctness problem
  – does program behave correctly for all inputs and all interleavings?

• Bugs due to concurrency are insidious
  – non-deterministic, timing dependent
  – data corruption, crashes
  – difficult to detect, reproduce, eliminate
Demo

• Debugging concurrent programs is hard
Program verification

• Program verification is undecidable
  – even for sequential programs
• Concurrency does not make the worst-case complexity any worse
• Why is verification of concurrent programs considered more difficult?
Undecidable problem!

P satisfies S
**Assertions:** Provide contracts to decompose problem into a collection of decidable problems
- pre-condition and post-condition for each procedure
- loop invariant for each loop

**Abstractions:** Provide an abstraction of the program for which verification is decidable
- Finite-state systems
  - finite automata
- Infinite-state systems
  - pushdown automata, counter automata, timed automata, Petri nets
Assertions: an example

ensures result >= 0 ==> a[result] == n
ensures result < 0  ==> forall j:int :: 0 <= j && j < a.length ==> a[j] != n

int find(int a[ ], int n) {
    int i = 0;

    while (i < a.length) {
        if (a[i] == n) return i;
        i++;
    }

    return -1;
}
Assertions: an example

ensures \( \text{result} \geq 0 \implies a[\text{result}] = n \)
ensures \( \text{result} < 0 \implies \forall j: \text{int} :: 0 \leq j && j < \text{a.length} \implies a[j] \neq n \)

```c
int find(int a[], int n) {
    int i = 0;
    loop_invariant 0 \leq i && i \leq a.length
    loop_invariant \forall j: \text{int} :: 0 \leq j && j < i \implies a[j] \neq n
    while (i < a.length) {
        if (a[i] == n) return i;
        i++;
    }
    return -1;
}
```
{true}
i = 0
{0 <= i && i <= a.length && forall j:int :: 0 <= j && j < i ==> a[j] != n}

{0 <= i && i <= a.length && forall j:int :: 0 <= j && j < i ==> a[j] != n}
assume i < a.length;
assume !(a[i] == n);
i++;
{0 <= i && i <= a.length && forall j:int :: 0 <= j && j < i ==> a[j] != n}

{0 <= i && i <= a.length && forall j:int :: 0 <= j && j < i ==> a[j] != n}
assume i < a.length;
assume (a[i] == n);
{(i >= 0 ==> a[i] == n) && (i < 0 ==> forall j:int :: 0 <= j && j < a.length ==> a[j] != n)}

{0 <= i && i <= a.length && forall j:int :: 0 <= j && j < i ==> a[j] != n}
assume !(i < a.length);
{(-1 >= 0 ==> a[-1] == n) && (-1 < 0 ==> forall j:int :: 0 <= j && j < a.length ==> a[j] != n)}
Abstractions: an example

requires m == UNLOCK
void compute(int n) {
    for (int i = 0; i < n; i++) {
        Acquire(m);
        while (x != y) {
            Release(m);
            Sleep(1);
            Acquire(m);
        }
        y = f(x);
        Release(m);
    }
}

requires m == UNLOCK
void compute(int n) {
    for (int i = 0; i < n; i++) {
        assert m == UNLOCK;
        m = LOCK;
        while (x != y) {
            assert m == LOCK;
            m = UNLOCK;
            Sleep(1);
            assert m == UNLOCK;
            m = LOCK;
        }
        y = f(x);
        assert m == LOCK;
        m = UNLOCK;
    }
}
Abstractions: an example

requires m == UNLOCK
void compute(int n) {
    for (int i = 0; i < n; i++) {
        assert m == UNLOCK;
        m = LOCK;
        while (x != y) {
            assert m == LOCK;
            m = UNLOCK;
            Sleep(1);
            assert m == UNLOCK;
            m = LOCK;
        }
        y = f(x);
        assert m == LOCK;
        m = UNLOCK;
    }
    y = f(x);
    assert m == LOCK;
    m = UNLOCK;
}

requires m == UNLOCK
void compute( ) {
    for ( ; * ; ) {
        assert m == UNLOCK;
        m = LOCK;
        while (*) {
            assert m == LOCK;
            m = UNLOCK;
            assert m == UNLOCK;
            m = LOCK;
        }
        assert m == LOCK;
        m = UNLOCK;
    }
    assert m == LOCK;
    m = UNLOCK;
}
Interference

```plaintext
pre  x = 0;

int t;

x := t;

post  x = 1;  Correct
```
Interference

pre \ x = 0;

A

\begin{align*}
\text{int t;} \\
& t := x; \\
& t := t + 1; \\
& x := t;
\end{align*}

post \ x = 2;

B

\begin{align*}
\text{int t;} \\
& t := x; \\
& t := t + 1; \\
& x := t;
\end{align*}

Incorrect!
Interference

pre  \( x = 0; \)
post  \( x = 2; \)

\[
\begin{align*}
A & \quad B \\
\text{int t;} & \quad \text{int t;} \\
\text{acquire(l);} & \quad \text{acquire(l);} \\
t := x; & \quad t := x; \\
t := t + 1; & \quad t := t + 1; \\
x := t; & \quad x := t; \\
\text{release(l);} & \quad \text{release(l);} \\
\end{align*}
\]
Compositional verification using assertions
Invariants

• Program: a statement $S_p$ for each control location $p$
• Assertions: a predicate $\varphi_p$ for each control location $p$
• Sequential correctness
  – If $p$ is a control location and $q$ is a successor of $p$, then
    $\{\varphi_p\} S_p \{\varphi_q\}$ is valid
• Non-interference
  – If $p$ and $q$ are control locations in different threads, then
    $\{\varphi_p \land \varphi_q\} S_q \{\varphi_p\}$ is valid
pre  \( x = 0; \)

\[
\begin{align*}
\text{L0:} &\quad \text{acquire}(l); \\
\text{L1:} &\quad t := x; \\
\text{L2:} &\quad t := t + 1; \\
\text{L3:} &\quad x := t; \\
\text{L4:} &\quad \text{release}(l); \\
\text{L5:} &
\end{align*}
\]

post  \( x = 2; \)

\[
\begin{align*}
\text{M0:} &\quad \text{acquire}(l); \\
\text{M1:} &\quad t := x; \\
\text{M2:} &\quad t := t + 1; \\
\text{M3:} &\quad x := t; \\
\text{M4:} &\quad \text{release}(l); \\
\text{M5:} &
\end{align*}
\]
Two other checks

• precondition $\Rightarrow (\varphi_{L0} \land \varphi_{M0})$
  
  $$(A@L0 \land B@M0 \land x == 0) \Rightarrow \left[ (B@M0 \Rightarrow x=0) \land (B@M5 \Rightarrow x=1) \land (A@L0 \Rightarrow x=0) \land (A@L5 \Rightarrow x=1) \right]$$

• $(\varphi_{L0} \land \varphi_{M0}) \Rightarrow$ postcondition
  
  $$\left[ A@L5 \land B@M5 \land (B@M0 \Rightarrow x=1) \land (B@M5 \Rightarrow x=2) \land (A@L0 \Rightarrow x=1) \land (A@L5 \Rightarrow x=2) \right] \Rightarrow x == 2$$
Annotation explosion!

• For sequential programs
  – assertion for each loop
  – assertion refers only to variables in scope

• For concurrent programs
  – assertion for each control location
  – assertion may need to refer to private state of other threads
Verification by analyzing abstractions
State-transition system

- Multithreaded program
  - Set of global states $G$
  - Set of local states $L_1, \ldots, L_n$
  - Set of initial states $I \subseteq G \times L_1 \times \ldots \times L_n$
  - Transition relations $T_1, \ldots, T_n$
    - $T_i \subseteq (G \times L_i) \times (G \times L_i)$
  - Set of error states $E \subseteq G \times L_1 \times \ldots \times L_n$

\[(g_1, a_1, b_1) \xrightarrow{T_1} (g_2, a_2, b_1) \xrightarrow{T_2} (g_3, a_2, b_2)\]
Example

• $G = \{ (x, l) \mid x \in \{0,1,2\}, l \in \{\text{LOCK}, \text{UNLOCK}\} \}$

• $L_1 = \{ (t_1, pc_1) \mid t_1 \in \{0,1,2\}, pc_1 \in \{L0,L1,L2,L3,L4,L5\} \}$

• $L_2 = \{ (t_2, pc_2) \mid t_2 \in \{0,1,2\}, pc_2 \in \{M0,M1,M2,M3,M4,M5\} \}$

• $I = \{ (0, \text{UNLOCK}), (0, L0), (0, M0) \}$

• $E = \{ (x, l), (t_1, pc_1), (t_2, pc_2) \mid x \neq 2 \land pc_1 = L5 \land pc_2 = M5 \}$
Reachability problem

• Does there exist an execution from a state in I to a state in E?
Reachability analysis

\[ F = I \]
\[ S = \{ \} \]

while \((F \neq \emptyset)\) {
  remove \(s\) from \(F\)
  if \((s \in S)\) continue
  if \((s \in E)\)
    return YES
  for every thread \(t:\)
    add every \(t\)-successor of \(s\) to \(F\)
  add \(s\) to \(S\)
}

return NO

Space complexity: \(O(|G| \times |L|^n)\)
Time complexity: \(O(n \times |G| \times |L|^n)\)
Reachability problem

- Does there exist an execution from a state in $I$ to a state in $E$?
- PSPACE-complete
  - Little hope of polynomial-time solution in the general case
Challenge

- State space increases exponentially with the number of interacting components
- Utilize structure of concurrent systems to solve the reachability problem for programs with large state space
Tackling annotation explosion

• New specification primitive
  – “guarded_by lock” for data variables
  – “atomic” for code blocks

• Two layered proof
  – analyze synchronization to ensure that each atomic annotation is correct
  – do proof with assertions assuming atomic blocks execute without interruption
Bank account

```c
Critical_Section l;
/*# guarded_by l */
int balance;

/*# atomic */
void deposit (int x) {
    acquire(l);
    int r = balance;
    balance = r + x;
    release(l);
}

/*# atomic */
int read( ) {
    int r;
    acquire(l);
    r = balance;
    release(l);
    return r;
}

/*# atomic */
void withdraw(int x) {
    acquire(l);
    int r = balance;
    balance = r - x;
    release(l);
}
```
Definition of atomicity

Serialized execution of deposit

- x → y → acq(l) → r=bal → bal=r+n → rel(l) → z

Non-serialized executions of deposit

- acq(l) → x → r=bal → y → bal=r+n → z → rel(l)
- acq(l) → x → y → r=bal → bal=r+n → z → rel(l)

- deposit is atomic if for every non-serialized execution, there is a serialized execution with the same behavior
Reduction
Four atomicities

\textbf{R}: right commutes
  - lock acquire

\textbf{L}: left commutes
  - lock release

\textbf{B}: both right + left commutes
  - variable access holding lock

\textbf{A}: atomic action, non-commuting
  - access unprotected variable
Sequential composition

Use atomicities to perform reduction

Theorem: Sequence \((R+B)^*; (A+\varepsilon); (L+B)^*\) is atomic

<table>
<thead>
<tr>
<th>;</th>
<th>B</th>
<th>L</th>
<th>R</th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>B</td>
<td>L</td>
<td>R</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>R</td>
<td>R</td>
<td>A</td>
<td>R</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>L</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

Diagram:

- \(S_0\) to \(S_0\) via \(R^*\) and \(x\)
- \(S_0\) to \(S_5\) via \(R^*\) and \(x\)
- \(S_5\) to \(S_5\) via \(A\)
- \(S_5\) to \(S_5\) via \(L^*\)
- \(S_5\) to \(S_5\) via \(Y\)
- \(S_5\) to \(S_5\) via \(R; B; A; L\)
- \(S_5\) to \(S_5\) via \(R; A; L\)
- \(S_5\) to \(A\)
- \(A\) to \(A\)
- \(A\) to \(C\)
Bank account

Critical_Section l;
/*# guarded_by l */
int balance;

/*# atomic */
void deposit (int x) {
    R acquire(l);
    B int r = balance;
    B balance = r + x;
    L release(l);
}

/*# atomic */
int read( ) {
    int r;
    R acquire(l);
    R r = balance;
    L release(l);
    B return r;
}

/*# atomic */
void withdraw(int x) {
    R acquire(l);
    B int r = balance;
    B balance = r - x;
    L release(l);
}

Correct!
Bank account

Critical_Section l;
/*#guarded_by l */
int balance;

/*# atomic */
void deposit (int x) {
R  acquire(l);
B  int r = balance;
B  balance = r + x;
L  release(l);
}

/*# atomic */
int read( ) {
int r;
R  acquire(l);
B  r = balance;
L  release(l);
B  return r;
}

/*# atomic */
void withdraw(int x) {
A  int r = read();
R  acquire(l);
A  int r = read();
B  r = balance;
A  int r = read();
L  release(l);
B  release(l);
B  balance = r – x;
L  release(l);
}

Incorrect!
pre \ x = 0;

\begin{align*}
\text{A} \\
/*# atomic */
\begin{align*}
\text{int t;} \\
\text{acquire(l);} \\
\text{t := x;} \\
\text{t := t + 1;} \\
\text{x := t;} \\
\text{release(l);} \\
\end{align*}
\end{align*}

\begin{align*}
\text{B} \\
/*# atomic */
\begin{align*}
\text{int t;} \\
\text{acquire(l);} \\
\text{t := x;} \\
\text{t := t + 1;} \\
\text{x := t;} \\
\text{release(l);} \\
\end{align*}
\end{align*}

post \ x = 2;
pre  \ x = 0;

\begin{align*}
\text{A} & \quad \text{B} \\
\text{int } \ t; & \quad \text{int } \ t; \\
\text{L0: } \{ & \quad \text{M0: } \{ \\
& \quad \text{acquire(l);} \quad \text{acquire(l);} \\
& \quad \ t := x; \quad \ t := x; \\
& \quad \ t := t + 1; \quad \ t := t + 1; \\
& \quad \ x := t; \quad \ x := t; \\
& \quad \text{release(l);} \quad \text{release(l);} \\
\} & \quad \} \\
\text{L5:} & \quad \text{M5:} \\
\end{align*}

post  \ x = 2;
Tackling state explosion

• Symbolic model checking
• Bounded model checking
• Context-bounded verification
• Partial-order reduction
Recapitulation

• Multithreaded program
  – Set of global states $G$
  – Set of local states $L_1, \ldots, L_n$
  – Set of initial states $I \subseteq G \times L_1 \times \ldots \times L_n$
  – Transition relations $T_1, \ldots, T_n$
    • $T_i \subseteq (G \times L_i) \times (G \times L_i)$
  – Set of error states $E \subseteq G \times L_1 \times \ldots \times L_n$

• $T = T_1 \cup \ldots \cup T_n$

• Reachability problem: Is there an execution from a state in $I$ to a state in $E$?
Symbolic model checking

• Symbolic domain
  – universal set $U$
  – represent sets $S \subseteq U$ and relations $R \subseteq U \times U$
  – compute $\cup$, $\cap$, $\subseteq$, $\setminus$, etc.
  – compute $\text{post}(S, R) = \{ s | \exists s' \in S. (s', s) \in R \}$
  – compute $\text{pre}(S, R) = \{ s | \exists s' \in S. (s, s') \in R \}$
Post({a,b}, R) = {b, c}
Pre({a, c}, R) = {a, b, d}

R = {(a,c), (b,b), (b,c), (d, a)}
Boolean logic as symbolic domain

- \( U = G \times L_1 \times \ldots \times L_n \)
- Represent any set \( S \subseteq U \) using a formula over \( \log |G| + \log |L_1| + \ldots + \log |L_n| \) Boolean variables
- \( R_i = \{ ((g,l_1,\ldots,l_i,\ldots,l_n), (g',l_1,\ldots,l_i',\ldots,l_n)) \mid ((g,l_i), (g',l_i')) \in T_i \} \)
- Represent \( R_i \) using \( 2 \times (\log |G| + \log |L_1| + \ldots + \log |L_n|) \) Boolean variables
- \( R = R_1 \lor \ldots \lor R_n \)
Forward symbolic model checking

\[ S := \emptyset \]

\[ S' := I \]

while \( S \subseteq S' \) do {
    if \( (S' \cap E) \neq \emptyset \)
        return YES
    
    \[ S := S' \]

    \[ S' := S \cup \text{post}(S, R) \]
}

return NO
Backward symbolic model checking

S := ∅
S' := E

while S ⊆ S' do {
    if (S' ∩ I ≠ ∅)
        return YES
    S := S'
    S' := S ∪ pre(S, R)
}

return NO
Symbolic model checking

• Symbolic domain
  – represent a set $S$ and a relation $R$
  – compute $\cup$, $\cap$, $\subseteq$, etc.
  – compute $\text{post}(S, R) = \{ s \mid \exists s' \in S. (s', s) \in R \}$
  – compute $\text{pre}(S, R) = \{ s \mid \exists s' \in S. (s, s') \in R \}$

• Often possible to compactly represent large number of states
  – Binary decision diagrams
- Vertex represents decision
- Follow green (dashed) line for value 0
- Follow red (solid) line for value 1
- Function value determined by leaf value
- Along each path, variables occur in the variable order
- Along each path, a variable occurs exactly once
(Reduced Ordered) Binary Decision Diagram

1. Identify isomorphic subtrees (this gives a dag)
2. Eliminate nodes with identical left and right successors
3. Eliminate redundant tests

For a given boolean formula and variable order, the result is unique.
(The choice of variable order may make an exponential difference!)
Reduction rule #1

Merge equivalent leaves
Reduction rule #2

Merge isomorphic nodes

Before:

After:

- For the top part, nodes x and y are merged.
- For the bottom part, nodes x_1 and x_2 are merged.
Reduction rule #3

Eliminate redundant tests
• Canonical representation of Boolean function
• For given variable ordering, two functions equivalent if and only if their graphs are isomorphic
• Test in linear time

Initial graph

Reduced graph

\[(x_1 \lor x_2) \land x_3\]
Examples

Constants

0: Unique unsatisfiable function
1: Unique tautology

Variable

Treat variable as function

Typical function

- \((x_1 \lor x_2) \land x_4\)
- No vertex labeled \(x_3\)
- Independent of \(x_3\)
- Many subgraphs shared

Odd parity

Linear representation
Effect of variable ordering

\[(a_1 \land b_1) \lor (a_2 \land b_2) \lor (a_3 \land b_3)\]

Good ordering

Bad ordering

Linear growth

Exponential growth
Bit-serial computer analogy

- **Operation**
  - Read inputs in sequence; produce 0 or 1 as function value.
  - Store information about previous inputs to correctly deduce function value from remaining inputs.

- **Relation to BDD Size**
  - Processor requires $K$ bits of memory at step $i$.
  - BDD has $\sim 2^K$ branches crossing level $i$. 

```
\begin{array}{cccc}
  x_n & \ldots & x_2 & x_1 \\
  0 & \ldots & 0 & 0 \\
\end{array}
```
\( (a_1 \land b_1) \lor (a_2 \land b_2) \lor (a_3 \land b_3) \)

**Good ordering**

\[
\begin{align*}
& a_1 \\
& b_1 \\
& a_2 \\
& b_2 \\
& a_3 \\
& b_3 \\
& 0 \\
& 1
\end{align*}
\]

\( K = 2 \)

**Bad ordering**

\[
\begin{align*}
& a_1 \\
& b_1 \\
& a_2 \\
& b_2 \\
& a_3 \\
& b_3 \\
& 0 \\
& 1
\end{align*}
\]

\( K = n \)
Lower bound for multiplication (Bryant 1991)

- Integer multiplier circuit
  - $n$-bit input words $A$ and $B$
  - $2n$-bit output word $P$
- Boolean function
  - Middle bit ($n-1$) of product
- Complexity
  - Exponential BDD for all possible variable orderings

Actual Numbers
- 40,563,945 BDD nodes to represent all outputs of 16-bit multiplier
- Grows 2.86x per bit of word size
BDD operations \(\neg, \land, \lor, \exists, \forall\)

BDD node

- BDD manager maintains a directed acyclic graph of BDD nodes
- `ite(x,a,b)` returns
  - if \(a = b\), then \(a\) (or \(b\))
  - if \(a \neq b\), then a node with variable \(x\), left child \(a\), and right child \(b\)

n.var = \(x\)
n.false = \(a\)
n.true = \(b\)
and(a,b)

if (a = false ∨ b = false) return false
if (a = true) return b
if (b = true) return a
if (a = b) return a
if (a.var < b.var)
    return ite(a.var, and(a.false,b), and(a.true,b))
if (b.var < a.var)
    return ite(b.var, and(a,b.false), and(a,b.true))
// a.var = b.var
return ite(a.var, and(a.false,b.false), and(a.true,b.true))

Complexity: O(|a| × |b|)
not(a)

if (a = true) return false
if (a = false) return true
return ite(a.var, not(a.false), not(a.true))

Complexity: \(O(|a|)\)
cofactor(a, x, p)

if (x < a.var) return a
if (x > a.var)
    return ite(a.var, cofactor(a.false, x, p),
               cofactor(a.true, x, p))

// x = a.var
if (p)
    return a.true
else
    return a.false

Complexity: O(|a|)
substitute(a,x,y)

Assumptions
- a is independent of y
- x and y are adjacent in variable order

if (a = true \lor a = false) return a
if (a.var > x) return a
if (a.var < x)
    return ite(a.var, substitute(a.false,x,y),
                substitute(a.true,x,y))
if (a.var = x) return ite(y,a.false,a.true)
Derived operations

\( \text{or}(a,b) \equiv \neg(\text{and}(\neg(a), \neg(b))) \)

\( \text{exists}(a,x) \equiv \text{or}(\text{cofactor}(a,x,\text{false}), \text{cofactor}(a,x,\text{true})) \)

\( \text{forall}(a,x) \equiv \text{and}(\text{cofactor}(a,x,\text{false}), \text{cofactor}(a,x,\text{true})) \)

\( \text{implies}(a,b) \equiv (\text{or}(\neg(a),b) = \text{true}) \)

\( \text{iff}(a,b) \equiv (a = b) \)
Tools that use BDDs

- Finite-state machine
  - SMV, VIS, ...
- Pushdown machine
  - Bebop, Moped, ...
- ...

Applying symbolic model checking is difficult

- Symbolic representation for successive approximations to the reachable set of states may explode
  - hardware circuits, cache-coherence protocols, etc.
Bounded model checking

Problem: Is there an execution from a state in \( I \) to a state in \( E \) of length at most \( d \)?

```text
for each \( (s \in I) \) {
    Explore(s, d)
}
exit(NO)

Explore(s, x) {
    if \( (s \in E) \) exit(YES)
    if \( (x == 0) \) return
    for each thread \( t \):
        for each \( t\)-successor \( s' \) of \( s \):
            Explore(s’, x-1)
}
```

Space complexity: \( O(d) \)
Time complexity: \( O(n^d) \)
NP-complete
Symbolic bounded model checking

• Construct Boolean formula $\varphi(d)$
  
  $$
  \neg I(s_0) \land R(s_0, s_1) \land \ldots \land R(s_{d-1}, s_d) \land (E(s_0) \lor \ldots \lor E(s_d))
  $$

• $\varphi$ is satisfiable iff there is an execution from a state in $I$ to a state in $E$ of length at most $d$

• Set $k$ to 0

• Check satisfiability of $\varphi(k)$

• If unsatisfiable, increment $k$ and iterate
Symbolic bounded model checking

• Pros
  – Eliminates existential quantification and state caching
  – Leverage years of research on SAT solvers
  – Finds shallow bugs (low k)

• Cons
  – Difficult to prove absence of bugs
  – May not help in finding deep bugs (large k)
• Many subtle concurrency errors are manifested in executions with few context switches
• Analyze all executions with few context switches
• Unbounded computation within each context
  – Different from bounded model checking
Context-bounded reachability problem

• An execution is c-bounded if every thread has at most c contexts
• Does there exist a c-bounded execution from a state in I to a state in E?
Context-bounded reachability (I)

A work item \((s, p)\) indicates that from state \(s\), \(p(i)\) contexts of thread \(i\) remain to be explored.

for each thread \(t\) {
    ComputeRTClosure\((t)\)
}
for each \((s \in I)\) {
    Explore\((s, \lambda t. c)\)
}
exit(NO)

Explore\((s, p)\) {
    if \((s \in E)\) exit(YES)
    for each \(u\) such that \((p(u) > 0)\) {
        for each \(s'\) such that \((s, s') \in RT(u)\)
            Explore\((s', p[u := p(u) - 1])\)
    }
}

ComputeRTClosure\((u)\) {
    \(F = \{(s, s) | s \in G \times L1 \times \ldots \times Ln\}\)
    while \((F \neq \emptyset)\) {
        Remove \((s, s')\) from \(F\)
        for each \(u\)-successor \(s''\) of \(s'\)
            add \((s, s'')\) to \(F\)
        Add \((s, s')\) to \(RT(u)\)
    }
}
for each \( s \in I \) {
    Explore(s, \lambda t. c)
}
exit(NO)

Explore(s, p) {
    if \((s \in E)\) exit(YES)
    for each \( u \) such that \((p(u) > 0)\) {
        ComputeRTClosure(u, s)
        for each \( s' \) such that \((s, s') \in RT(u)\)
            Explore(s', p[u := p(u)-1])
    }
}

ComputeRTClosure(u, s) {
    if \((s, s) \in RT(u)\) return
    F = \{(s, s)\}
    while \((F \neq \emptyset)\) {
        Remove \((s, s')\) from F
        for each \( u\)-successor \( s'' \) of \( s'\)
            add \((s, s'')\) to \( F\)
        Add \((s, s')\) to \( RT(u)\)
    }
}
Complexity analysis

Space complexity: \( n \times (|G| \times |L|)^2 \)

Time complexity: \((nc)!/(c!)^n \times |I| \times (|G| \times |L|)^{nc}\)

Although space complexity becomes polynomial, time complexity is still exponential.
Context-bounded reachability is NP-complete

Membership in NP: Witness is an initial state and nc sequences each of length at most \(|G \times L|\)

NP-hardness: Reduction from the CIRCUIT-SAT problem
Complexity of safety verification

<table>
<thead>
<tr>
<th></th>
<th>Unbounded</th>
<th>Context-bounded</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite-state systems</td>
<td>PSPACE complete</td>
<td>NP-complete</td>
</tr>
<tr>
<td>Pushdown systems</td>
<td>Undecidable</td>
<td>NP-complete</td>
</tr>
</tbody>
</table>

P = # of program locations  
G = # of global states  
n = # of threads  
c = # of contexts
Applying symbolic model checking is difficult

• Symbolic representation for successive approximations to the reachable set of states may explode
  – hardware circuits, cache-coherence protocols, etc.

• State might be complicated
  – finding suitable symbolic representation might be difficult
  – stacks, heap, queues, etc.
Enumerative model checking

• A demonic scheduler
  – Start from the initial state
  – Execute some enabled transition repeatedly

• Systematically explore choices
  – breadth-first, depth-first, etc.
Stateful vs. Stateless

- **Stateful**: capture and cache visited states
  - an optimization on terminating programs
  - required for termination on non-terminating programs

- **Stateless**: avoid capturing state
  - capturing relevant state might be difficult
  - depth-bounding for termination on non-terminating programs
Simplifying assumption

• **Multithreaded program**
  – Set of global states $G$
  – Set of local states $L_1, …, L_n$
  – Unique initial state $\text{init} \in G \times L_1 \times … \times L_n$
  – Partial transition functions $T_1, …, T_n$
    • $T_i : (G \times L_i) \rightarrow (G \times L_i)$
  – Set of error states $E \subseteq G \times L_1 \times … \times L_n$

• $T = T_1 \cup … \cup T_n$

• State transition graph defined by $T$ is acyclic

• Reachability problem: Is there an execution from $\text{init}$ to a state in $E$?
Observations

- A path is a sequence of thread ids
- Some paths are executable and called executions
- An execution results in a unique final state
- A multithreaded program with acyclic transition graph has a finite number of executions
T1
acquire(m)
x := 1
release(m)
acquire(n)
y := 1
release(n)

T2
acquire(m)
x := 2
release(m)
acquire(n)

• T1, T2 is not an execution

• T1, T1, T1, T2, T2, T2 is an execution

• T1, T1, T1, T2, T2, T2, T1, T1, T1 is a terminating execution
A simple stateless search algorithm

Explore(ε)
exit(NO)

Explore(e) {
    if (e ∈ E) exit(YES)
    for every thread t enabled in e {
        Explore(e•t)
    }
}
Tools

• Stateful: SPIN, Murphi, Java Pathfinder, CMC, Bogor, Zing, …

• Stateless: Verisoft, CHESS
CHESS: Systematic testing for concurrency

Tester Provides a Test Scenario

CHESS runs the scenario in a loop
• Each run is a different interleaving
• Each run is repeatable

Program
TestScenario()

While(not done){
  TestScenario()
  ...
}

Win32 API

Kernel:
  Threads, Scheduler,
  Synchronization Objects
State-space explosion

- Number of executions
  \[ = O(n^{nk}) \]
- Exponential in both \( n \) and \( k \)
  - Typically: \( n < 10 \quad k > 100 \)
- Limits scalability to large programs

Goal: Scale CHESS to large programs (large \( k \))
Preemption-bounding

- Prioritize executions with small number of preemptions
- Two kinds of context switches:
  - Preemptions – forced by the scheduler
    - e.g. Time-slice expiration
  - Non-preemptions – a thread voluntarily yields
    - e.g. Blocking on an unavailable lock, thread end

```
x = 1;
if (p != 0) {
p = 0;
    p = p->f;
    x = p->f;
}
```

Thread 1

```
p = 0;
```

Thread 2

preemption

non-preemption
Preemption-bounding in CHESS

• The scheduler has a budget of \( c \) preemptions
  – Nondeterministically choose the preemption points
• Resort to non-preemptive scheduling after \( c \) preemptions
• Once all executions explored with \( c \) preemptions
  – Try with \( c+1 \) preemptions
Property 1: Polynomial bound

- Terminating program with fixed inputs and deterministic threads
  - n threads, k steps each, c preemptions
- Number of executions $\leq \binom{n}{k} \cdot \binom{n+c}{c} \cdot (n+c)!$
  $= O\left( (n^2k)^c \cdot n! \right)$

Exponential in n and c, but not in k

- Choose c preemption points
- Permute n+c atomic blocks
Property 2: Simple error traces

• Finds smallest number of preemptions to the error

• Number of preemptions better metric of error complexity than execution length
Property 3: Coverage metric

- If search terminates with preemption-bound of c, then any remaining error must require at least c+1 preemptions.

- Intuitive estimate for:
  - The complexity of the bugs remaining in the program
  - The chance of their occurrence in practice
Property 4: Many bugs with few preemptions

<table>
<thead>
<tr>
<th>Program</th>
<th>KLOC</th>
<th>Threads</th>
<th>Preemptions</th>
<th>Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work-Stealing Queue</td>
<td>1.3</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>CDS</td>
<td>6.2</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>CCR</td>
<td>9.3</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>ConcRT</td>
<td>16.5</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Dryad</td>
<td>18.1</td>
<td>25</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>APE</td>
<td>18.9</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>STM</td>
<td>20.2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>PLINQ</td>
<td>23.8</td>
<td>8</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>TPL</td>
<td>24.1</td>
<td>8</td>
<td>2</td>
<td>9</td>
</tr>
</tbody>
</table>
Coverage vs. Preemption-bound

The graph shows the percentage of state space covered as a function of context bound for various models:

- File System Model
- Bluetooth
- Transaction Manager
- Work Stealing Queue
Dryad (coverage vs. time)

![Graph showing the number of states covered over time for different execution numbers and methods. The graph plots # Executions (x1000) on the x-axis and # States Covered on the y-axis. The methods compared are icb, idfs-125, dfs, idfs-100, and idfs-75.](image)

Partial-order reduction

- Commuting actions
  - accesses to thread-local variables
  - accesses to different shared variables
- Useful in both stateless and stateful search

Goal: Explore only one interleaving of commuting actions by different threads
Happens-before graph

T1
acquire(m)
x := 1
release(m)
acquire(n)
y := 1
release(n)

T2
acquire(m)
x := 2
release(m)

T1, T1, T1, T2, T2, T2, T1, T1, T1
Happens-before graph

T1
acquire(m)  
x := 1  
release(m)
acquire(n)  
y := 1  
release(n)

T2
acquire(m)  
x := 2  
release(m)

T1: 
acq(m)  
T1: 
x:=1  
T1: 
rel(m)  

T2: 
acq(m)  
T2: 
x:=2  
T2: 
rel(m)
Happens-before graph

- An execution is a **total order** over thread actions
- The happens-before graph of an execution is a **partial order** over thread actions
  - All linearizations of a happens-before graph are executions and generate the same final state
- The happens-before graph is a canonical representative of an execution

T1: acq(m) → T1: x:=1 → T1: rel(m) → T1: acq(n) → T1: y:=1 → T1: rel(n)

T2: acq(m) → T2: x:=2 → T2: rel(m)
A partial-order reduction algorithm

Theorem: The algorithm explores exactly one linearization of each happens-before graph
Eliminating the HB-cache

• Order on thread ids: t1 < … < tn
• Extend < to the dictionary order on executions
• Rep(e) ≡ \min \{e' \mid \text{HB}(e) = \text{HB}(e')\}

Explore(e) { 
    if (e ∈ E) exit(YES) 
    for every thread t enabled in e { 
        if (Rep(e•t) < e•t) continue 
        Explore(e•t) 
    } 
}
Checking $\text{Rep}(e) < e$

- Straightforward
  - compute $\text{Rep}(e)$ (linear time)
  - check $\text{Rep}(e) < e$ (linear time)

- Incremental
  - We know that $\text{Rep}(e) < e$
  - What about $\text{Rep}(e \cdot t) < e \cdot t$?
Checking $\text{Rep}(e) < e$

$e \cdot t$

$t_1 \circ \ldots \circ t_p \circ t_{p+1} \circ \ldots \circ t_q \circ t$

$t_p$ is the latest action with a happens-before edge to $t$

$\text{Rep}(e \cdot t) < e \cdot t$

iff

$t < t_k$ for some $k$ in $(p, q]$
Comments

• Algorithm equivalent to the “sleep sets” algorithm
• Particularly useful for stateless search
  – converts the search tree into a search graph