Dynamic Partial-Order Reduction for Model Checking Software

Patrice Godefroid
Bell Labs

Cormac Flanagan
UC Santa Cruz

Presented by:
Ulrich Müller
Model Checking

- Given a multithreaded program
- We'd like to check for deadlocks and safety property violations
  - Proove it!
  - Ideally in a push-button fashion
  - Directly using source code (implementation vs abstraction)

- Idea:
  - Exhaustive state space exploration
  - Here:
    - Threads are subdivided into transitions
    - All transition interleavings over all threads
Naïve Algorithm

- Let’s do a depth-first search in the state space
- Pros:
  - Covers the whole state space, hence we have certainty of our software correctness
- Cons:
  - Requires a finite state-space
  - Requires an acyclic state-space
  - Can be immensely time-intensive due to the state-explosion problem
State-Explosion Problem

Given $n=3$ independent threads:

This is how the global state space looks:

Possible Interleavings: $n!$
(Distinct paths in state graph)

Number of states: $\sum_{i=0}^{n} \binom{n}{i} = 2^n$
Partial-Order Reduction

- The idea:
  - Use independence between transitions to reduce the state space

Two transitions are independent if both of the following hold:

- They neither disable nor enable each other and
- They commute
Partial-Order Reduction

- Since independent transitions commute, we can swap two adjacent ones in a given trace.

- Leads to an equivalence class of traces.
- We only need to check one of those traces per equivalence class!

Also called Model Checking using Representatives.
Partial-Order Reduction

- Why the name?
  - Natural representation of transitions in a concurrent system isn’t a trace (linear order) ...

  ... but a partial order

By using linear orders, we’re unnessecarily adding information to the relation!
Partial-Order Reduction

- In practice model checker only has a local view of the state graph.
- In a given state, it has to evaluate the minimal subset of enabled transitions to follow that still guarantees soundness.
Persistent Transitions

- An enabled transition $b$ in the state $s$ is a persistent transition, if it is independent with every transition $r_i$ reachable from $s$ without executing $b$.

- If a transition $b$ is persistent, it is sound to only explore $b$.

- Persistent sets: generalization to many threads.

Based on Flanagan’s presentation of the paper.
Static POR

- Use **static source code analysis** to determine the persistent set of a state **as soon as** it is reached

- **Problem**
  - Static analysis is **approximate** and doesn’t catch all independency
  - For example, whether two pointers refer to the same location is determined **conservatively**
    - This is where **dynamic partial order reduction** shines
Dynamic POR

- Using **runtime information**, resolving aliasing suddenly becomes easy

**Approach:**
- Execute an *arbitrary* trace to completion
- During this execution, take note of possible conflicts and mark them for later *backtracking*
- **Persistent set** is built *on-the-fly*
DPOR Algorithm

Summary:
- Analyzed 3 of the 6 possible traces
- Ideal POR would analyze 2
- Persistent sets constructed dynamically!
Summary

- **Pros**
  - No approximate/expensive/complicated static analysis
  - Supports pointer-rich data structures
  - Supports dynamic creation of threads/objects

- **Cons**
  - Finite state space
  - Acyclic state space
    - Can be extended to be stateful, but is memory expensive

- **Open questions**
  - Liveness properties and LTL?
Questions?
Implementations

Inspect: A Framework for Dynamic Verification of Multithreaded C Programs
Sadly, very sparsely documented.
http://www.cs.utah.edu/~yuyang/inspect/

Cute: A Concolic Unit Testing Engine for C and Java
Uses a simplified version of DPOR to guide symbolic model checking.
http://osl.cs.uiuc.edu/~ksen/cute/
DPOR Algorithm

```plaintext
proc explore(stack S)
    s = last(S);
    for all processes p {
        t_p = next(s,p)
        if (∃ latest transition S_i in S with (S_i,t_p) ∈ D for which
            S_i and t_p may be co-enabled and
            after S_i no dependent transitions from p have happened)
            add all processes to backtrack(pre(S,i)) which
            were necessary to enable t_p in this partial order
            or if there are no such processes, add all enabled processes
    }
    if (∃p ∈ enabled(s)) {
        backtrack(s) := {p}
        let done := Ø
        while (∃p ∈ (backtrack(s) \ done)) {
            add p to done
            explore(S.next(s,p))
        }
    }
}

Between S_i and s only transitions of p have happened that are independent of all other transitions between S_i and s.

More processes might be added by states ‘after’ s.

Every state s in S gets a set backtrack(s): the set of processes to backtrack on.
```

From a lecture of Theo Ruys on the subject
Indexer Benchmark

From Flanagan's presentation of the paper