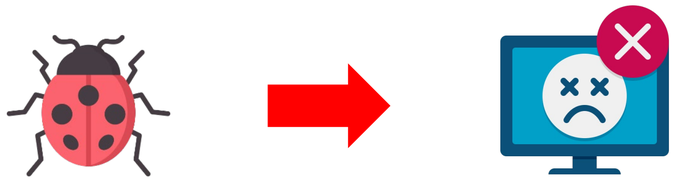
CONVEROS: Practical Model Checking for Verifying Rust OS Kernel Concurrency

Ruize Tang[1,*], Minghua Wang[2], Xudong Sun[3], Lin Huang[2], Yu Huang[1], Xiaoxing Ma[1]

[1] Nanjing University
[2] Ant Group
[3] University of Illinois Urbana-Champaign

[*] Work done during internship at Ant Group

Background and motivation

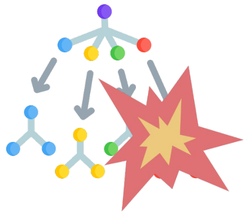


- **Asterinas OS kernel**
  - Rust-based, open-source kernel
  - Linux ABI compatibility
  - 100K LoC, actively developed
  - Designed for high reliability
- **Problem: concurrency bugs**
  - Persist even in Rust (e.g., deadlocks and data races from design flaws)
  - Lead to kernel panics, hangs, etc.
- **Approach: model checking**
  - Explores all states of concurrent systems
  - Proven effective in detecting deep bugs (e.g., AWS, Microsoft, MongoDB)

Challenges in applying model checking to OS kernels

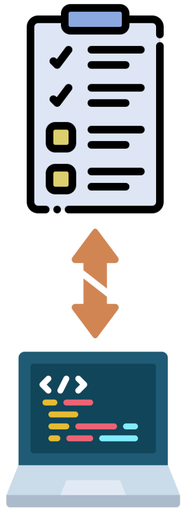
Challenge 1: Writing high-quality specs

- **High barrier to entry**: Concurrency modules are tricky, requiring both domain knowledge and formal modeling expertise
- **Scalability issues**: Poorly structured specs cause state explosion and limit verification depth



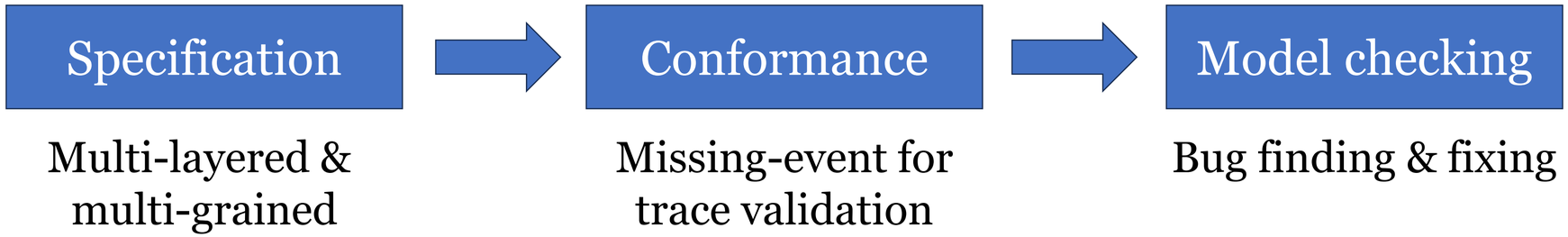
Challenge 2: Checking spec-code conformance

- **Discrepancies:** Caused by modeling errors or code changes
- **Impact:** Leads to false positives and negatives, reducing confidence

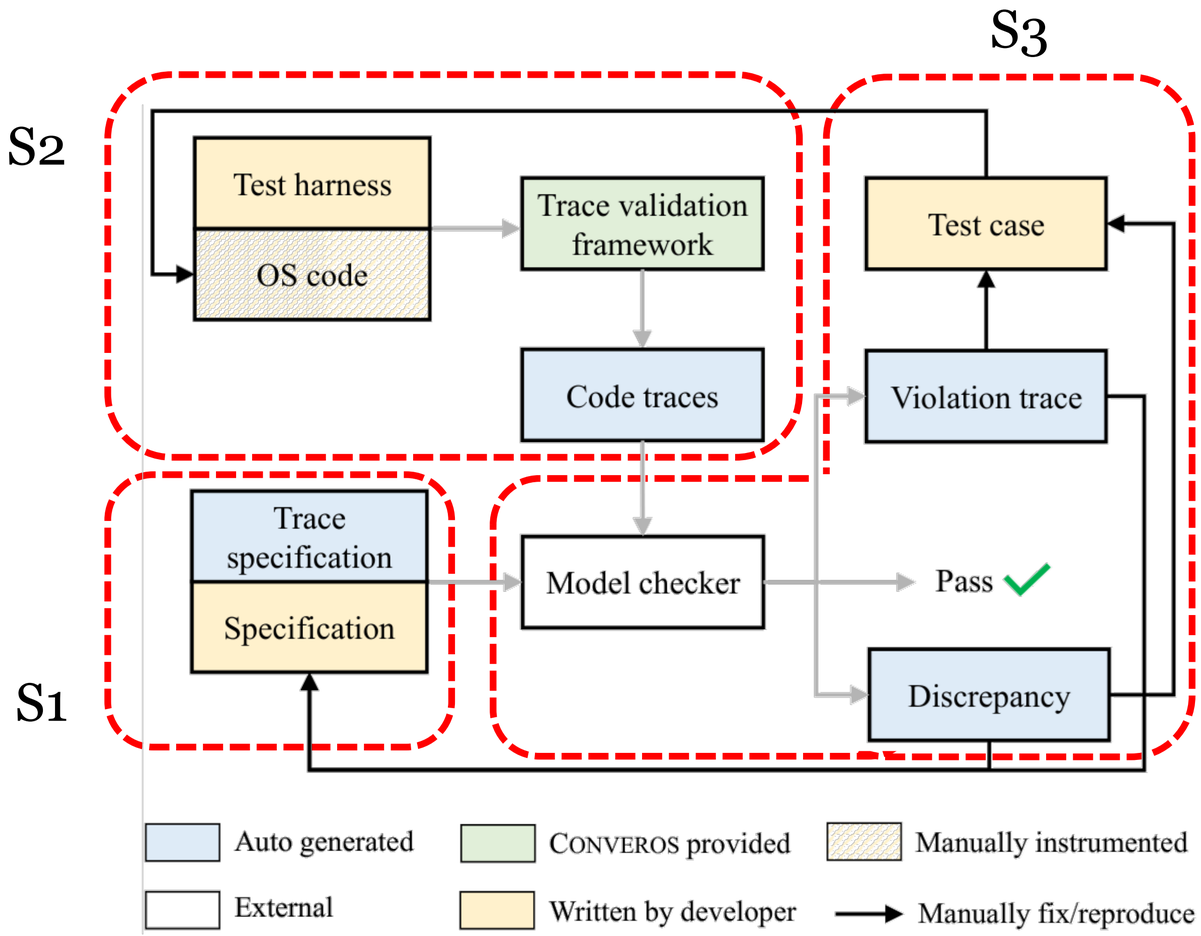


Contributions

- **CONVEROS: a practical model checking methodology**
  - Workflow: Specification development, conformance checking, model checking
- Applied to **12** concurrency modules in Asterinas OS
  - Found **20** real bugs (e.g., deadlocks, livelocks, data races)
  - Only **4** person-months of effort
  - Low spec-to-code ratio (**0.3–2.3**)
- What makes CONVEROS practical
  - **For specification:** Multi-layered and multi-grained specs to reduce effort and state space
  - **For conformance:** Trace validation with “missing event” support



CONVEROS overview



- S1: Specification development
  - PlusCal
  - Multi-layered & multi-grained
- S2: Conformance checking
  - Instrumentation for logging
  - Enhanced trace validation (e.g., missing-event support)
  - Revise spec/code for discrepancies
- S3: Model checking
  - Fix code bugs
  - Check the correctness of fixes

Multi-layered specifications

- Multi-layered: High-level & low-level specs
- **High-level specs**
  - Capture the design intent of sync primitives
  - Model public APIs (e.g., lock(), unlock())
  - Abstract away internal implementation details
  - Benefits
    - Easy to write and verify
    - Enables early conformance checking
    - Serve as a basis for developing low-level specifications

```
CONSTANTS PS \* Process set
(*--algorithm AbstractLock {
variables locked = FALSE;
procedure acquire_lock() {
lock: \* A label
  await locked = FALSE;
  locked := TRUE;
  return;
}
procedure release_lock() {
unlock:
  locked := FALSE;
  return;
}
fair process (p \in PS) {
start:
  while (TRUE) {
    call acquire_lock();
  cs:
    skip;
    call release_lock();
  }
}}*)
```

```
VARIABLES locked, pc
\* Some variables are omitted
vars == <<locked, pc>>
lock(p) ==
  /\ pc[p] = "lock"
  /\ locked = FALSE
  /\ locked' = TRUE
  /\ pc' = [pc EXCEPT ![p]="cs"]
acquire_lock(p) == lock(p)
start(p) ==
  /\ pc[p] = "start"
  /\ pc' = [pc EXCEPT ![p]="lock"]
  /\ UNCHANGED locked
\* Some actions are omitted
Init ==
  /\ locked = FALSE
  /\ stack = [p \in PS |-> <<>>]
  /\ pc = [p \in PS |-> "start"]
Next == \E p \in PS:
  \/ acquire_lock(p)
  \/ release_lock(p)
  \/ start(p)
  \/ cs(p)
```

High-level specification of the lock() and unlock() API (written in PlusCal, compiled to TLA+ for verification)

Multi-layered specifications

```rust
let mut guard = mutex.lock();
*guard += 1;
// `guard` goes out of scope here, lock released via drop()
```

Rust background: implicit unlock() via drop()

Low-level specs

- Build based on high-level specs to closely reflect implementation
- Use labeled blocks to define atomic actions
- Translate Rust logic into PlusCal processes and actions
  - Capture detailed control logic and state changes
  - Explicitly model Rust-specific semantics (e.g., drop())

```
variables locked = FALSE, has_woken_of_wakers = <<>>,
  num_wakers = 0, wakers_lock = FALSE, wakers = <<>>;
procedure acquire_lock()
variable waiter_ref; (* Local variable *) {
fast_path:
  if (locked = FALSE) { locked := TRUE; return; };
slow_path:
  has_woken_of_wakers := Append(has_woken_of_wakers, FALSE);
  waiter_ref := Len(has_woken_of_wakers);
loop:
  while (TRUE) {
\*enqueue: skip;
    acquire_wakers_lock:
      await wakers_lock = FALSE; wakers_lock := TRUE;
    enqueue_waker:
      wakers := Append(wakers, waiter_ref);
    inc_num_wakers:
      num_wakers := num_wakers + 1;
    release_wakers_lock:
      wakers_lock := FALSE;
  try_lock:
    if (locked = FALSE) { locked := TRUE; goto drop_waiter; };
  do_wait:
    await has_woken_of_wakers[waiter_ref] = TRUE;
    has_woken_of_wakers[waiter_ref] := FALSE;
  };
drop_waiter:
  has_woken_of_wakers[waiter_ref] := TRUE; return; }
```

Low-level spec for Mutex::lock() in PlusCal

```rust
pub struct Waker { has_woken: AtomicBool } // has_woken_of_wakers
pub struct Waiter { waker: Arc<Waker> }
pub struct WaitQueue { num_wakers: AtomicU32, // num_wakers
  wakers: SpinLock<VecDeque<Arc<Waker>>>} // wakers_lock, wakers
pub struct Mutex<T> {lock: AtomicBool, queue: WaitQueue, val: T}
impl<T> Mutex<T> {
  pub fn lock(&self) -> MutexGuard<T> { // acquire_lock in spec
    self.queue.wait_until(|| self.try_lock())
  }}
impl WaitQueue {
  pub fn wait_until(&self, cond: ...) -> ... {
    if let Some(res) = cond() { return res; }      // fast_path
    let (waiter, _) = Waiter::new_pair();          // slow_path
    let cond = || { self.enqueue(waiter.waker()); // enqueue
                    cond() };                      // try_lock
    loop {                                         // loop
      if let Some(res)=cond() {/* drop(waiter); */ return res;};
      waiter.waker.do_wait();                      // do_wait
    }
  }
  pub fn enqueue(&self, waker: Arc<Waker>) {
    let mut wakers = self.wakers.lock(); // acquire_wakers_lock
    wakers.push_back(waker);             // enqueue_waker
    self.num_wakers.fetch_add(1);        // inc_num_wakers
    /* drop(wakers); */                  // release_wakers_lock
  }}
impl Drop for Waiter {                   // drop_waiter
  fn drop(&mut self) { self.waker.has_woken.swap(true); }}
```

Corresponding Rust implementation of Mutex::lock()

Multi-grained specifications

- Fine-grained specs lead to state explosion
- Coarse-grained may miss bugs
- We adopt a multi-grained approach
  - Fine-grained specs for basic sync primitives
  - Mix-grained for modules built on top of sync primitives
    - Omit internal details of verified components
    - Coarsen verified behaviors (e.g., model them as atomic variables)
    - Correctness is retained in coarsened specs by the interaction-preserving principle[1]

```
3 pub struct WaitQueue { num_wakers: AtomicU32, // num_wakers
4   wakers: SpinLock<VecDeque<Arc<Waker>>>} // wakers_lock, wakers
5 pub struct Mutex<T> {lock: AtomicBool, queue: WaitQueue, val: T}
6 impl<T> Mutex<T> {
7   pub fn lock(&self) -> MutexGuard<T> { // acquire_lock in spec
8     self.queue.wait_until(|| self.try_lock())
9   }}
```

Code snippet of Mutex::lock()

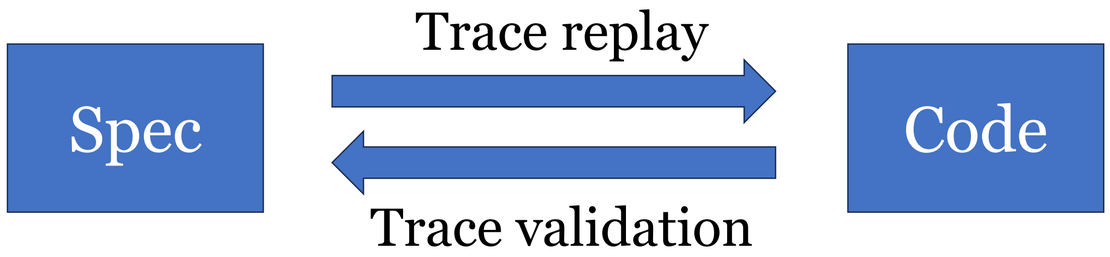
```
1  variables locked = FALSE, has_woken_of_wakers = <<>>,
2    num_wakers = 0, wakers_lock = FALSE, wakers = <<>>;
3  procedure acquire_lock()
4  variable waiter_ref; (* Local variable *) {
5  fast_path:
6    if (locked = FALSE) { locked := TRUE; return; };
7  slow_path:
8    has_woken_of_wakers := Append(has_woken_of_wakers, FALSE);
9    waiter_ref := Len(has_woken_of_wakers);
10 loop:
11   while (TRUE) {
12 \*enqueue: skip;
13     acquire_wakers_lock:
14       await wakers_lock = FALSE; wakers_lock := TRUE;
15     enqueue_waker:
16       wakers := Append(wakers, waiter_ref);
17     inc_num_wakers:
18       num_wakers := num_wakers + 1;
19     release_wakers_lock:
20       wakers_lock := FALSE;
21   try_lock:
22     if (locked = FALSE) { locked := TRUE; goto drop_waiter; };
23   do_wait:
24     await has_woken_of_wakers[waiter_ref] = TRUE;
25     has_woken_of_wakers[waiter_ref] := FALSE;
26   };
27 drop_waiter:
28   has_woken_of_wakers[waiter_ref] := TRUE; return; }
```

Low-level spec for Mutex::lock()

[1] L. Ouyang, X. Sun, R. Tang, Y. Huang, M. Jivrajani, X. Ma, and T. Xu. Multi-Grained Specifications for Distributed System Model Checking and Verification. In Proceedings of the 20th European Conference on Computer Systems (EuroSys), 2025.

Conformance checking

- **Goal**: Bridge the gap between specification and implementation
- Two common approaches in TLA+
  - Trace reply (spec → code): generate traces from the spec and replay them in the code
    - Challenging for OS-level code: requires deterministic control of thread interleavings
  - **Trace validation** (code → spec): run the code to generate execution traces, and check whether the observed behavior allowed by the specification
    - Practical for OS kernels: only requires lightweight logging instrumentation



Trace validation

- Trace collection
  - Develop test harnesses and instrument the code according to the spec's labels
  - Trace logs include timestamp, process ID, event name, and variable updates

```
CONSTANTS PS \* Process set
(*--algorithm AbstractLock {
variables locked = FALSE;
procedure acquire_lock() {
lock: \* A label
  await locked = FALSE;
  locked := TRUE;
  return;
}
procedure release_lock() {
unlock:
  locked := FALSE;
  return;
}
fair process (p \in PS) {
start:
  while (TRUE) {
    call acquire_lock();
  cs:
    skip;
    call release_lock();
  }
}}*)
```

Specification

```
pub struct MutexModel;
impl TlaModel for MutexModel {
  fn run(&self, procs: usize, loops: usize) {
    let mutexlock = Arc::new(Mutex::new(0));
    let proc_func = move || {
      for _i in 0..loops {
        TlaLogger::new("start").next_pc("lock").record();
        let guard = mutexlock.lock();
        TlaLogger::new("lock").next_pc("cs").record();
        TlaLogger::new("cs").next_pc("unlock").record();
        drop(guard);
        TlaLogger::new("unlock").record();
      }
    };
    run_procs(procs, proc_func); // Provided by trace framework
  }}
```

Test harness

```
{"timestamp":100,"event":"start","p":2,"pc":[{"op":"Update", "path":[2],"args":["lock"]}]}
{"timestamp":110,"event":"start","p":1,"pc":[{"op":"Update", "path":[1],"args":["lock"]}]}
```

Trace log

Trace validation

- Trace specification
  - ❶ Extends high-level/low-level specs
  - ❷ Constrains allowed actions
- Validation process
  - Reads trace logs
  - Constrain each log's event and variable updates against the spec
  - Explore allowed (enabled) actions using TLC model checker
  - If a discrepancy is detected
    - At the low-level spec: revise the spec to fix modeling errors
    - At the high-level spec: fix code deviation bugs

```
1  \* Simplified actions of the high-level lock specification
2  start(self) ==
3    /\ pc[self] = "start" /\ pc' = [pc EXCEPT ![self] = "lock"]
4    /\ UNCHANGED locked
5  lock(self) ==
6    /\ pc[self] = "lock" /\ pc' = [pc EXCEPT ![self] = "cs"]
7    /\ locked = FALSE /\ locked' = TRUE
8  \* Trace specification (below)
9  VARIABLES l \* Initialized to 1
10 Trace == LoadTrace("trace.ndjson") \* Loaded from trace file
11 cur_line == Trace[l]
12 UpdateVariables(line) ==
13   "pc" \in DOMAIN line => pc' = UpdateVar(pc, line.pc) /\ ...
14 IsEvent(e) ==
15   /\ l \in 1..Len(Trace) /\ l' = l + 1
16   /\ pc[cur_line.p] = e \* Use PlusCal's `pc` as event constraint
17   /\ UpdateVariables(cur_line)
18 SpecActions(p) == start(p) \/ lock(p) \/ ...
19 TraceNext == IsEvent(cur_line.event) /\ SpecActions(cur_line.p)
```

❶ ❷

Trace spec

```
1 {"timestamp":100,"event":"start","p":2,"pc":[{"op":"Update", "path":[2],"args":["lock"]}]}
2 {"timestamp":110,"event":"start","p":1,"pc":[{"op":"Update", "path":[1],"args":["lock"]}]}
```

Trace log

Enhanced trace validation

- Issues:
  - Logged timestamps may be inaccurate
  - Logging every label can be burdensome
- Our improvement: Allow missing events
  - Let TLC infer valid events
  - Insert “missing event” logs before and after an action to define a possible time window
  - Consecutive missing events are merged
    - Ensures that each missing event is followed by a normal event

```rust
pub struct MutexModel;
impl TlaModel for MutexModel {
  fn run(&self, procs: usize, loops: usize) {
    let mutexlock = Arc::new(Mutex::new(0));
    let proc_func = move || {
      for _i in 0..loops {
        TlaLogger::new("start").next_pc("lock").record();
        TlaLogger::new_missing().cur_pc("lock").record();
        let guard = mutexlock.lock();
        TlaLogger::new_missing().cur_pc("lock").record();
        TlaLogger::new("cs").next_pc("unlock").record();
        TlaLogger::new_missing().cur_pc("unlock").record();
        drop(guard);
        TlaLogger::new_missing().cur_pc("unlock").record();
      }
    };
    run_procs(procs, proc_func);
  }
}
```

Test harness (missing event)


```
{"timestamp":110,"event":"MissingEvent","ps":{1,2},"cur_pc":[{"op":"in","path":[1],"args":["lock"]},{"op":"in","path":[2],"args":["lock"]}]}
{"timestamp":120,"event":"cs","p":1}
{"timestamp":130,"event":"MissingEvent","ps":{1,2},"cur_pc":[{"op":"in","path":[1],"args":["lock"]},{"op":"in","path":[2],"args":["lock"]}]}
{"timestamp":140,"event":"cs","p":2,"converged":[1]}
```


Trace log (missing event)

Enhanced trace specification

- Handling missing events in trace specification
  - **Branch ❶**: Matches the next non-missing event
  - **Branch ❷**: Explores all enabled actions without consuming the missing event, keeping it active for further steps
- Limit search space from missing events for efficiency

  ① Max depth pruning (e.g., depth < 10)

  ② Convergence points: stop exploration for a missing event once its corresponding convergence log is encountered

```
next_line == Trace[l + 1]
MissingEvent(Actions(_)) == \* Actions: operator with 1 argument
① /\ depth < cur_line.max_depth /\ depth' = depth + 1 \* Max depth pruning
② /\ TLGGet(l) \* Converge point pruning
  /\ UNCHANGED l    \* ps (below): allowed process set
  /\ \E p \in cur_line.ps: UpdateVariables(cur_line) /\ Actions(p)
IsNextEvent(e) ==
  /\ l \in 1..(Len(Trace)-1) /\ l' = l + 2
  /\ pc[next_line.p] = e
  /\ UpdateVariables(next_line)
② /\ \A i \in cur_line.converged: TLCSet(i, FALSE) \* Set converge points
ExecuteEvent(Actions(_)) ==
  IF cur_line.event /= "MissingEvent"
  THEN IsEvent(cur_line.event) /\ Actions(cur_line.p)
❶ ELSE \/ IsNextEvent(next_line.event) /\ Actions(next_line.p)
❷      \/ MissingEvent(Actions)
TraceNext == ExecuteEvent(SpecActions)
```

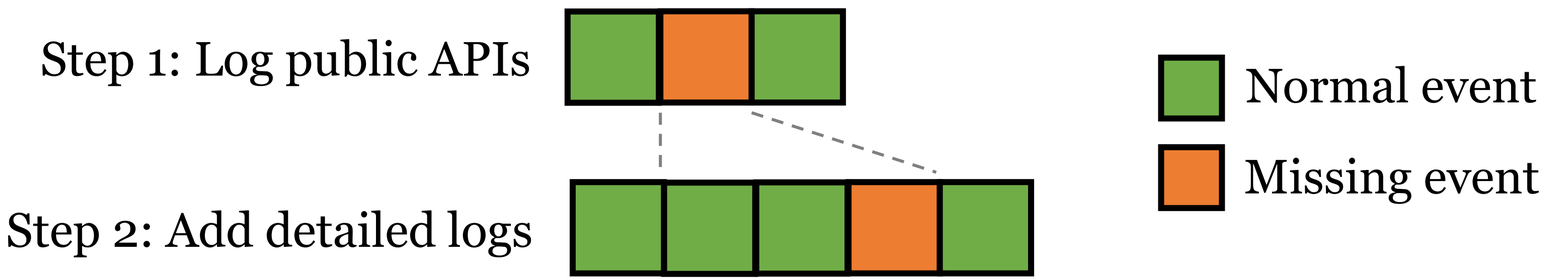
Trace spec (missing event)


```
{"timestamp":110,"event":"MissingEvent","ps":{1,2},"cur_pc":[{"op":"in","path":[1],"args":["lock"]},{"op":"in","path":[2],"args":["lock"]}]}
{"timestamp":120,"event":"cs","p":1}
{"timestamp":130,"event":"MissingEvent","ps":{1,2},"cur_pc":[{"op":"in","path":[1],"args":["lock"]},{"op":"in","path":[2],"args":["lock"]}]}
{"timestamp":140,"event":"cs","p":2,"converged":[1]}
```


Trace log (missing event)

Enhanced trace collection

- Incremental trace collection
  - Start from public APIs of synchronization primitives
  - Gradually instrument internal statements
  - Enabled by missing events
- Benefits
  - Enables early detection and debugging of spec–code discrepancies
  - Reduces logging overhead in early development stages



Model checking

- Use TLC model checker to check for property violations
  - E.g., common safety and liveness properties

```
MutualExclusion     == \* Safety
  \A i, j \in Procs: (i /= j) => ~(pc[i] = "cs" /\ pc[j]="cs")
DeadAndLiveLockFree == \* Liveness
  \E i \in Procs: pc[i]="lock" ~> (\E j \in Procs: pc[j]="cs")
StarvationFree      == \* Liveness
  \A i \in Procs: pc[i]="lock" ~> (pc[i]="cs")
```

- Create test cases based on violation traces
  - If the bug is reproducible
    - Fix the spec and the code
    - Rerun trace validation to check conformance
  - If the bug is not reproducible
    - The discrepancy likely lies in the spec, revise the spec
- By-product bugs (e.g., panics or hangs) may also be uncovered during test execution

Evaluation

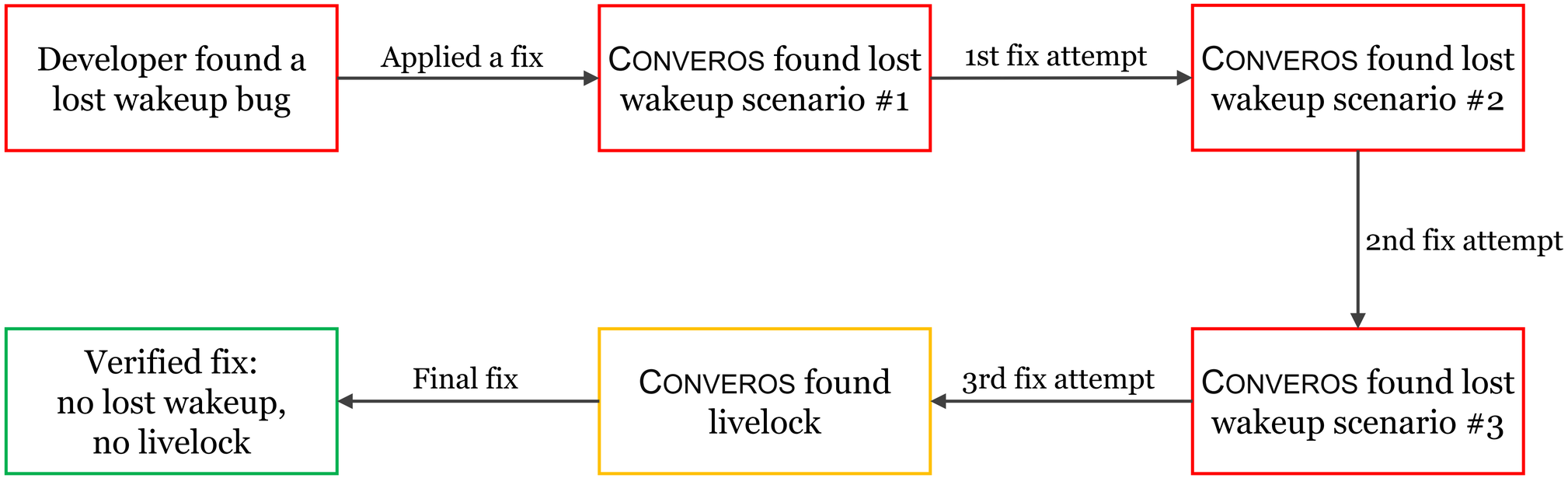
- Verified 12 concurrency modules in Asterinas OS
  - ~4,000 lines of Rust code
- Bugs found: 20 in total
  - 9 from model checking
  - 11 by-product bugs
    - 2 found during modeling
    - 9 found during test case execution
  - All confirmed, 14 fixed
- Effort: ~4 person-months in total

Bugs found

ID	Module	Violation/Manifestation	Description
Found by CONVEROS			
#1	RangeLock	Deadlock-free	Lost wakeup due to waiting on an outdated wait queue
#2	Mutex	Mutual exclusion	Incorrect guard construction in lock() causing unintended unlock() on drop()
#3	RwLock	Livelock/Starvation-free	Non-atomic read() operation racing with downgrade()
#4	RwMutex	Deadlock-free	Wrong wakeup condition causing a dropped upreader to never wake up a waiter
#5	Semaphore	Semaphore count ≥ 0	Semaphore count decreases below 0 due to unhandled multi-operation semop()
#6	Futex	Deadlock-free	Signal or timeout leaving outdated futex item uncleared, causing lost wakeup
#7	Flock	Deadlock-free	Incorrect wait condition causing permanent wait
#8	Pipe	Write atomicity	Writes smaller than PIPE_BUF size not guaranteed to be atomic
#9	TTY	Deadlock-free	Circular dependencies from incorrect locking order causing TTY hang
By-product bugs			
#10	Semaphore	(Found during modeling)	Incorrect initialization of the waiter count, leading to wrong lookup results
#11	Futex	(Found during modeling)	Poor hash algorithm leads to uneven key distribution, overloading a few buckets
#12	Atomic mode	Kernel panic	Misuse of spinlock/mutex breaks atomic mode
#13	IRQ	Stack overflow	Improper IRQ enabling during the bottom half leads to nested IRQs
#14	TLB	Kernel hang	Deadlock caused by a spinlock used in IRQ without disabling IRQ outside it
#15	Trap	Kernel panic and hang	Uncleared direction flag (rflags.DF) leads to memory corruption on traps
#16	Task switch	User process crash	Missing FPU state save/restore causes memory corruption on task switch
#17	Sendfile syscall	User data truncation	Short writes cause data truncation when input is read but not fully written
#18	Kernel tests	Kernel test hang	Assertion in kernel test threads leads to infinite error message unwraps
#19	Kernel threads	Kernel panic	New scheduler feature breaks CPU affinity setting
#20	Logging	Kernel hang	Logging allocating memory during low-memory rescue triggers recursive rescues

Example bug: Hard-to-fix lost wakeup in RangeLock

- RangeLock's (i.e., file record lock) complex semantics make it easy to introduce subtle concurrency bugs
- Developer found and fixed a lost wakeup bug, but the fix was flawed
- CONVEROS helped identify issues in each fix until the bug was correctly resolved



Example discrepancy: Subtle modeling error in RwMutex

- Discrepancy in try_upread logic
  - Code: sets the UPREADER bit and checks previous lock state (WRITER | UPREADER)
  - Buggy spec: checked current writer_lock after update (not previous state)
  - Correct spec: stores previous states of both writer_lock and upreader_lock
  - Found by trace validation: writer_lock changed before the check label

```rust
pub fn try_upread(&self) -> Option<RwMutexUpReadGuard<T>> {
  let lock = self.lock.fetch_or(UPREADER) & (WRITER | UPREADER);
  if lock == 0 {
    Some(RwMutexUpReadGuard { inner: self })
  }
```

RwMutex implementation

```
procedure try_upread()
variable prev_lock; {
fetch_or:
-  prev_lock := upreader_lock;
+  prev_lock := <<upreader_lock, writer_lock>>;
   upreader_lock := TRUE;
check:
-  if (~prev_lock /\ ~writer_lock) {
+  if (~prev_lock[1] /\ ~prev_lock[2]) {
   success:
     role[self] := UPREADER;
     return;
   }
```

Another thread holding the write lock released it between the “fetch_or” and “check” labels

RwMutex specification (discrepancy and fix)

Verification effort

Code		Specification			Estimate effort	
Module	**#LOC**	**#LOC**	**#Var.**	**#Act.**	**Spec.**	**Conf.**
SpinLock	147	49	1	5	1.5	5
Mutex	113	89	4	14	4	3
RwLock	404	261	5	19	5	2.5
RwMutex	202	460	9	47	3.5	2.5
CondVar	238	199	6	25	3.5	1.5
Semaphore	622	490	10	37	9	2
PageCursor	468	297	10	21	5	-
Pipe	392	143	7	17	1.5	1
RangeLock	457	529	9	24	4	2
Flock	144	176	7	12	1.5	0.5
Futex	469	201	5	21	4	1
TTY	309	138	4	20	1	0.5
Total	3965	3032	77	262	43.5	21.5

- Specification-to-code ratio
  - Ranges from 0.3 to 2.3
- Specifications development
  - 43.5 person-days
- Conformance checking
  - 21.5 person-days
- Total effort
  - ~4 person-months

Conclusion

- CONVEROS: a practical model checking methodology
  - 3-step workflow: specification → conformance checking → model checking
  - Multi-layered, multi-grained specifications for usability and scalability
  - Enhanced trace validation with missing-event support for trace validation
- Evaluation summary
  - Verified **12** concurrency modules (~4,000 LoC) in Asterinas OS
  - Found **20** bugs (e.g., deadlocks, livelocks, kernel panics)
  - ~4 person-months of effort; spec-to-code ratio: 0.3–2.3

THANK YOU!