Parallelizing Concuerror: A Dynamic Partial Order Reduction Testing Tool for Erlang Programs

Φυτάς Παναγιώτης

ΣΗΜΜΥ - ΕΜΠ

03112113

Summary

Background

Aim of Thesis

- Develop parallel version for source-DPOR algorithm.

Aim of Thesis

- Develop parallel version for source-DPOR algorithm.
- Develop parallel version for optimal-DPOR algorithm.

Aim of Thesis

- Develop parallel version for source-DPOR algorithm.
- Develop parallel version for optimal-DPOR algorithm.
- Implement those parallel algorithms at Concuerror.

Aim of Thesis

- Develop parallel version for source-DPOR algorithm.
- Develop parallel version for optimal-DPOR algorithm.
- Implement those parallel algorithms at Concuerror.
- Evaluate the performance of our implementation.

Background

Concurrent Computing

Concurrent Computing is a form of computing in which several computations are executed during overlapping time periods concurrently, instead of sequentially (one completing before the next starts).
Essential because:

Concurrent Computing

Concurrent Computing is a form of computing in which several computations are executed during overlapping time periods concurrently, instead of sequentially (one completing before the next starts).

Essential because:

- Multithreaded computer architectures

Concurrent Computing

Concurrent Computing is a form of computing in which several computations are executed during overlapping time periods concurrently, instead of sequentially (one completing before the next starts).
Essential because:

- Multithreaded computer architectures
- Availability of services

Concurrent Computing

Concurrent Computing is a form of computing in which several computations are executed during overlapping time periods concurrently, instead of sequentially (one completing before the next starts).
Essential because:

- Multithreaded computer architectures
- Availability of services
- Controllability

Concurrent Computing

However, concurrency is difficult to get right:

Concurrent Computing

However, concurrency is difficult to get right:

- Deadlocks

Concurrent Computing

However, concurrency is difficult to get right:

- Deadlocks
- Race conditions

Concurrent Computing

However, concurrency is difficult to get right:

- Deadlocks
- Race conditions
- Resource starvation

Concurrent Computing

However, concurrency is difficult to get right:

- Deadlocks
- Race conditions
- Resource starvation
- Scheduling non-determinism

Concurrent Computing

However, concurrency is difficult to get right:

- Deadlocks
- Race conditions
- Resource starvation
- Scheduling non-determinism
    - Interleaving non-determinism

Concurrent Computing

However, concurrency is difficult to get right:

- Deadlocks
- Race conditions
- Resource starvation
- Scheduling non-determinism
  - Interleaving non-determinism
  - Timing non-determinism

Concurrent Computing

However, concurrency is difficult to get right:

- Deadlocks
- Race conditions
- Resource starvation
- Scheduling non-determinism
    - Interleaving non-determinism
    - Timing non-determinism

Concurrent Computing

However, concurrency is difficult to get right:

- Deadlocks
- Race conditions
- Resource starvation
- Scheduling non-determinism
    - Interleaving non-determinism
    - Timing non-determinism

Errors can occur only on specific rare interleavings. Detecting and reproducing bugs becomes extremely hard (Heisenbugs).

Modeling our Problem

- An interleaving represents a scheduling of the concurrent program.

Modeling our Problem

- An interleaving represents a scheduling of the concurrent program.
- The state-space is the set of all possible interleavings.

Modeling our Problem

- An interleaving represents a scheduling of the concurrent program.
- The state-space is the set of all possible interleavings.
- In order to verify a program, the complete state-space must be explored.

Stateless Model Checking

- Stateless Model Checking systematically explores all possible interleavings.

Stateless Model Checking

- Stateless Model Checking systematically explores all possible interleavings.
- Combinatorial state-space explosion.

Stateless Model Checking

- Stateless Model Checking systematically explores all possible interleavings.
- Combinatorial state-space explosion.
- Different interleavings can be equivalent.

Stateless Model Checking

- Stateless Model Checking systematically explores all possible interleavings.
- Combinatorial state-space explosion.
- Different interleavings can be equivalent.

Stateless Model Checking

- Stateless Model Checking systematically explores all possible interleavings.
- Combinatorial state-space explosion.
- Different interleavings can be equivalent.

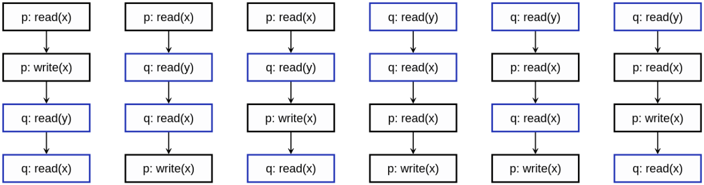

Figure: Stateless Model Checking Example

Partial Order Reduction

Partial Order Reduction tries to avoid exploring equivalent interleavings through race detection.

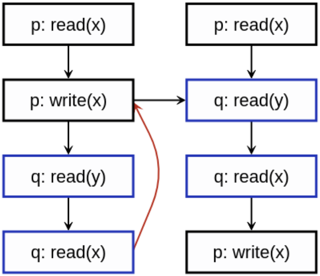

Figure: Partial Order Reduction Example

Partial Order Reduction

- Static Partial Order Reduction: Dependencies are tracked before execution, by statically analazing the code.

Partial Order Reduction

- Static Partial Order Reduction: Dependencies are tracked before execution, by statically analazing the code.
- Dynamic Partial Order Reduction (DPOR): Actual dependencies are observed during runtime.

General DPOR

DPOR: performs a DFS using a backtrack set. Exploration is based on two techniques:

- Persistent sets: only a provably sufficient subset of the enabled processes gets explored.

General DPOR

DPOR: performs a DFS using a backtrack set. Exploration is based on two techniques:

- Persistent sets: only a provably sufficient subset of the enabled processes gets explored.
- Sleep sets: contain processes, whose exploration would be redundant, preventing equivalent interleavings from being fully explored.

Important Concepts

- The complete execution of a process p splits into different execution steps, which are to be executed atomically. Those steps are referred to as $events$. Each event must be deterministic.

Important Concepts

- The complete execution of a process p splits into different execution steps, which are to be executed atomically. Those steps are referred to as $events$. Each event must be deterministic.
- An execution sequence E of a system is a finite sequence of execution steps of its processes that is performed from a unique initial state.

Important Concepts

- The complete execution of a process p splits into different execution steps, which are to be executed atomically. Those steps are referred to as $events$. Each event must be deterministic.
- An execution sequence E of a system is a finite sequence of execution steps of its processes that is performed from a unique initial state.
- We use $E \simeq E'$ to denote that E and E' are equivalent, and $[E]_{\simeq}$ to denote the equivalence class of E.

Source Sets

Definition 1 (Initials after an execution sequence $E.w$, $I_{[E]}(w)$)

$p \in I_{[E]}(w)$ if and only if there is a sequence w' such that $E.w \simeq E.p.w'$.

Definition 2 (Weak Initials after an execution sequence $E.w$, $WI_{[E]}(w)$)

$p \in WI_{[E]}(w)$ if and only if there are sequences w' and v such that $E.w.v \simeq E.p.w'$.

Source Sets

Definition 3 (Source Sets)

Let E be an execution sequence, and let W be a set of sequences, such that $E.w$ is an execution sequence for each $w \in W$. A set T of processes is a source set for W after E if for each $w \in W$ we have $WI_{[E]}(w) \cap T \neq \emptyset$.

Source-DPOR

Function *Explore*(E,$Sleep$)

```
if ∃p ∈ (enabled(s_[E]) \ Sleep) then
    backtrack(E) := p;
    while ∃p ∈ (backtrack(E) \ Sleep) do
        foreach e ∈ dom(E) such that e ≲_{E.p} next_[E](p) do
            let E' = pre(E, e);
            let u = notdep(e, E).p;
            if I_[E'](u) ∩ backtrack(E') = ∅ then
                add some q' ∈ I_[E'](u) to backtrack(E');
            end
        end
        let Sleep' := {q ∈ Sleep | E ⊨ p♢q};
        Explore(E.p, Sleep');
        add p to Sleep;
    end
end
```

Algorithm 1: Source-DPOR