Getting more out of a stateless model checker

Viktor Vafeiadis

MAX PLANCK INSTITUTE
FOR SOFTWARE SYSTEMS

2 June 2023

The grand challenge of software engineering

Produce **software** that is

- correct, ⇝ *use verification techniques*
- efficient, and ⇝ *exploit parallelism*
- useful *(application-dependent)*

with **minimal cost**

- in developer expertise and ⇝ *automated verification*
- in developer time/effort. ⇝ *with no false positives*

Software model checking

Given a program P and a property Φ,
check that all executions of P satisfy Φ.

In our setting:

- P is a multi-threaded program.
- Φ is some safety property
  e.g., the program does not segfault.

Software model checking

Given a program P and a property Φ,
check that all executions of P satisfy Φ.

- It is **fully automated** ("push button" technique).
  One only has to provide P and Φ.
- Supports **weak memory consistency**.
- Unsuccessful verification returns **error traces**.
  i.e. program traces that violate Φ.

Software model checking

Given a program P and a property Φ,
check that all executions of P satisfy Φ.

- It is **fully automated** ("push button" technique).
- Supports **weak memory consistency**.
- Unsuccessful verification returns **error traces**.

- It assumes programs are **bounded**.
- It is **does not scale** well.
- One still has to provide **the program** P and **the property** Φ.

Model checking approaches

Stateful

- Visit all program states by DFS recording visited states to avoid repetition.
- Requires program to have **bounded** state-space.
- Uses **a lot of** memory.

Stateless

- Generate all program executions **without** recording visited states
- Assumes program to **always terminate**.
- **Low** memory usage.

SMT-based: State-space exploration is done by SMT solver.

My favorite model checker: GenMC

State-of-the-art stateless model checker

- Correct, optimal, highly parallelizable
- Works with almost any weak memory model
- Small memory footprint
- Open source

https://github.com/mpi-sws/genmc/

Scalability

Scalability issues and remedies

There are ***way too many*** interleavings.

(exponential in the number of threads and the size of the program)

Scalability issues and remedies

There are ***way too many*** interleavings.

(exponential in the number of threads and the size of the program)

But exploring all interleavings is ***unnecessary***.

- Many interleavings lead to the same outcome.

  DPOR: *Avoid exploring 'equivalent' interleavings*

- Many interleavings are symmetric.

  SR: *Avoid exploring 'symmetric' interleavings*

- The same bug can be exposed by multiple interleavings.

  Bounding: *Explore only 'simple' interleavings*

Scalability and DPOR

The ideal case: Threads access disjoint locations.

$$\begin{array}{l} x[1] := 1 \\ x[x[1]] := 2 \times x[1] \\ \textbf{assert}(x[1] > 0) \end{array} \;\Big\|\; \ldots \;\Big\|\; \begin{array}{l} x[N] := N \\ x[x[N]] := 2 \times x[N] \\ \textbf{assert}(x[N] > 0) \end{array}$$

- **One** execution – **O(N)** verification time

The worst case: Threads access the same locations.

$$\begin{array}{l} lock() \\ unlock() \end{array} \;\Big\|\; \ldots \;\Big\|\; \begin{array}{l} lock() \\ unlock() \end{array}$$

- Without symmetry reduction, **O(N!)** executions.

Concrete example: reader-writer locks?

GenMCv0.10 does not have built-in support for RW-locks.

- How shall we implement them?

Concrete example: reader-writer locks?

GenMCv0.10 does not have built-in support for RW-locks.

- How shall we implement them?

Count the number of readers holding a lock?

- Readers synchronize $\rightsquigarrow$ slow verification.

read-lock()		read-lock()
$x[1] := y$	...	$x[N] := y$
read-unlock()		read-unlock()

- **O(N!)** executions.

Implementing reader-writer locks (2)

A better implementation

- Keep one lock per-thread
- Read-lock acquires the lock of the calling thread
- Write-lock acquires the locks of all the threads

- No contention on the following example

read-lock()		read-lock()
$x[1] := y$	...	$x[N] := y$
read-unlock()		read-unlock()

- **One** execution – **O(N)** verification time

Writing **the program** P

and **the property** Φ

Test clients

- Suppose we want to verify a concurrent queue library.
- We need to write **test clients**, such as:

$$\begin{array}{l} \mathit{enqueue}(1); \\ \mathit{enqueue}(2); \end{array} \;\Big\|\; \begin{array}{l} a := \mathit{dequeue}(); \\ \mathbf{assert}(a \neq 2); \end{array} \qquad \begin{array}{l} \mathit{enqueue}(1); \\ a := \mathit{dequeue}(); \\ \mathbf{assert}(a \neq \bot); \end{array} \;\Big\|\; \begin{array}{l} \mathit{enqueue}(2); \\ b := \mathit{dequeue}(); \\ \mathbf{assert}(b \neq \bot); \end{array}$$

$$\mathit{enqueue}(1); \;\Big\|\; a := \mathit{dequeue}(); \;\Big\|\; b := \mathit{dequeue}(); \\ \mathbf{assert}(a = \bot \vee b = \bot);$$

Test clients

- Suppose we want to verify a concurrent queue library.
- We need to write **test clients**, such as:

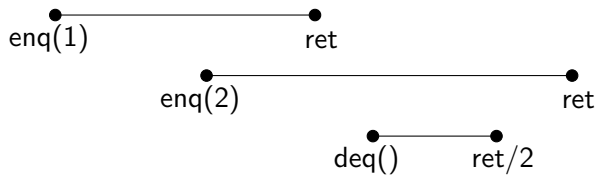
$$\begin{array}{l} \mathit{enqueue}(1); \\ \mathit{enqueue}(2); \end{array} \;\Big\|\; \begin{array}{l} a := \mathit{dequeue}(); \\ \mathbf{assert}(a \neq 2); \end{array} \qquad \begin{array}{l} \mathit{enqueue}(1); \\ a := \mathit{dequeue}(); \\ \mathbf{assert}(a \neq \bot); \end{array} \;\Big\|\; \begin{array}{l} \mathit{enqueue}(2); \\ b := \mathit{dequeue}(); \\ \mathbf{assert}(b \neq \bot); \end{array}$$

$$\mathit{enqueue}(1); \;\Big\|\; a := \mathit{dequeue}(); \;\Big\|\; b := \mathit{dequeue}(); \\ \mathbf{assert}(a = \bot \vee b = \bot);$$

Is there a better way?

Linearizability [Herlihy&Wing, 1990]

- The standard specification of concurrent libraries
- Every concurrent history can be linearized into a legal sequential history (preserving the real-time order of method calls).



- "Global lock" semantics
- Linearizability = Refinement of an atomic specification

Linearizability and weak memory consistency

- There is no such thing as **real-time order**.
  $\rightsquigarrow$ *use "happens-before" order instead.*
- Linearizability specifies only **output** correctness.
- It does not specify **induced synchronization**.

Can the following program print 0?

$$\begin{array}{l} x := 1 \\ \mathit{enqueue}(28) \end{array} \;\Big\|\; \begin{array}{l} \textbf{if } \mathit{dequeue}() > 5 \textbf{ then} \\ \quad \text{print } x \end{array}$$

Refinement of a global lock specification?

- Representation: unbounded array with front and back pointers.
- Enforce atomicity with a **global lock**.

$enqueue(v) \triangleq$
  $lock()$;
  $array[back] := v$;
  $back := back + 1$;
  $unlock()$;

$dequeue() \triangleq$
  $lock()$;
  $v := \bot$;
  $f := front$;
  if $f < back$ **then**
    $v := array[f]$;
    $front := f + 1$;
  $unlock()$;
  return v

Refinement of a global lock specification?

"Global lock" specification:

- Guarantees correct message-passing behavior.
- But enforces **too much** synchronization.
- Is not refined by standard implementations.

We need a more advanced specification.

- Directly use C/C++11 atomics?
  Too complex.

Library-scoped locks

A specification device:

- Like normal locks, provide mutual exclusion and synchronization but only **within** the library.
- They do **not** induce synchronization outside the library.

Simplifies specifications:

- Disentagle atomicity from synchtonization.
- Use C/C++11 atomics to specify desired synchronization.

Queue specification with library-scoped locks

- Enforce synchronization between matching enqueue/dequeue.

$enqueue(v) \triangleq$
  lib-lock();
  $array[back] :=_{\mathbf{rel}} v$;
  $back := back + 1$;
  lib-unlock();

$dequeue() \triangleq$
  lib-lock();
  $v := \bot$;
  $f := front$;
  if $f < back$ **then**
    $v :=_{\mathbf{acq}} array[f]$;
    $front := f + 1$;
  lib-unlock();
  return v

Checking linearizability for particular client

- Generate all executions of the specification.
- Collect the set of **allowed outcomes**:
  results of the methods & induced synchronization.

- Generate all executions of the implementation.
- For each execution, check that its outcome is allowed.
  *i.e. the results of the methods agree, but the implementation may induce **more** synchronization than the specification.*

Checking linearizability for all clients?

Defining client that exposes bug can be tricky.

- Minimal client exposing weak memory bug in HW queue:

$$\begin{array}{c|c|c|c} \mathit{enqueue}(1); & b := \mathit{dequeue}(); & c := \mathit{dequeue}(); & \mathit{enqueue}(4); \\ \mathit{enqueue}(2); & \mathit{enqueue}(3); & d := \mathit{dequeue}(); & a := \mathit{dequeue}(); \end{array}$$
$$\textbf{assert}(b \neq 2 \vee c \neq 3 \vee d \neq 4 \vee a \neq 1);$$

We want to check linearizability for **all *bounded*** clients.

- Use the most parallel client?

Most parallel client (MPC)

Inkove K library operations in parallel, e.g.:

$$enqueue(1) \parallel enqueue(2) \parallel dequeue() \parallel dequeue()$$

Key property:

- *Every* client execution can be obtained by *some* MPC execution by adding synchronization and applying a value substitution.

- MPC **sufficient** for showing **assertion safety**.
- On its own, **not so useful** for showing **linearizability**.

Checking linearizability with MPC

Phase 1: Analyze the specification

- For each outcome, calculate minimal happens-before extensions forbidding it.

Phase 2: Verify the implementation

For each implementation execution:

- Check that projection allowed by the specification (as before)
- Additionally, check that every recorded hb-extension renders the execution inconsistent.

Implementation – Evaluation

- Prototype implementation over **GenMC**
  Employing DPOR, SR & other optimizations
- Standard data structure benchmarks:
  Michael-Scott, DGLM, Herlihy-Wing, etc.
- Scales to **8-10 operations**
  Successfully proves refinement of atomic specification
- Finds subtle **weak memory bug** in Herlihy-Wing queue
  4 enqueues, 4 dequeues, $<$ 1 min

Conclusion

Summary

- Model checking for linearizability
- Scalability issues of SMC

Future work

- Further enhance SMC scalability
- Non-linearizable libraries