Verifying Concurrent Programs
(Tutorial)

Aarti Gupta
Systems Analysis & Verification Group
NEC Labs America, Princeton, USA
Acknowledgements

- Malay Ganai, Vineet Kahlon, Nishant Sinha*, Chao Wang* (NEC Labs)
- Akash Lal (MSR, India)
- Madanlal Musuvathi (MSR)
- Antoine Miné (CNRS)
- Kedar Namjoshi (Alcatel-Lucent)
- Andrey Rybalchenko, Ashutosh Gupta, Corneliu Poppea (TU Munich), Alexander Malkis (Imdea)
- Arnab Sinha (Princeton University)
- Tayssir Touili (LIAFA)
- Thomas Wahl (Northeastern University)
Motivation

- **Key Computing Trends**
  - **Mobile**
  - **Server**
  - **Gaming**
  - **High Performance, Low Power**

- **Parallel/Multi-threaded Programming**
  - Difficult to get right
    - Dependencies due to shared data
    - Subtle effects of synchronizations
    - Often manually parallelized
  - Difficult to debug
    - too many interleavings of threads
    - hard to reproduce bugs

- Single core solutions don’t work
- Multi-core platforms
- Need parallel, multi-threaded programming
- Distributed, networked systems
What will I (try to) cover?

- **Basic elements**
  - Model of concurrency
    - Asynchronous interleaving model (unlike synchronous hardware)
    - Explosion in interleavings
  - Synchronization & Communication (S&C)
    - Shared variables: between threads or shared memory for processes
    - Locks, semaphores: for critical sections, producer/consumer scenarios
    - Atomic blocks: for expressing atomicity (non-interference)
    - Pair-wise rendezvous
    - Asynchronous rendezvous
    - Broadcast: one-to-many communication
  - On top of other features of sequential programs
    - Recursive procedures, Loops, Heaps, Pointers, Objects, …
    - (Orthogonal concerns and techniques)

- **Will cover Static and Dynamic verification techniques**
What I will not be able to cover

- Active topics of research (but out of scope here)
  - Parallel programs: Message-passing (e.g. MPI libraries), HPC applications
  - Synthesis/Optimization of locks/synchronizations for performance
  - Memory models: Relaxed memory models (e.g. TSO), Transactional memories
  - Object-based verification: Linearizability checking
  - Concurrent data structures/libraries: Lock-free structures
  - Separation logic: pointers & heaps, local reasoning
  - Theorem-proving, type analysis, runtime monitoring …
Models for Verifying Concurrent Programs

- **Finite state systems**
  - Asynchronous composition, S&C (including buffers/channels for messages), but no recursion
  - Setting: Inline procedures up to some bound to get finite models
  - Techniques: Bounded analysis (e.g. dynamic analysis, BMC)

- **Sequential programs**
  - Recursive procedures and other features, but no S&C and no interleavings
  - Setting: Add support for S&C and interleavings (thread interference)
  - Techniques: Bounded as well as unbounded analysis

- **Pushdown system models**
  - Stack of a pushdown system (PDS) models recursion, finite control, data is finite or infinite (with abstractions)
  - Setting: Consider interacting PDSs with various S&C
  - Techniques: PDS-based model checking
Outline

- Introduction

  - PDS-based Model Checking
    - Theoretical results

- Static Verification Methods
  - Reduction: Partial order reduction, Symmetry
  - Bounding: Context-bounded analysis, Memory Consistency-based analysis
  - Program Abstraction: Static analysis, Thread-modular reasoning

- Dynamic Verification Methods
  - Preemptive context bounding
  - Predictive analysis
  - Coverage-guided systematic testing

- Conclusions
Each thread is modeled as a PDS:
- Finite Control: models control flow in a thread (data is abstracted)
- Stack: models recursion, i.e., function calls and returns

PDS Example:
States: \{s, t, u, v\}
Stack Symbols: \{A, B, C, D\}
Transition Rules:
\[
\begin{align*}
<s,A> &\rightarrow <t, e > \\
<s,A> &\rightarrow <t, B > \\
<s,A> &\rightarrow <t, C B >
\end{align*}
\]

If the state is s, and A is the symbol at the top of the stack, then transit to state t, pop A, and push B, C on the stack.
PDS-based Model Checking

- Close relationship between Data Flow Analysis for sequential programs and the model checking problem for Pushdown Systems (PDS)
  - The set of configurations satisfying a given property is regular
  - Has been applied to verification of sequential Boolean programs
    [Bouajjani et al., Walukeiwicz, Esparza et al.]

- Analogous to the sequential case, dataflow analysis for concurrent program reduces to the model checking problem for interacting PDSs

- Problems of Interest: To study multi-PDSs interacting via the standard synchronization primitives
  - Locks
  - Pairwise and Asynchronous Rendezvous
  - Broadcasts
Problem: For multi-PDS systems, the set of configurations satisfying a given property is not regular, in general

Strategy: exploit the situations where PDSs are loosely coupled

Key Challenge
Capture interaction based on synchronization patterns
Capturing Interaction in presence of Synchronizations

- Key primitive: **Static Reachability**
  - A global control state $t$ is *statically reachable* from state $s$ if there exists a computation from $s$ to $t$ that respects the constraints imposed by synchronization primitives, e.g., locks, wait/notifies, ...

- However, static reachability is undecidable
  - for pairwise rendezvous
  - for arbitrary lock accesses
  - Undecidability hinges on a close interaction between synchronization and recursion
  - (Note: Even for finite data abstractions)

- Strategies to get around this undecidability
  - Special cases of programming patterns: Nested Locks, Bounded Lock Chains
  - Place restrictions on synchronization and communication (S&C)
Nested Locks:
Along every computation, each thread can only release that lock which it acquired last, and that has not yet been released

Example:
```java
f( ) {
    g( ){
        acquire(b);
        acquire(a);
        acquire(c);
        g( );
        release(a);
        release(b);
        // h ( );
        release(b);
        release(c);
        acquire(c);
    }
    h( ){
        acquire(c);
        release(b);
    }
    acquire(c);
}
```

f calls g: nested locks
f calls h: non-nested locks

Programming guidelines typically recommend that programmers use locks in a nested fashion

Multiple locks are enforced to be nested in Java_{1.4} and C#
Programming Pattern: Lock Chains

- **Lock Chains**

- **Nested Locks: Chains of length one**

- **Most lock usage is nested**
- **Non-nested usage occurs in niche applications, often bounded chains**
  - Serialization, e.g. 2-phase commit protocol uses chains of length 2
  - Interaction of mutexes with synchronization primitives like wait/notify
  - Traversal of shared data structures, e.g. length of a statically-allocated array
Key Challenge: Capture interaction based on synchronization patterns

General Problem for arbitrary lock patterns: Undecidable  [Kahlon et al. CAV 2005]

For nested locks and bounded lock chains: Decidable  [POPL 07, LICS 09, CONCUR 11]
- Tracks lock access patterns thread-locally as regular automata
- Incorporates a consistency check in the acceptance condition
Reachability is decidable for PDS Networks with:
- acyclic communication graph
- lossy FIFO channels

[Atig et al. 08]
PDS-based Model Checking: Summary

Reachability Problem

- Undecidable for Pairwise Rendezvous [Ramalingam 00]
- Undecidable for PDSs interacting via Locks [Kahlon et al. CAV 05]
- Decidable for PDSs interacting via Nested Locks [Kahlon et al. CAV 05]
- Decidable for PDSs interacting via Bounded Lock Chains [Kahlon LICS 09, CONCUR 11]

Reachability/Model Checking is Decidable under Other Restrictions

- Constrained Dynamic Pushdown Networks [Bouajjani et al. TACAS 07]
- Asynchronous Dynamic Pushdown Network [Bouajjani et al. FSTTCS 05]
- Reachability of Acyclic Networks of Pushdown Systems [Atig et al. CONCUR 08]
- Context-bounded analysis for concurrent programs with dynamic creation of threads [Atig et al. TACAS 09]
Hard to apply PDS-based methods directly
- Huge gap between model and modern programming languages

In addition to state space explosion due to data (as in finite state systems and sequential programs)
the complexity bottleneck is exhaustive exploration of interleavings

The next section describes various strategies to tackle this in practice
- Reduce number of interleavings to consider
  1. Partial Order Reduction (POR)
  2. Exploit symmetry
- Bound the problem
  3. Context-bounded analysis
  4. Memory Consistency-based analysis
- Use program abstractions and compositional techniques
  5. Static analysis
  6. Thread-modular reasoning
Some Preliminaries

- What is checked in practice?

- **Common concurrency bugs**
  - Dataraces, deadlocks, atomicity violations

- **Standard runtime bugs**
  - Null pointer dereferences
  - Memory safety bugs

- **Properties**
  - Safety, e.g. mutual exclusion
  - Liveness, e.g. absence of starvation
Common Concurrency Bugs

- **Race Condition**: simultaneous memory access (at least one write)
  
  ```
  /*----- Thread 1 -----*/
  ...
  Write (globalVar);
  ...
  /*----- Thread 2 -----*/
  ...
  Read (globalVar);
  ```

- **Deadlock**: hold-and-wait cycles
  
  ```
  /*----- Thread 1 -----*/
  lock(A);
  ...
  lock(B);
  /*----- Thread 2 -----*/
  lock(B);
  ...
  lock(A);
  ```

- **Atomicity violation**: e.g. a common three-access pattern
  
  ```
  /*----- Thread 1 -----*/
  if (account_ptr != NULL) {
    ...
    account_ptr -> amount -= debit;
  }
  /*----- Thread 2 -----*/
  if (account_ptr != NULL) {
    free(account_ptr);
    account_ptr = NULL;
  }
  ```
Data Race Detection

- Data Race: If two *conflicting* memory accesses happen *concurrently*

- Two memory accesses *conflict* if
  - They target the same location
  - They are not both read operations

- Data races may reveal synchronization errors
  - Typically caused because programmer forgot to take a lock
  - Many programmers tolerate “benign” races
  - Racy programs risk obscure failures caused by memory model relaxations in the hardware and the compiler
Data Race Detection

- Two popular approaches for datarace detection

  - **Lockset analysis**
    - Lockset: set of locks held at a program location
    - Method:
      - Compute locksets for all locations in a program (statically or dynamically)
      - Race: When there are conflicting accesses from program locations with disjoint locksets
    - Gives too many false warnings, since program locations may not be concurrent
      - Provides opportunity for more precise analysis (discussed later)

  - **Happens-Before (HB) analysis**
    - Happens-Before order: a partial order over synchronization events
      - Method:
        - Observe HB order during dynamic execution
        - Race: If conflicting accesses are not ordered by HB
    - This is precise, but dynamic executions have limited coverage
      - Provides opportunity for improving coverage over alternate schedules (discussed later)

[Savage et al. 97, ERASER]
Happens-Before Order

- Use logical clocks and timestamps to define a partial order called *happens-before* on events in a concurrent system.

- States precisely when two events are *logically* concurrent (abstracts away real time).

- Cross-edges from send events to receive events.

- \((a_1, a_2, a_3)\) happens before \((b_1, b_2, b_3)\) iff \(a_1 \leq b_1\) and \(a_2 \leq b_2\) and \(a_3 \leq b_3\).

- Distributed Systems: Cross-edges from send to receive events.

- Shared Memory Systems: Cross-edges represent *ordering effects of synchronization*.
  - Edges from lock release to subsequent lock acquire.
  - Long list of primitives that may create edges: Semaphores, Waithandles, Rendezvous, System calls (asynchronous IO).
Consider the following thread executions.

**Thread 1**
- $x=1$
- $g=g+2$

**Thread 2**
- $y=1$
- $g=g*2$

The full-blown state-space can be large.  
**Good news:** the order of independent events does not affect the state that is reached.
Consider the following thread executions.

**Thread 1**
- \( x = 1 \)
- \( g = g + 2 \)

**Thread 2**
- \( y = 1 \)
- \( g = g \ast 2 \)

The full-blown state-space can be large.

**Good news:** *the order of independent events does not affect the state that is reached.*

Different orders of independent events constitute an equivalence class (Mazurkiewicz trace equivalence).

**It suffices to explore only one representative from each equivalence class.**
Consider the following thread executions.

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=1</td>
<td>y=1</td>
</tr>
<tr>
<td>g=g+2</td>
<td>g=g*2</td>
</tr>
</tbody>
</table>

The full-blown state-space can be large.

**Good news:** the order of independent events does not affect the state that is reached.

Different orders of independent events constitute an equivalence class (Mazurkiewicz trace equivalence).

**It suffices to explore only one representative from each equivalence class.**
POR in Model Checking

- POR in explicit-state model checking / stateless search
  - Persistent sets, stubborn sets, sleep sets
    - [Godefroid 1996], [Peled 1993], [Valmari 1990], …
  - Dynamic POR (uses HB to derive precise conflict sets), Cartesian POR
    - [Flanagan & Godefroid, POPL 2005], [Gueta et al, SPIN 2007]

- POR in Software Model Checkers
  - SPIN [Holzmann], VeriSoft [Godefroid], JPF [Visser et al., Stoller et al.]
    - Pioneering efforts on model checking concurrent programs

- POR in symbolic model checking / bounded model checking
  - In BDD based model checking
    - [Alur et al, 2001], [Theobald et al, 2003],…
  - In SAT/SMT based BMC
    - [Cook, Kroening, Sharygina, 2005],
    - [Grumberg, Lerda, Strichman, Theobald, 2005],
    - [Kahlon et al. 2006], [Wang et al. 2008], [Kahlon et al. 2009]
2. Exploiting Symmetry in MC

There is a lot of redundancy in Replicated Systems

Many reachable states are “symmetric”, i.e.

\( (N_1, T_2, N_3, T_4) \quad (T_1, T_2, N_3, N_4) \quad (T_1, N_2, N_3, T_4) \)

**Counter Abstraction:** collapse the above into

\[ \langle \#N : 2, \#T : 2, \#C : 0 \rangle \]
Local States and Counter Abstraction

Local state explosion
Combating Local State Explosion in Symbolic Exploration

Solution: omit zero-valued counters from global states!

\[ \rightarrow \text{abstract state space down from } n^{2^v} \text{ to } n^{\min\{n,2^v\}} \]

Huge improvement if \( n \ll 2^v \)!

Symbolic Counter Abstraction: \( n \) threads, \( v \) local variables

[Basler et al. CAV 09]
Computing Reachable Program States
3. Context-Bounded Analysis

- Recall
  - The general problem of verifying a concurrent program (recursive procedures with synchronization) is undecidable.
  - We have seen various strategies to get around undecidability
    - Exploiting patterns of synchronization
    - Restricting synchronization & communication
    - Ignoring recursion by (bounded) function inlining

- Another key idea: Bound number of context switches
  - Context-bounded analysis of PDSs is decidable [Qadeer & Rehof, TACAS 05]
  - Note: There can be recursion within each segment between context switches
  - In practice, many bugs are found within a small number of context switches
  - Implemented in tools: KISS, CHESS (Microsoft), ...
Sequentialization: Reduce CBA to sequential program analysis

- Efficient reduction:
  - $P_s$ has $K$ times more global variables
  - No increase in local variables
- Can borrow all the cool stuff from the sequential world
Sequentialization: From Concurrent to Sequential

- $K =$ number of chances that each thread gets
- Guess $(K-1)$ global states: $s_1 = \text{init}, s_2, \ldots, s_K$

$T_1$ processes all contexts first, guesses states of $T_2$
$T_2$ goes next, using states of $T_1$
At the end: Check the guesses, i.e. $s''_1 = s_2$ and $s''_2 = s_3$, ...
Sequentialization: From Concurrent to Sequential

- Pushes “guesses” about interleaved states into inputs

- $T_1 \rightarrow T_1^s$ and $T_2 \rightarrow T_2^s$

- $(T_1 \parallel T_2) \rightarrow (T_1^s; T_2^s; \text{Checker}; \text{assert(no\_error)})$

Main idea:
Reduce control non-determinism to data non-determinism
4. Memory Consistency-based Analysis

- Interleaving model
  - Partially ordered traces
  - Context-switching, interleaved traces
  - Is control-centric: Control induces data-flow

- Instead, consider a Memory Consistency (MC) model
  - e.g. Sequential Consistency (SC), Total Store Order (TSO), ....
  - MC model specifies rules under which a read may observe some write

- Data Nondeterminism in MC model
  - Reason about read-write interference directly
  - No need to have a scheduler!
  - Is data-centric: data-flow induces control-flow
  - Examples: Nemos, Checkfence, x86-TSO, Memsat, Staged Analysis
  - Symbolic exploration using SAT/SMT solvers avoids explicit enumeration of interleavings
Sequential Consistency (SC) based Verification

- Three steps
  - Obtain an Interference Skeleton (IS) from (unrolled) Program
    - Global read and write events and their program order
    - Encoded as $\Phi_{IS}$
  - SC axioms for reads/writes in IS
    - Quantified first-order logic formula $\Pi$
  - Encode Property as a formula $\Phi_P$
    - data race, assertion violation, ...
  - Check $\Phi_{IS} \land \Pi \land \Phi_P$ for satisfiability (using an SMT solver)

[Sinha & Wang POPL 11]
Sequential Consistency Axioms

- **Axioms of Sequential Consistency (SC)**
  - each read must observe (link with) some write
  - read must link with most recent write in execution order

- **Specified in typed first-order logic**
  - read $r$, write $w$: Access type

- **Link** Predicate: $\text{link} (r,w)$
  - holds if read $r$ observes write $w$ in an execution
  - Exclusive: $\text{link} (r,w) \Rightarrow \forall w'. \neg \text{link} (r,w')$

- **Must-Happen-before** Predicate: $\text{hb} (w,r)$
  - $w$ must happen before $r$ in the execution
  - strict partial order

- These axioms are added to the Program **precisely encoded** using reads/writes and program order
Example

**Goal:** Detect NULL pointer access violation

- so \( rp \) must be enabled
- \( \text{en}(\text{rp}) = (\text{en}(\text{rc}) \land \text{val}(\text{rc}) = \text{true}) \)

\[
\text{en}(\text{rp}) \implies \text{en}(\text{rc}) \quad \text{(Path conditions)}
\]

and, \( \text{en}(\text{rp}) \implies \text{val}(\text{rc}) = \text{true} \) \hspace{1cm} (*

Because \( \text{en}(\text{rp}) \), so \( \text{link}(\text{rp},\text{wp}) \) \hspace{1cm} (Π)

So, \( \text{hb}(\text{wp},\text{rp}) \) \hspace{1cm} (Π)

\[
\text{link}(\text{rc},\text{wc}_1) \lor \text{link}(\text{rc},\text{wc}_2) \quad \text{(Π)}
\]

Try \( \text{link}(\text{rc},\text{wc}_1) \)

so, \( \text{val}(\text{rc}) = \text{val}(%545099) = \text{false} \) \hspace{1cm} (Π)

Contradicts with (*)

so, \( \text{link}(\text{rc},\text{wc}_2) \)

so, \( \text{hb}(\text{wc}_2,\text{rc}) \) \hspace{1cm} (Π)

Check \( (Π) \) for \( \text{rc} \): intruding write \( \text{wc}_1 \)

so, Add \( \text{hb}(\text{wc}_1,\text{wc}_2) \)

linearize to obtain a feasible trace
5. Motivating Example for Static Analysis

Consider all possible pairs of locations where shared variables are accessed (e.g. for checking data races)
Motivating Example: Lockset Analysis

void Alloc_Page ( ) {
    a = c;
    pt_lock(&plk);
    if (pg_count >= LIMIT) {
        pt_wait (&pg_lim, &plk);
        incr (pg_count);
        pt_unlock(&plk);
        sh1 = sh;
    } else {
        pt_lock (&count_lock);
        pt_unlock (&plk);
        page = alloc_page();
        sh = 5;
        if (page)
            incr (pg_count);
        pt_unlock(&count_lock);
    } else {
        pt_lock (&count_lock);
        pt_unlock (&plk);
        page = alloc_page();
        sh = 5;
        if (page)
            incr (pg_count);
        pt_unlock(&count_lock);
    }
    b = a+1;
}

void Dealloc_Page ( )
    pt_lock(&plk);
    if (pg_count == LIMIT) {
        sh = 2;
        decre (pg_count);
        sh1 = sh;
        pt_notify (&pg_lim, &plk);
        pt_unlock(&plk);
    } else {
        pt_lock (&count_lock);
        pt_unlock (&plk);
        decre (pg_count);
        sh = 4;
        pt_unlock(&count_lock);
        end-if
    }

Lockset Analysis: Compute the set of locks at location / 
Here, lock \( plk \) is held in both locations. 
Hence, these locations are simultaneously unreachable. 
Therefore, there is no datarace.
void Alloc_Page ( ) {
    a = c;
    pt_lock(&plk);
    if (pg_count >= LIMIT) {
        pt_wait (&pg_lim, &plk);
        incr (pg_count);
        pt_unlock(&plk);
    } else {
        pt_lock (&count_lock);
        pt_unlock (&plk);
        page = alloc_page();
        sh = 5;
        if (page)
            incr (pg_count);
        pt_unlock(&count_lock);
    }
    b = a+1;
}

void Dealloc_Page ( ) {
    pt_lock(&plk);
    if (pg_count == LIMIT) {
        sh = 2;
        incr (pg_count);
        sh1 = sh;
        pt_notify (&pg_lim, &plk);
        pt_unlock(&plk);
    } else {
        pt_lock (&count_lock);
        pt_unlock (&plk);
        page = alloc_page();
        sh = 5;
        if (page)
            incr (pg_count);
        pt_unlock(&count_lock);
    }
}

Motivating Example: Synchronization Constraints

These locations are simultaneously unreachable due to wait-notify ordering constraint. Therefore, no data race.
Motivating Example

void Alloc_Page() {
    a = c;
    pt_lock(&plk);
    if (pg_count >= LIMIT) {
        pt_wait(&pg_lim, &plk);
        incr(pg_count);
        pt_unlock(&plk);
        sh1 = sh;
    } else {
        pt_lock(&count_lock);
        page = alloc_page();
        sh = 5;
        if (page)
            incr(pg_count);
        pt_unlock(&count_lock);
    }
    b = a + 1;
}

void Dealloc_Page() {
    pt_lock(&plk);
    if (pg_count == LIMIT) {
        sh = 2;
        decr(pg_count);
        sh1 = sh;
        pt_notify(&pg_lim, &plk);
        pt_unlock(&plk);
    } else {
        pt_lock(&count_lock);
        page = alloc_page();
        sh = 4;
        pt_unlock(&count_lock);
    }
}

Data race?

NO, due to invariants at these locations
pg_count is in (-inf, LIMIT) in T1
pg_count is in [LIMIT, +inf) in T2
Therefore, these locations are not simultaneously reachable

How do we get these invariants?
By using abstract interpretation, model checking, …
Intuitively, one can reason over a set of product control states
- Not all product (global) control states, but only the \textit{statically reachable} states
- Transaction Graph:
  - Each node is a \textit{statically reachable} global control state,
  - Each edge is a \textit{transaction}, i.e. an uninterruptible sequence of actions by a single thread

Two main (inter-related) problems
- How to find which global control states (nodes) are reachable?
- How to find (large) transactions?
  - Larger the transactions, smaller the number of interleavings to consider

Refinement Approach
- At any stage, the transaction graph \textit{over-approximates} the set of thread interleavings for sound static analysis or model checking
- \textit{Iteratively refine} the transaction graph by computing \textit{invariants}
repeat (forever) {
    lock(posLock);
    while (pos > SLOTS) {
        unlock(posLock);
        wait(full);
        lock(posLock);
    }
    data[pos++] := ...;
    if (pos > 0) {
        signal(emp);
    }
    unlock(posLock);
}
Refining Transactions

[Kahlon, Sankaranarayanan & Gupta, TACAS 09]

- **Initial Transaction Graph**
  - Make this as small as possible
  - Use static partial order reduction (POR) to consider non-redundant interleavings
    - Over control states only, but need to consider CFL-reachability
  - Use synchronization constraints to eliminate statically unreachable nodes
    - Recall: Static reachability wrt synchronization operations
    - Precise analysis for nested locks, bounded lock chains, locks with wait-notify
    [Kahlon et al. 05, Kahlon 08, Kahlon & Wang 10]

- **Iterative Refinement of Transaction Graph**
  
  \textit{Repeat}
  
  - Compute invariants over the transaction graph using abstract interpretation
    - Abstract domains: range, octagons, polyhedra [Cousot & Cousot, Miné. …]
  
  - Use invariants to prove nodes unreachable, and simplify graph
  
  - Re-compute transactions (POR, synchronization analysis)
  
  \textit{Until} transactions cannot be refined further.
Application: Detection of Data Races

- Implemented in the CoBe (Concurrency Bench) tool

- Phase 1: Static Warning Generation
  - Shared variable detection, Lockset analysis
  - Generate warnings at global control states (c1, c2) when
    - The same shared variable is accessed, at least one access is a write, and
    - Locksets at c1 and c2 are disjoint

- Phase 2: Static Warning Reduction (for improved precision)
  - Create a Transaction Graph, and generate sound invariants
    - POR reductions, synchronization analysis, abstract interpretation
  - If (c1, c2) is proved unreachable, then eliminate the warning

- Phase 3: Model Checking
  - Otherwise, create a model for model checking reachability of (c1, c2)
    - Slicing, constant propagation, enforcing invariants: lead to smaller models
    - Makes bounded model checking viable
    - Provides a concrete error trace
## CoBe: Experiments

### Linux device drivers with known data race bugs

<table>
<thead>
<tr>
<th>Linux Driver</th>
<th>KLOC</th>
<th>#Sh Vars</th>
<th>#Warnings</th>
<th>Time (sec)</th>
<th># After Invariants</th>
<th>Time (sec)</th>
<th>#Witness</th>
<th>#Unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>pci_gart</td>
<td>0.6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>jfs_dmap</td>
<td>0.9</td>
<td>6</td>
<td>13</td>
<td>2</td>
<td>1</td>
<td>52</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>hugetlb</td>
<td>1.2</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ctrace</td>
<td>1.4</td>
<td>19</td>
<td>58</td>
<td>7</td>
<td>3</td>
<td>143</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>autosfs_expire</td>
<td>8.3</td>
<td>7</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>12</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>ptrace</td>
<td>15.4</td>
<td>3</td>
<td>1</td>
<td>15</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>raid</td>
<td>17.2</td>
<td>6</td>
<td>13</td>
<td>2</td>
<td>6</td>
<td>75</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>tty_io</td>
<td>17.8</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>11</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>ipoib_multicast</td>
<td>26.1</td>
<td>10</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>16</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>decoder</td>
<td>2.9</td>
<td>4</td>
<td>256</td>
<td>5min</td>
<td>15</td>
<td>22min</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bzip2smp</td>
<td>6.4</td>
<td>25</td>
<td>15</td>
<td>18</td>
<td>12</td>
<td>35</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **After Phase 1 (Warning Generation)**
- **After Phase 2 (Warning Reduction)**
- **After Phase 3 (Model Checking)**
Static Analysis: Propagating Interference Effects

Example analysis

abstract consumer/producer

\[ \varepsilon_0 : X = 1 \]

### p1:
- while 1 do
  - lock(m);\(^1\)
  - if \( X > 0 \) then \( X \leftarrow X - 1; \)
  - unlock(m);
- \( Y = X \)

### p2:
- while 1 do
  - lock(m);
  - if \( X > 10 \) then \( X \leftarrow 10; \)
  - unlock(m)

- at \(^1\) the `unlock - lock` effect from `p2` imports \( \{X\} \times [1, 10] \)
- at \(^2\) \( X \in [1, 10] \), no effect from `p2`: \( X \leftarrow X - 1 \) is safe
- at \(^3\) \( X \in [0, 9] \), and \( p2 \) has the effects \( \{X\} \times [1, 10] \)
  - so, \( Y \in [0, 10] \)
Preliminary results

**Target code:**
- embedded avionic code
- reactive code, network code, list, string, message management
- ARINC 653 real-time operating system (2.6 Klines C model)

**Analysis results (Intel 64-bit 2.66 GHz server)**

<table>
<thead>
<tr>
<th>lines</th>
<th># threads</th>
<th># iters.</th>
<th>time</th>
<th># alarms</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 K</td>
<td>5</td>
<td>3</td>
<td>1h</td>
<td>80</td>
</tr>
<tr>
<td>1.6 M</td>
<td>15</td>
<td>4</td>
<td>21h</td>
<td>6747</td>
</tr>
</tbody>
</table>

efficiency on par with analyses of synchronous code
- few thread reanalyses (up to 4)
- few partitions (up to 4 for environments, 52 for interferences)
  fits in 32 GB of memory

but still too many alarms (99.6% selectivity, but not zero alarm)
6. Thread-Modular Reasoning

- As we just saw, invariants play a key role in static analysis
- **Compositional verification**
  - *Proofs rules* typically use **inductive invariants**
  - Advantage: Avoids explicit reasoning over interleavings

- **Some Basics**

  - An *assertion* is a set of states
  - Assertion $\phi$ is *invariant* if it includes all reachable states
  - Invariance is proved using an auxiliary *inductive* invariant
    - (initiality) $[I \Rightarrow \theta]$
    - (inductiveness) $[\text{next}(T, \theta) \Rightarrow \theta]$
    - (adequacy) $[\theta \Rightarrow \phi]$
  - $\text{next}(T, \xi)$ is the set of successors of states in $\xi$ (by $T$)
  - $R$ is the strongest inductive invariant
Localized Inductive Invariants

Idea: build an inductive invariant out of "little" pieces

- Restrict $\theta$ to the shape $\theta_1(X, L_1) \land \theta_2(X, L_2) \land \ldots \land \theta_N(X, L_N)$
- $X$ is the set of (globally) shared program variables (e.g., locks)
- $L_i$ is the set of variables local to process $P_i$ (e.g., program counter, stack, temporary variables)
- The shape inherently limits correlations between local variables of different components (e.g., $(x > l_1)$ is OK but not $(l_1 + l_2 > l_3)$)
Localized Inductive Invariants = Compositional Proof

Inductiveness for a localized assertion turns into the rely-guarantee form

- (initiality) For all $i$: $[l_i \Rightarrow \theta_i]$
- (inductiveness) For all $i$: $[\text{next}(T_i, \theta_i) \Rightarrow \theta_i]$
- (non-interference) For all $i, j : j \neq i$:
  $[\text{next}(\text{intf}_j^\theta \land \text{unchanged}(L_i), \theta_i) \Rightarrow \theta_i]$
- The effect of process $P_j$ on the shared state is called interference, represented by $\text{intf}_j^\theta(X, X') = (\exists L_j, L_j' : T_j \land \theta_j)$

(We’ll call a localized inductive invariant a “split invariant”.)
Computing the Strongest Split Invariant

The Knaster-Tarski Theorem also gives a simple iterative scheme to compute the fixpoint.

1. Set the initial vector $\theta^0 = (false, false, \ldots, false)$
2. At stage $i$, compute $\theta^{i+1} = F(\theta^i)$
3. Stop when a fixpoint is reached (no change in any component)

**Theorem: Complexity**

This algorithm takes time polynomial in $N$ and in the size $L$ (the number of states) of each component. The complexity is (roughly) $O(N^3 L^3)$.

---

**Local Proofs** [Cohen & Namjoshi CAV 07, CAV 08, CAV 10]

- Can handle safety and liveness properties
- Works well on many examples (Bakery, Peterson’s, Szymanski, …)
Automation and experimental comparison of “Owicki-Gries” and “Rely-Guarantee” reasoning

- Applicable to arbitrary (ad-hoc) synchronization patterns (not only nested locking or datarace-free code)
- Analyze implicitly an unbounded number of context switches (not restricted to context-bounded switching)
- Handles non-thread-modular proofs (not restricted to thread-modular, global-only, assumptions)

http://www.model.in.tum.de/~popeea/research/threader.html

Uses well-known techniques from software model checking (predicate abstraction refinement, CEGAR) for automating the proof rules
Given a transition system: \((\varphi_{init}, \rho_1 \lor \cdots \lor \rho_N, \varphi_{err})\)

Find auxiliary assertions \(R_1, \ldots, R_N\) (reachable states) and \(E_1, \ldots, E_N\) (environment transitions) such that:

\[
\begin{align*}
\varphi_{init} & \rightarrow R_i & \text{for } i \in 1..N \\
R_i \land (\rho_i \lor E_i \land \bar{\rho_i}) & \rightarrow R'_i & \text{for } i \in 1..N \\
\land_{i=1..N} R_i \land \varphi_{err} & \rightarrow \text{false} \\
(\lor_{j=1..N\setminus\{i\}} R_j \land \rho_j) & \rightarrow E_i & \text{for } i \in 1..N
\end{align*}
\]
Outline

✓ Introduction

✓ PDS-based Model Checking
  ✓ Theoretical results

✓ Static Verification
  ✓ Reductions: Partial order reduction, Counter-based abstraction
  ✓ Bounding: Context-bounded analysis, Memory Consistency-based analysis
  ✓ Abstraction: Unbounded context analysis, Thread-modular reasoning

➢ Dynamic Analysis Methods
  – Preemptive context bounding
  – Predictive analysis
  – Coverage-guided systematic testing

☐ Conclusions

PDS-based model checking, Static Verification may not scale to large programs

Interest in Dynamic Analysis based on executions
User expectation:
If the program fails the given test, the user wants to see the bug

The reality:
Even if the program may fail (under a certain schedule), the user likely won’t see it

Why?
Thread scheduling is controlled by the OS and the Pthreads library

Tools: VeriSoft, Chess, Fusion, Inspect
Take control of the scheduler to execute alternate schedules
State space explosion in all interleaved executions

- **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)
CHESS: Preemptive Context Bounding (PCB)

- Terminating program with fixed inputs and deterministic threads
  - n threads, k steps each, c preemptions
  - Preemptions are context switches forced by the scheduler

- Number of executions \( \leq {n^k \choose c} \cdot (n+c)! \)
  \[ = O((n^2 k)^c \cdot n!) \]

Exponential in n and c, but not in k

- Choose c preemption points
- Permute n+c atomic blocks

Many bugs found in a small number of preemptions

[Musuvathi et al. PLDI 07, OSDI 08]
Motivation: Trace Based Verification

Full formal verification is often intractable

Alternate approaches

Collect shared access footprint

Online/offline monitoring of trace

Predict errors in alternate interleavings

Larger set of interleavings is explored.
Atomicity Violations

- Atomicity is a desired correctness criterion for concurrent programs.
  - Non-interference on shared accesses from code residing outside and inside an atomic region.
  - Serializability is a notion that checks atomicity.

- A recent study shows 69% of concurrency bugs due to atomicity violations [Lu et al. ASPLOS’08]
Predictive Analysis (based on traces)

- **Predictive analysis** [Rosu et al. CAV 07, Farzan et al. TACAS 09, … ]
  - Run a test execution and log information about events of interest
  - Generate a *predictive* model over the events, by relaxing some ordering constraints
  - Analyze the predictive model to check alternate interleavings of these events
  - Note: Does not cover events not observed in the trace

- **Symbolic Predictive Analysis** [Wang et al. FM 09, TACAS 10]
  - Generate a *precise* predictive model by considering constraints due to synchronization and dataflow
    - No false bugs
  - Symbolically explore all possible thread interleavings of events in that trace, *using an SMT solver*
    - Performs better than explicit enumeration
C program: multi-threaded, using Pthreads

```
int x = 0;
int y = 0;
pthread_t t1, t2;
main() {
    t1 = pthread_create(t1, foo);
    t2 = pthread_create(t2, bar);
    pthread_join(t2);
    pthread_join(t1);
    assert(x != y);
}
```

```
foo() {
    int a;
    t11: a = y;
    t12: if (a == 0) {
        t13: x = 1;
        t14: a = x + 1;
        t15: x = a;
    } else
    t16: x = 0;
    t17: x = 0;
    t18: }
```

```
bar() {
    int b;
    t21: b = x;
    t22: if (b == 0) {
        t23: y = 1;
        t24: b = y + 1;
        t25: y = b;
    } else
    t26: x = 0;
    t27: y = 0;
    t28: }
```

Execution trace

Symbolic Predictive Model

“assume( c )” means the (c)-branch is taken
Symbolic Predictive Analysis for Detection of Violations

- Build a SAT formula (in some quantifier-free first-order logic)
  - \( F_{\text{program}} \): a feasible thread interleaving of CTP
  - \( F_{\text{property}} \): e.g. an assertion is violated

- Solve using an SMT solver
  \(( F_{\text{program}} \& \& F_{\text{property}} )\)
  - Sat \(\rightarrow\) found a real error
  - Unsat \(\rightarrow\) no error in any interleaving

- Improves precision over other predictive techniques, while providing coverage over all possible interleavings over the observed events.
What is the root cause of a “concurrency bug”?  
- Programmers often make, but fail to enforce, some implicit assumptions regarding the concurrency control of the program
  - Certain blocks should be mutually exclusive → data race
  - Certain blocks should be executed atomically → atomicity violation
  - Certain operations should be executed in a fixed order → order violation

To chase “concurrency bugs”, we would like to go after the “broken assumptions”…
- Exhaustively test all concurrency control scenarios
- But not all possible thread interleavings
Coverage-Guided Systematic Testing

- Coverage Metric:
  HaPSet (History-aware Predecessor Set)

- How do we use this metric?
  - Use a framework for systematically generating interleavings
    - e.g. stateless model checking
  - Keep track of HaPSets covered so far
  - Instead of DPOR/PCB, use HaPSet to prune away interleavings
  - Idea: Don’t generate an interleaving to test if the “concurrency control scenario” (HaPSet) has already been covered

- Based on PSet (Predecessor Set)
  - Psets were used for enforcing safe executions

  Jie Yu, Satish Narayanasamy
PSet (Predecessor Set) [Yu & Narayanasamy ISCA 09]

Psets are tracked for statements in code, not for events

PSet (statement): the set of immediately dependent “remote” statements

\[
\begin{align*}
P\text{Set}(W_1) &= \{\} \\
P\text{Set}(R_1) &= \{\} \\
P\text{Set}(R_2) &= \{W_1\} \\
P\text{Set}(R_3) &= \{W_1\} \\
P\text{Set}(R_4) &= \{\} \\
P\text{Set}(W_2) &= \{R_3, R_4\} \\
P\text{Set}(W_3) &= \{W_2\}
\end{align*}
\]
1. Synchronization statements
   - PSet ignored synchronizations, e.g. lock/unlock, wait/notify
   - HaPSet considers synchronizations – essential for concurrency

2. Context & thread sensitivity
   - PSet (effectively) treats a statement as a (file, line) pair
   - HaPSet treats a “statement” as a tuple (file, line, thr, ctx), where
     • \( \text{thr} = \{\text{local\_thread}, \text{remote\_thread}\} \) (exploits symmetry)
     • \( \text{ctx} = \) the truncated calling context

[Wang et al. ICSE 2011]
Intuition: Why are HaPSets Useful?

Thread T1
...

\[
\begin{align*}
\text{e2} & \quad \{ \text{if } (p \neq 0) \} \\
\text{e3} & \quad *(p) = 10; \\
\end{align*}
\]

Thread T2
...

\[
\begin{align*}
\text{e1} & \quad \{ \quad p = &a; \\
\quad \} \\
\text{e4} & \quad \{ \quad p = 0; \\
\quad \} \\
\end{align*}
\]

Observations:

#1. In all good runs, HaPSet[e3] = { }

#2. In all good runs, e2 is not in HaPSet[e4]

From the given run

\[
\begin{align*}
\text{HaPSet(e1)} & = \{ \} \\
\text{HaPSet(e2)} & = \{ e1 \} \\
\text{HaPSet(e3)} & = \{ \} \\
\text{HaPSet(e4)} & = \{ e3 \} \\
\end{align*}
\]

From all good runs

\[
\begin{align*}
\text{HaPSet(e1)} & = \{ e2 \} \\
\text{HaPSet(e2)} & = \{ e1, e4 \} \\
\text{HaPSet(e3)} & = \{ \} \\
\text{HaPSet(e4)} & = \{ e3 \} \\
\end{align*}
\]

Need only 2 test runs to capture all “good” runs
Why are HaPSets Useful?

Thread T1
... 
{ 
  if (p != 0) 
  { 
    *(p) = 10; 
  } 
}

Thread T2
... 
{ 
  p = &a; 
}
... 
{ 
  p = 0; 
}

Observations:
#1. In all good runs, HaPSet[e3] = {}
#2. In all good runs, e2 is not in HaPSet[e4]

Steer search directly to a “bad” run

From the given run
HaPSet(e1) = {}
HaPSet(e2) = {e1}
HaPSet(e3) = {}
HaPSet(e4) = {e3}

From all good runs
HaPSet(e1) = {e2}
HaPSet(e2) = {e1,e4}
HaPSet(e3) = {}
HaPSet(e4) = {e3}

From all (good and bad) runs
HaPSet(e1) = {e2}
HaPSet(e2) = {e1,e4}
HaPSet(e3) = {e4}
HaPSet(e4) = {e3,e2}
Thrift is a software framework by Facebook, for scalable cross-language services development.

The C++ library has **18.5K lines of C++ code**. It has a known **deadlock**.

<table>
<thead>
<tr>
<th>Test Program</th>
<th>LoC</th>
<th>thrsds</th>
<th>bug type</th>
<th>HaPSet runs</th>
<th>HaPSet time(s)</th>
<th>DPOR runs</th>
<th>DPOR time(s)</th>
<th>PCB0 runs</th>
<th>PCB0 time(s)</th>
<th>PCB1 runs</th>
<th>PCB1 time(s)</th>
<th>PCB2 runs</th>
<th>PCB2 time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lib-w2-5t</td>
<td>18.5k</td>
<td>3</td>
<td>deadlk</td>
<td>14</td>
<td>27.8</td>
<td>23</td>
<td>18.6</td>
<td>512(no)</td>
<td>247.2</td>
<td>26</td>
<td>29.2</td>
<td>215</td>
<td>146.9</td>
</tr>
<tr>
<td>lib-w3-5t</td>
<td>18.5k</td>
<td>4</td>
<td>deadlk</td>
<td>18</td>
<td>27.5</td>
<td>733</td>
<td>TO</td>
<td>1301</td>
<td>TO</td>
<td>399</td>
<td>229.7</td>
<td>876</td>
<td>TO</td>
</tr>
<tr>
<td>lib-w4-5t</td>
<td>18.5k</td>
<td>5</td>
<td>deadlk</td>
<td>22</td>
<td>33.7</td>
<td>665</td>
<td>TO</td>
<td>1111</td>
<td>TO</td>
<td>980</td>
<td>TO</td>
<td>677</td>
<td>TO</td>
</tr>
<tr>
<td>lib-w5-5t</td>
<td>18.5k</td>
<td>6</td>
<td>deadlk</td>
<td>25</td>
<td>38.1</td>
<td>572</td>
<td>TO</td>
<td>899</td>
<td>TO</td>
<td>670</td>
<td>TO</td>
<td>582</td>
<td>TO</td>
</tr>
</tbody>
</table>

**Much faster than DPOR or PCB**

Did not miss bugs in practice (many other examples in paper)
Summary and Challenges

- **Verifying Concurrent Programs**
  - Concurrency is pervasive, and very difficult to verify
  - Active area of research
    - Model checking, Static analysis, Testing/dynamic verification, …
    - Precise analysis requires reasoning about synchronization
      - Exploit programming patterns that are amenable for precise analysis
    - Efficient analysis requires controlling complexity of interleavings
      - Reductions, Implicit search, Abstractions, Compositional proofs
  - Precision AND efficiency of analysis are needed for practical impact
    - Applications guided by practical concerns
      - Context-bounding, Coverage-directed testing
    - Advancements in Decision Procedures (SAT/SMT) offer hope

- **Hierarchy of Practical Challenges**
  - Multi-core systems, Many-core systems
  - Distributed systems

- Great opportunity due to continuing growth of networked multi-core systems