EECS
Electrical Engineering and
Computer Sciences

BERKELEY PARLAB

PARALLEL COMPUTING LABORATORY

CONCURRIT: Testing Concurrent Programs with Programmable State-Space Exploration

(A DSL for Writing Concurrent Tests)

Jacob Burnim, **Tayfun Elmas***
George Necula, Koushik Sen
University of California, Berkeley

HotPar 2012

How to write an xUnit-like test for a concurrent program?

- Consider:
  - Mozilla SpiderMonkey JavaScript Engine
    - Used in Firefox browser
    - 121K lines of code
  - Want to test `JS_NewContext`, `JS_DestroyContext`
    - Contain 2K < lines of code

How to write an xUnit-like test for a *sequential* program?

- Fix inputs → Deterministic test
  - If there is a bug, every run manifests it!

```
// check if any assertion fails
test_Context() {
   ...
   JSContext *cx = JS_NewContext(rt, 0x1000);
   if (cx) {
      ...
      JS_DestroyContext(cx);
   }
}
```

How to write an xUnit-like test for a concurrent program?

- Nondeterminism due to thread schedules
  - **Hard** to specify and control schedule!

```
// check if any assertion fails
test_Context() {

    ... // create 10 threads to run testfunc

}

testfunc() {
  JSContext *cx = JS_NewContext(rt, 0x1000);
  if (cx) {
     ...
     JS_DestroyContext(cx);
  }
}
```

Approaches to testing concurrent programs

1. **Stress testing:** No control over thread schedules

    ➔ No guarantee about generated schedules

```
// check if any assertion fails
test_Context() {
  Loop 1000 times {
    ... // create 100 threads to run testfunc
  }
}

testfunc() {
  JSContext *cx = JS_NewContext(rt, 0x1000);
  if (cx) {
     ...
     JS_DestroyContext(cx);
  }
}
```

Approaches to testing concurrent programs

1. **Stress testing:** No control over thread schedules

   ➔ No guarantee about generated schedules

2. **Model checking:** Enumerate all possible schedules

   – Too many schedules

   ➔ Not scalable for large programs!

Missing: Programmer has no direct control on thread schedule

- **Key** to effective and efficient testing

Programmers have often insights/ideas about which schedules to look at

Wan-Teh Chang   2002-08-29 16:08:33 PDT   Description [reply] [-] [reply] [-]

```
This bug affects the pthreads version of NSPR, which
is used on most Unix platforms.

There is a race condition when we use PR_Interrupt to
interrupt PR_WaitCondVar.

Suppose thread A is calling PR_WaitCondVar and thread
B is interrupting thread A.  The following event
sequence is problematic.

Thread A                      Thread B
==========================    ========================
Test its interrupt flag

Set thred->waiting to cvar

                              Set thread A's interrupt
                              flag

                              Call pthread_cond_broadcast
                              on thread A's 'waiting'
                              cvar

Call pthread_cond_wait

======================================================

Thread A misses the broadcast and blocks in
pthread_cond_wait forever.

This can be reproduced with the 'join' test program,
at least on Red Hat Linux 6.2.
```

DO NOT READ!

Programmers have often insights/ideas about which schedules to look at

paul.barnetta@smx.co.nz 2009-02-04 13:54:41 PST Description [reply] [-] [reply] [-]

```
I have a multi-threaded application that periodically crashes. I maintain a
pool of JSContexts: as they're requested from the pool JS_SetContextThread and
JS_BeginRequest are called; when they're returned JS_EndRequest and
JS_ClearContextThread are called.
```

DO NOT READ!

```
The crashes consistently occurs inside js_GC in the following code block:

     while ((acx = js_ContextIterator(rt, JS_FALSE, &iter)) != NULL) {
         if (!acx->thread || acx->thread == cx->thread)
             continue;
         memset(acx->thread->gcFreeLists, 0, sizeof acx->thread->gcFreeLists);
         GSN_CACHE_CLEAR(&acx->thread->gsnCache);
     }

acx always appears to be valid but acx->thread == NULL when the application
crashes (which may be in the memset or GSN_CACHE_CLEAR line). This shouldn't
occur as these lines should be skipped if (!acx->thread)..

What I suspect is happening is that one thread is calling JS_GC while a second
is calling JS_EndRequest and JS_ClearContextThread (in returning a context to
the pool). The call to JS_GC will block until JS_EndRequest finishes.. JS_GC
then resumes.. but while JS_GC is running JS_ClearContextThread also runs (no
locking is done in this?), modifying the value of acx->thread as the code above
runs. acx->thread becomes NULL just before it gets dereferenced and the
application exits.
```

Programmers have often insights/ideas about which schedules to look at

Igor Bukanov 2009-03-09 17:47:12 PDT Comment 5 [reply] [-] [reply] [-]

At least one problem that I can see from the code is that js_GC does the check:

```
if (rt->state != JSRTS_UP && gckind != GC_LAST_CONTEXT)
    return;
```

DO NOT READ!

outside the GC lock. Now suppose there are 3 threads, A, B, C. Threads A and B calls js_DestroyContext and thread C calls js_NewContext.

Fixed, known schedule for threads A and B

First thread A removes its context from the runtime list. That context is not the last one so thread does not touch rt->state and eventually calls js_GC. The latter skips the above check and tries to to take the GC lock.

Before this moment the thread B takes the lock, removes its context from the runtime list, discovers that it is the last, sets rt->state to LANDING, runs the-last-context-cleanup, runs the GC and then sets rt->state to DOWN.

At this stage the thread A gets the GC lock, setup itself as the thread that runs the GC and releases the GC lock to proceed with the GC when rt->state is DOWN.

Unknown schedule for A and C

Now the thread C enters the picture. It discovers under the GC lock in js_NewContext that the newly allocated context is the first one. Since rt->state is DOWN, it releases the GC lock and starts the first context initialization procedure. That procedure includes the allocation of the initial atoms and it will happen when the thread A runs the GC. This may lead precisely to the first stack trace from the comment 4.

Inserting sleeps to enforce a schedule

Sleeps:

- **Lightweight and convenient tool** for programmer
- **BUT:** Ad hoc, not reliable for long, complex schedules.

➔ **Need: Formal** and **robust** way to describe schedules!

```
With the patched NSPR library, run the 'join' test.
The events will happen at the following time instants:

Thread A                            Thread B
===========================         ========================
T0: Test its interrupt flag         T0: Sleep 1 second

T0: Set thred->waiting to
cvar

T0: Sleep 2 seconds

                                    T1: Set thread A's interrupt
                                    flag

                                    T1: Call pthread_cond_broadcast
                                    on thread A's 'waiting' cvar

T2: Call pthread_cond_wait
```

Case study: A bug in SpiderMonkey (1.8rc1)

- In RADBench [Jalbert, Sen, HotPar'10]

DO NOT READ!

```
I have a multi-threaded application that periodically crashes, giving the
following assertion error:

$ ./a.out
Assertion failure: rt->state == JSRTS_UP || rt
jscntxt.cpp:465

I've attached a test program which dem
spawns many threads, each of which create
exiting. I'd expect the number of contexts act
[0..THREADS], possibly transitioning between 0
and showing a race condition in the code?

Reproducible: Always

Steps to Reproduce:
Below is a simple application that exhibits th
me) when run directly from the command line:

8<----

#include <stdlib.h>
#include <pthread.h>

#include "jsapi.h"

static JSRuntime *rt;

#define THREADS 100

static void * testfunc(void *ignored) {

    JSContext *cx = JS_NewContext(rt, 0x1000
    if (cx) {
        JS_BeginRequest(cx);
        JS_DestroyContext(cx);
    }

    return NULL;
}

int main(void) {

    rt = JS_NewRuntime(0x100000);
    if (rt == NULL)
        return 1;

    /* Uncommenting this to guarantee there's always at least
     * one context in the runtime prevents this problem. */
//  JSContext *cx = JS_NewContext(rt, 0x1000);

    int i;
    pthread_t thread[THREADS];
    for (i = 0; i < THREADS; i++) {
        pthread_create(&thread[i], NULL, testfunc, NULL);
    }

    for (i = 0; i < THREADS; i++) {
        pthread_join(thread[i], NULL);
    }

    return 0;
}

8<----

It seems to be very sensitive to timings as I have trouble reproducing the
issue in gdb. For me to trigger it there I just need create/destroy more
contexts per thread, but YMMV.

8<----
```

```
$ ./a.out
Assertion failure: rt->state == JSRTS_UP || rt->state == JSRTS_LAUNCHING, at
jscntxt.cpp:465
```

```
#define THREADS 100

static void * testfunc(void *ignored) {

    JSContext *cx = JS_NewContext(rt, 0x1000);
    if (cx) {
        JS_BeginRequest(cx);
        JS_DestroyContext(cx);
    }

    return NULL;
}
```

Possible buggy schedule from bug report

DO NOT READ!

Igor Bukanov 2009-03-09 17:47:12 PDT Comment 5 [reply] [-] [reply] [-]

```
At least one problem that I can see from the code is that js_GC does the check:

if (rt->state != JSRTS_UP && gckind != GC_LAST_CONTEXT)
    return;

outside the GC lock. Now suppose there are 3 threads, A, B, C. Threads A and B
calls js_DestroyContext and thread C calls js_NewContext.
```

Fixed, known schedule for threads A and B

```
First thread A removes its context from the runtime list. That context is not
the last one so thread does not touch rt->state and eventually calls js_GC. The
latter skips the above check and tries to to take the GC lock.

Before this moment the thread B takes the lock, removes its context from the
runtime list, discovers that it is the last, sets rt->state to LANDING, runs
the-last-context-cleanup, runs the GC and then sets rt->state to DOWN.

At this stage the thread A gets the GC lock, setup itself as the thread that
runs the GC and releases the GC lock to proceed with the GC when rt->state is
DOWN.
```

Unknown schedule for A and C

```
Now the thread C enters the picture. It discovers under the GC lock in
js_NewContext that the newly allocated context is the first one. Since
rt->state is DOWN, it releases the GC lock and starts the first context
initialization procedure. That procedure includes the allocation of the initial
atoms and it will happen when the thread A runs the GC. This may lead precisely
to the first stack trace from the comment 4.
```

Concurrit: A DSL for writing concurrent tests


Igor Bukanov 2009-03-09 17:47:12 PDT Comment 5 [reply] [-] [reply] [-]

At least one problem that I can see from the code is that js_GC does the check:

```
if (rt->state != JSRTS_UP && gckind != GC_LAST_CONTEXT)
    return;
```

outside the GC lock. Now suppose there are 3 threads, A, B, C. Threads A and B calls js_DestroyContext and thread C calls js_NewContext.

DOWN.

Now the thread C enters the picture. It discovers under the GC lock in js_NewContext that the newly allocated context is the first one. Since rt->state is DOWN, it releases the GC lock and starts the first context initialization procedure. That procedure includes the allocation of the initial atoms and it will happen when the thread A runs the GC. This may lead precisely to the first stack trace from the comment 4.

Software Under Test

```
#define THREADS 100

static void * testfunc(void *ignored) {

    JSContext *cx = JS_NewContext(rt, 0x1000);
    if (cx) {
        JS_BeginRequest(cx);
        JS_DestroyContext(cx);
    }

    return NULL;
}
```

+

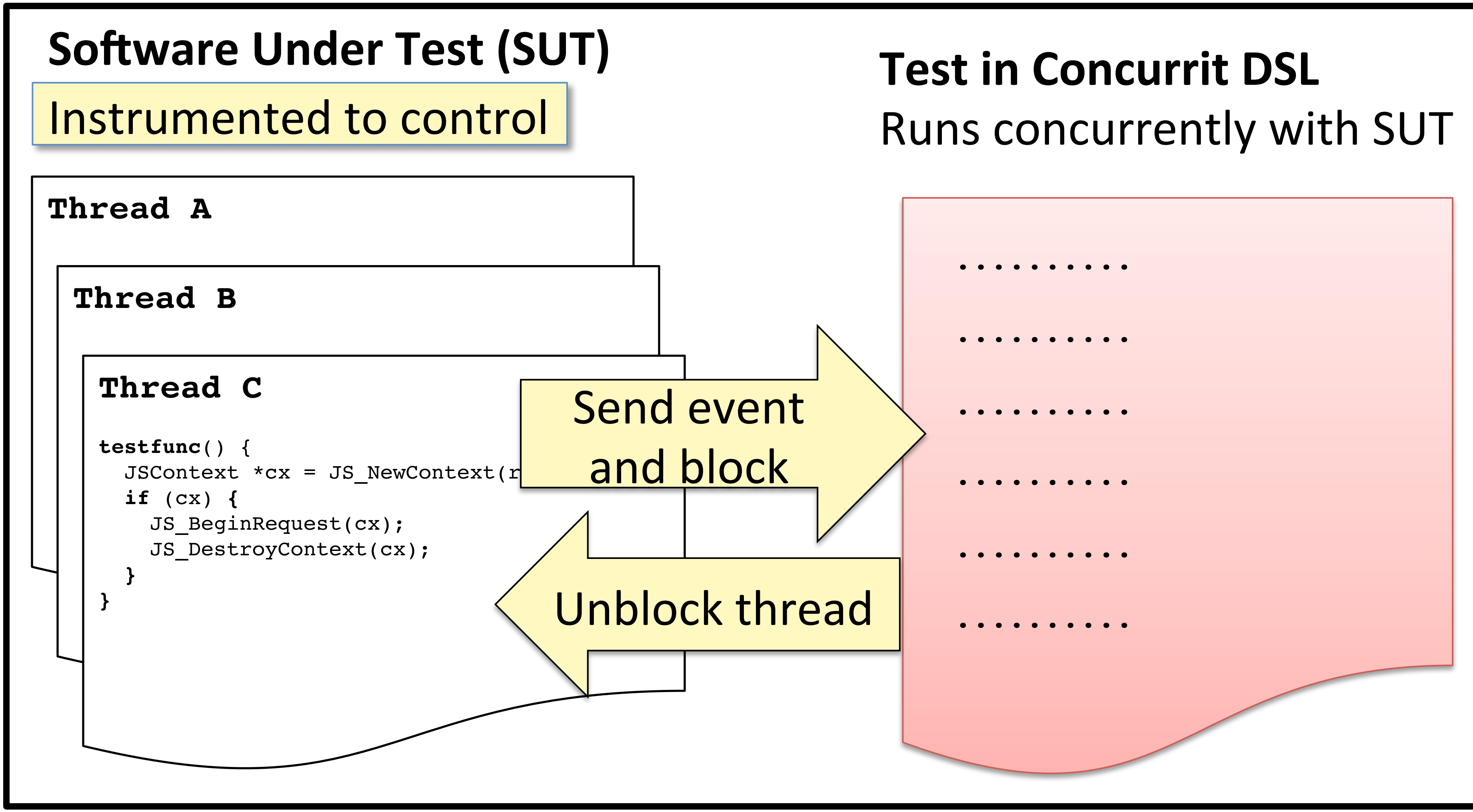
Test in Concurrit DSL

Systematically explore **all-and-only** thread schedules specified by DSL

Specify a set of schedules in **formal**, **concise**, and **convenient** way

Unit-testing programs with Concurrit

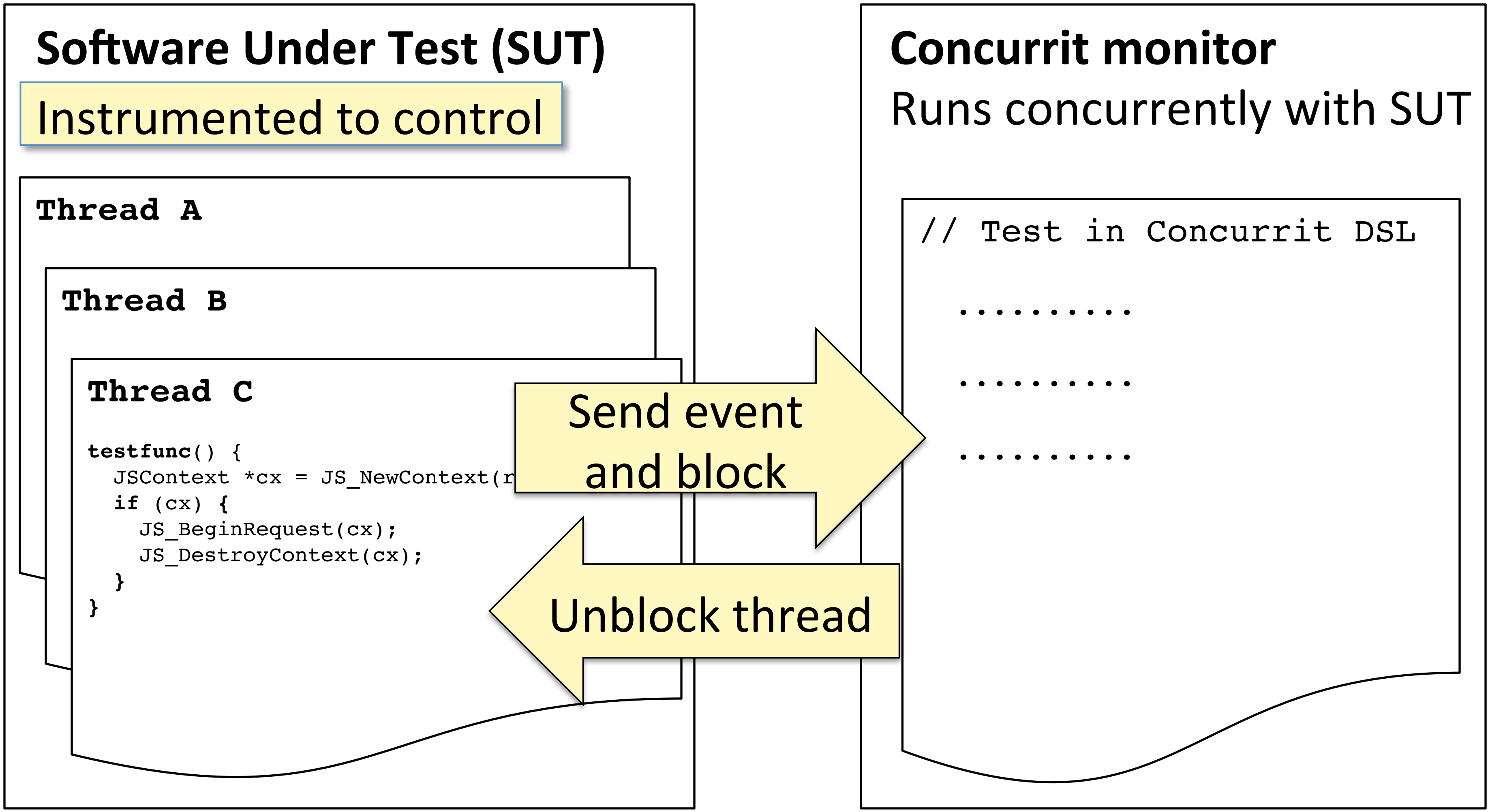
(What about integration tests?: Wait for conclusion)



Kinds of events: Memory read/write, function enter/return, function call, end of thread, at particular source line, user-defined

Unit-testing programs with Concurrit

(What about integration tests?: Wait for conclusion)



Kinds of events: Memory read/write, function enter/return, function call, end of thread, at particular source line, user-defined

Outline

- Bug report for Mozilla SpiderMonkey
- Write tests in Concurrit DSL to generate buggy schedule
  - **Simple schedules:**
    - Few schedules **BUT** not manifesting bug
  - **All schedules:**
    - Manifests bug **BUT** too many schedules
  - **Target buggy schedule in bug report**
    - Few schedules **AND** manifests bug

Possible buggy schedule from bug report

Igor Bukanov 2009-03-09 17:47:12 PDT Comment 5 [reply] [-] [reply] [-]

At least one problem that I can see from the code is that js_GC does the check:

```
if (rt->state != JSRTS_UP && gckind != GC_LAST_CONTEXT)
    return;
```

outside the GC lock. Now suppose there are 3 threads, A, B, C. Threads A and B calls js_DestroyContext and thread C calls js_NewContext.

First thread A removes its context from the runtime list. That context is not the last one so thread does not touch rt->state and eventually calls js_GC. The latter skips the above check and tries to to take the GC lock.

Before this moment the thread B takes the lock, removes its context from the runtime list, discovers that it is the last, sets rt->state to LANDING, runs the-last-context-cleanup, runs the GC and then sets rt->state to DOWN.

At this stage the thread A gets the GC lock, setup itself as the thread that runs the GC and releases the GC lock to proceed with the GC when rt->state is DOWN.

Now the thread C enters the picture. It discovers under the GC lock in js_NewContext that the newly allocated context is the first one. Since rt->state is DOWN, it releases the GC lock and starts the first context initialization procedure. That procedure includes the allocation of the initial atoms and it will happen when the thread A runs the GC. This may lead precisely to the first stack trace from the comment 4.

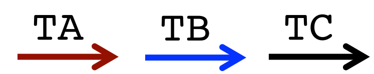
First test: Run each thread sequentially until completion (No interleaving)

```
// Test in Concurrit DSL

TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

LOOP UNTIL TA, TB, TC COMPLETE {

  WITH T IN [TA, TB, TC]

  RUN T UNTIL COMPLETES
}
```

First test: Run each thread sequentially until completion (No interleaving)

```
// Test in Concurrit DSL

TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

LOOP UNTIL TA, TB, TC COMPLETE {

  WITH T IN [TA, TB, TC]

  RUN T UNTIL COMPLETES
}
```

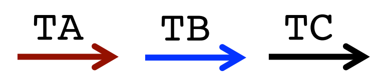
Wait until 3 distinct threads sending events



First test: Run each thread sequentially until completion (No interleaving)

```
// Test in Concurrit DSL

TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

LOOP UNTIL TA, TB, TC COMPLETE {

  WITH T IN [TA, TB, TC]

  RUN T UNTIL COMPLETES
}
```

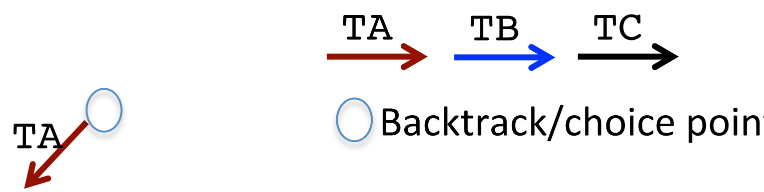
Loop until all 3 threads complete



First test: Run each thread sequentially until completion (No interleaving)

```
// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3:    WITH T IN [TA, TB, TC]

4:    RUN T UNTIL COMPLETES
   }
```

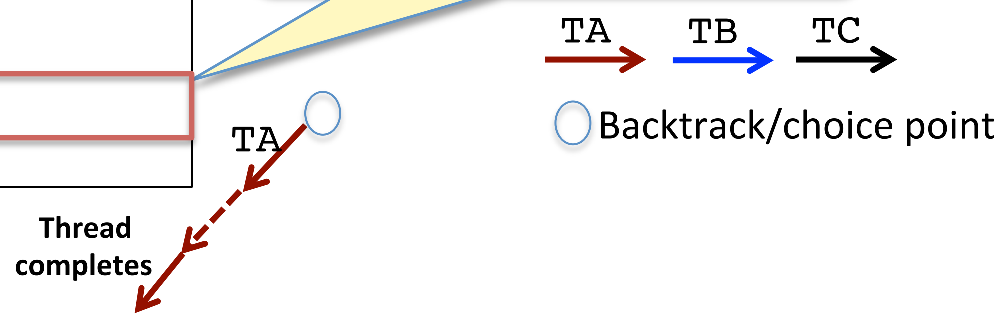
Pick one of the threads



First test: Run each thread sequentially until completion (No interleaving)

```
// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3:   WITH T IN [TA, TB, TC]

4:   RUN T UNTIL COMPLETES
   }
```

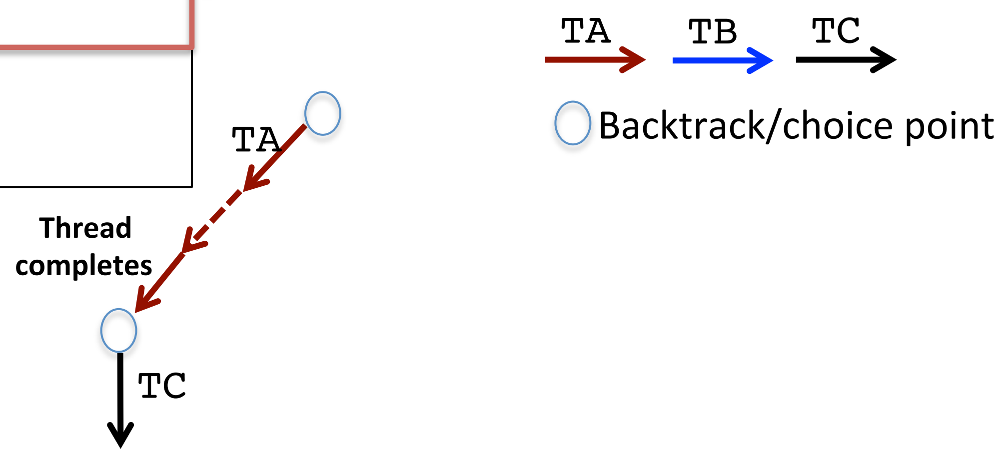
Run selected thread until it completes



First test: Run each thread sequentially until completion (No interleaving)

```
// Test in Concurrit DSL

TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

LOOP UNTIL TA, TB, TC COMPLETE {

    WITH T IN [TA, TB, TC]

    RUN T UNTIL COMPLETES
}
```

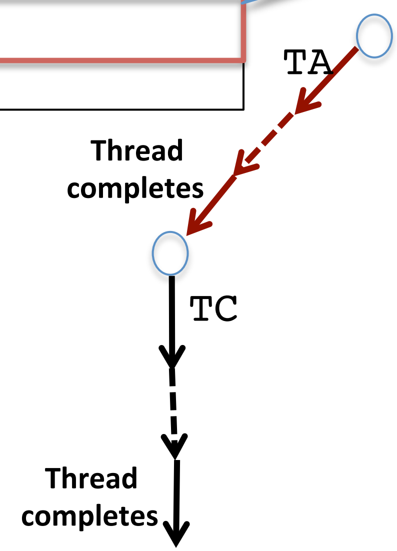
Pick one of the threads



First test: Run each thread sequentially until completion (No interleaving)

```
// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3:   WITH T IN [TA, TB, TC]

4:   RUN T UNTIL COMPLETES
   }
```

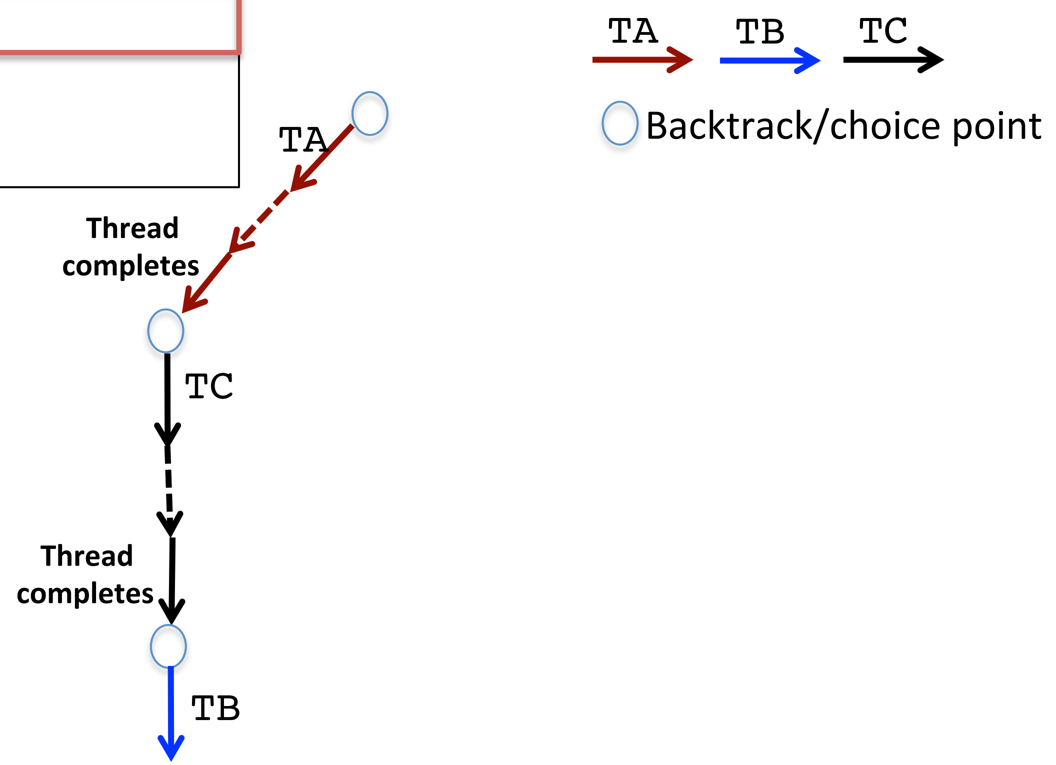
Run selected thread until it completes



First test: Run each thread sequentially until completion (No interleaving)

```
// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3:    WITH T IN [TA, TB, TC]

4:    RUN T UNTIL COMPLETES
   }
```

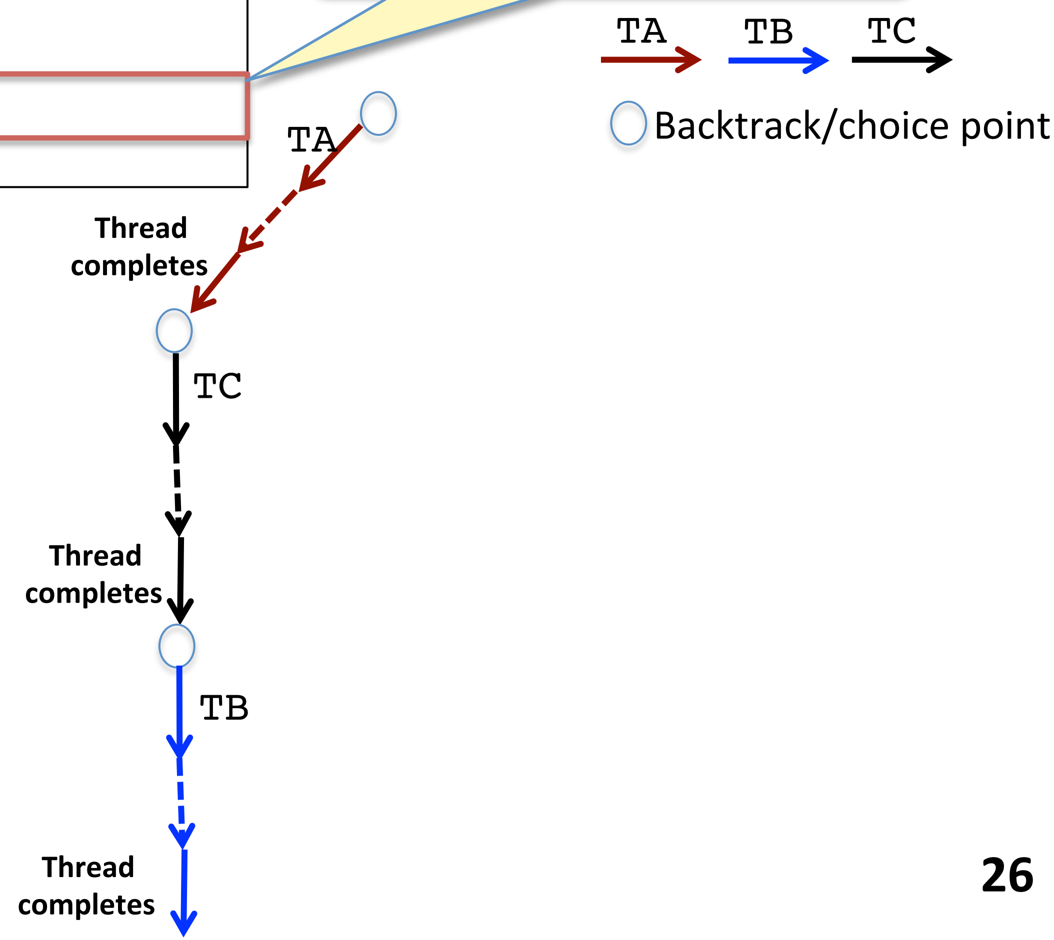
Pick one of the threads



First test: Run each thread sequentially until completion (No interleaving)

```
// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3:   WITH T IN [TA, TB, TC]

4:   RUN T UNTIL COMPLETES
   }
```

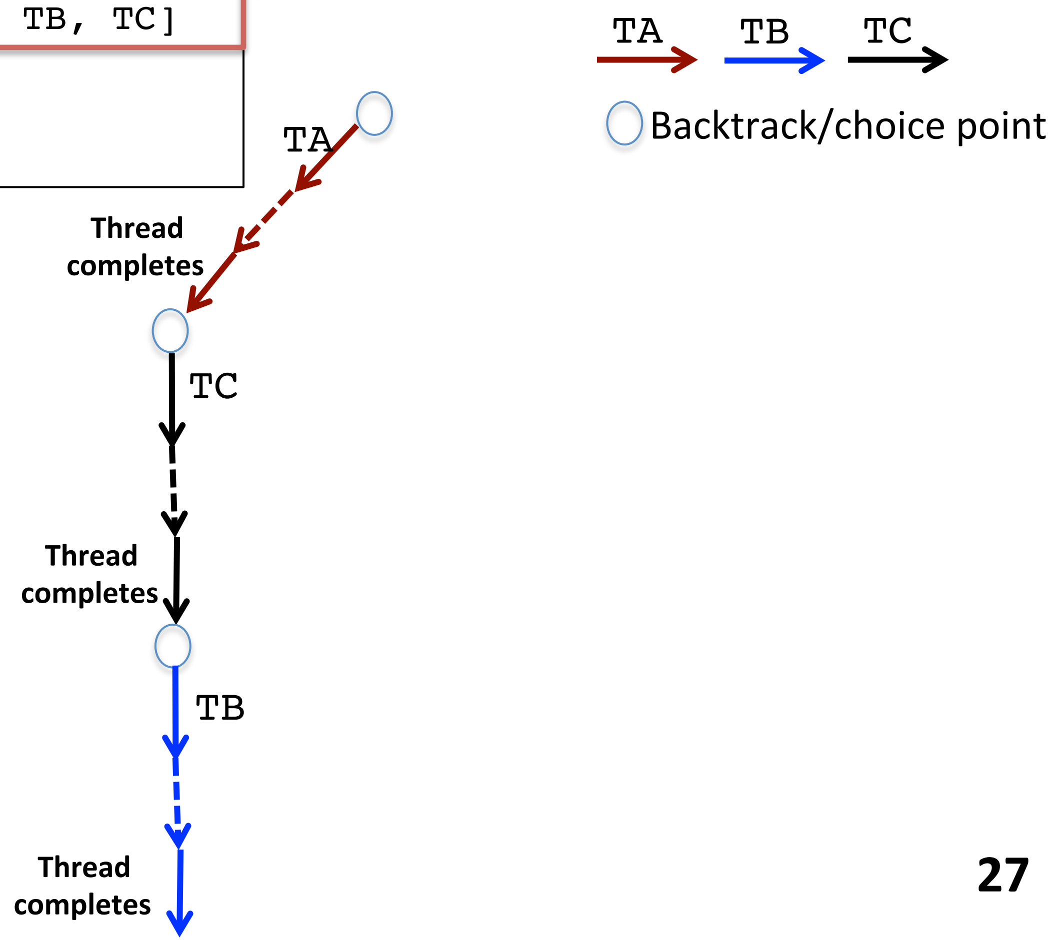
Run selected thread until it completes



First test: Run each thread sequentially until completion (No interleaving)

```
// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3:    BACKTRACK HERE WITH T IN [TA, TB, TC]

4:    RUN T UNTIL COMPLETES
   }
```

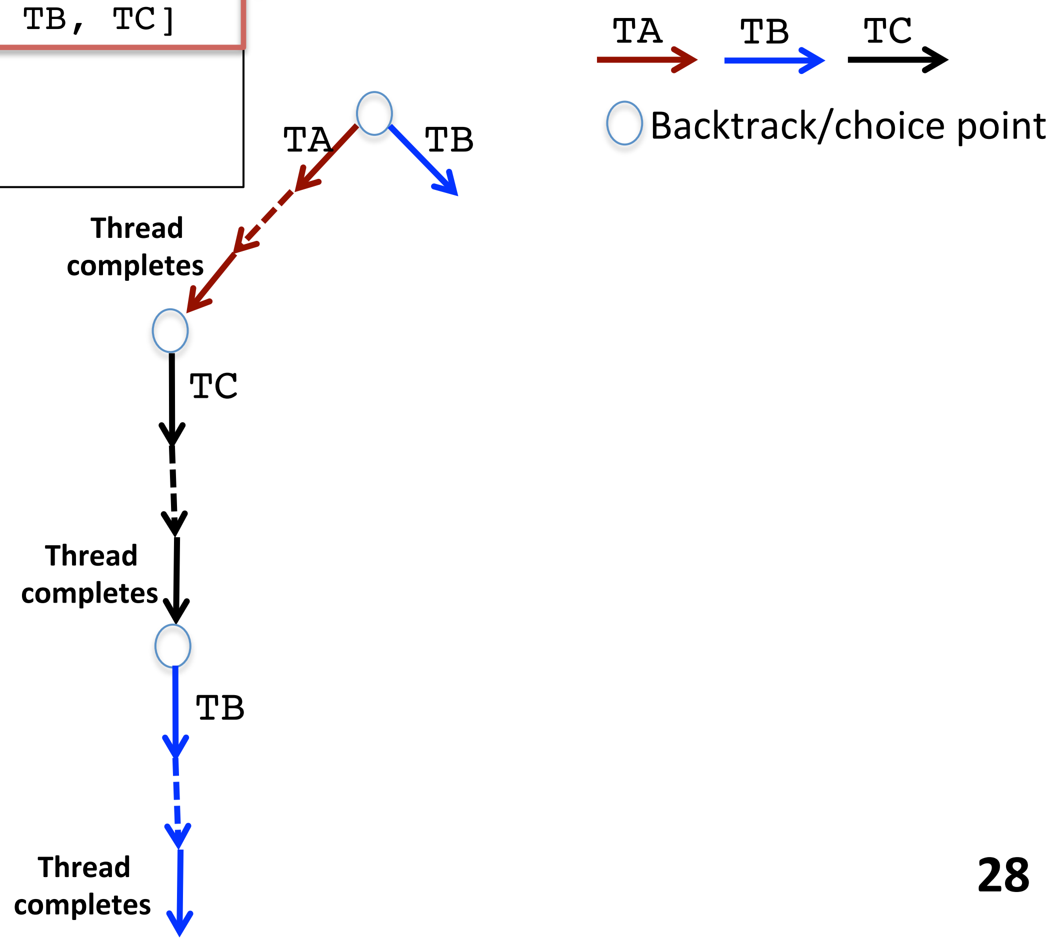
Pick a different thread when backtracked



First test: Run each thread sequentially until completion (No interleaving)

```
// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3:    BACKTRACK HERE WITH T IN [TA, TB, TC]

4:    RUN T UNTIL COMPLETES
   }
```

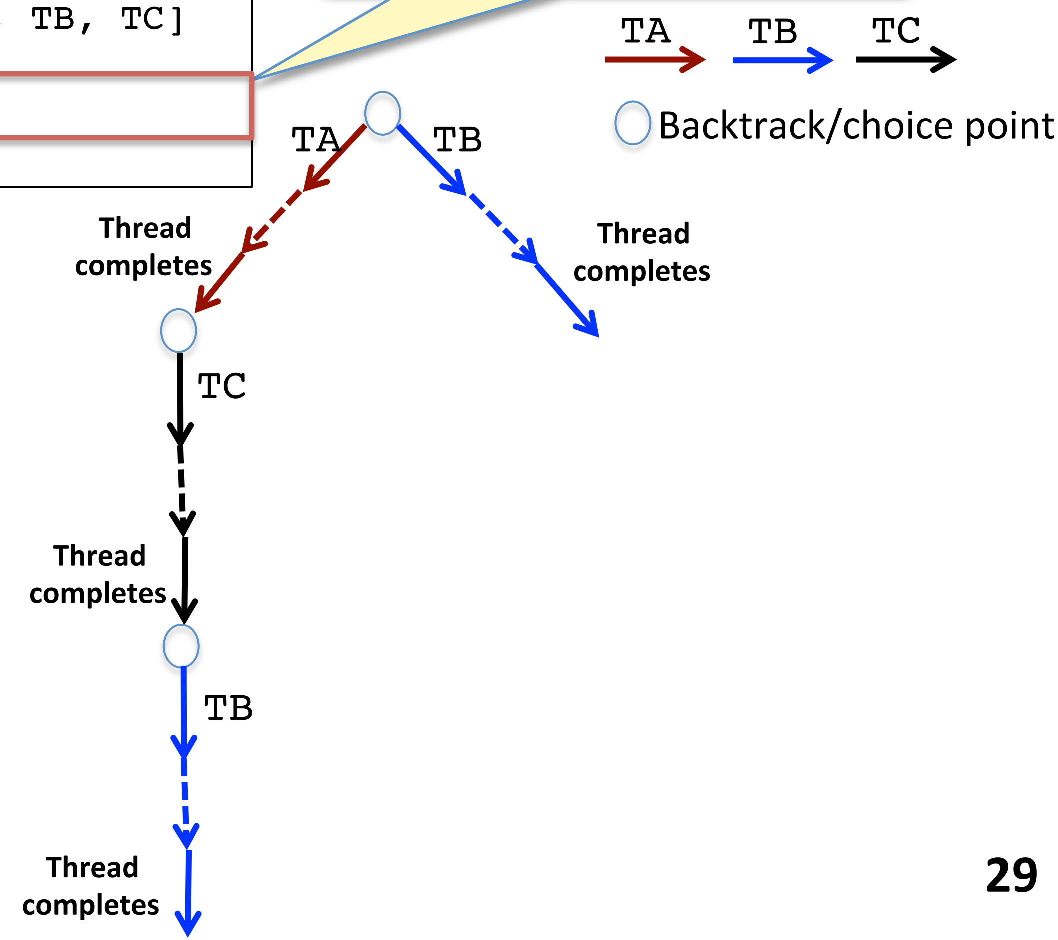
Pick a different thread when backtracked



First test: Run each thread sequentially until completion (No interleaving)

```
// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3:   BACKTRACK HERE WITH T IN [TA, TB, TC]

4:   RUN T UNTIL COMPLETES
   }
```

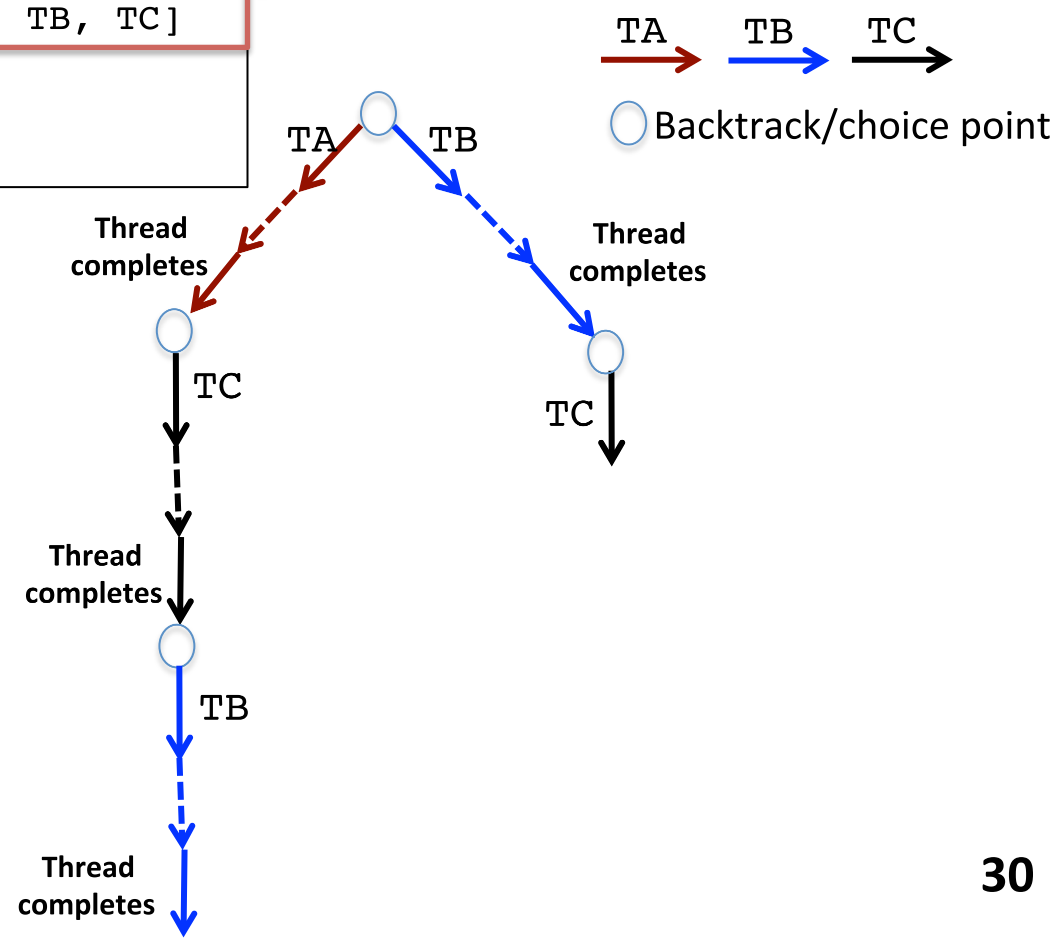
Run selected thread until it completes



First test: Run each thread sequentially until completion (No interleaving)

```
// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3:   BACKTRACK HERE WITH T IN [TA, TB, TC]

4:   RUN T UNTIL COMPLETES
   }
```

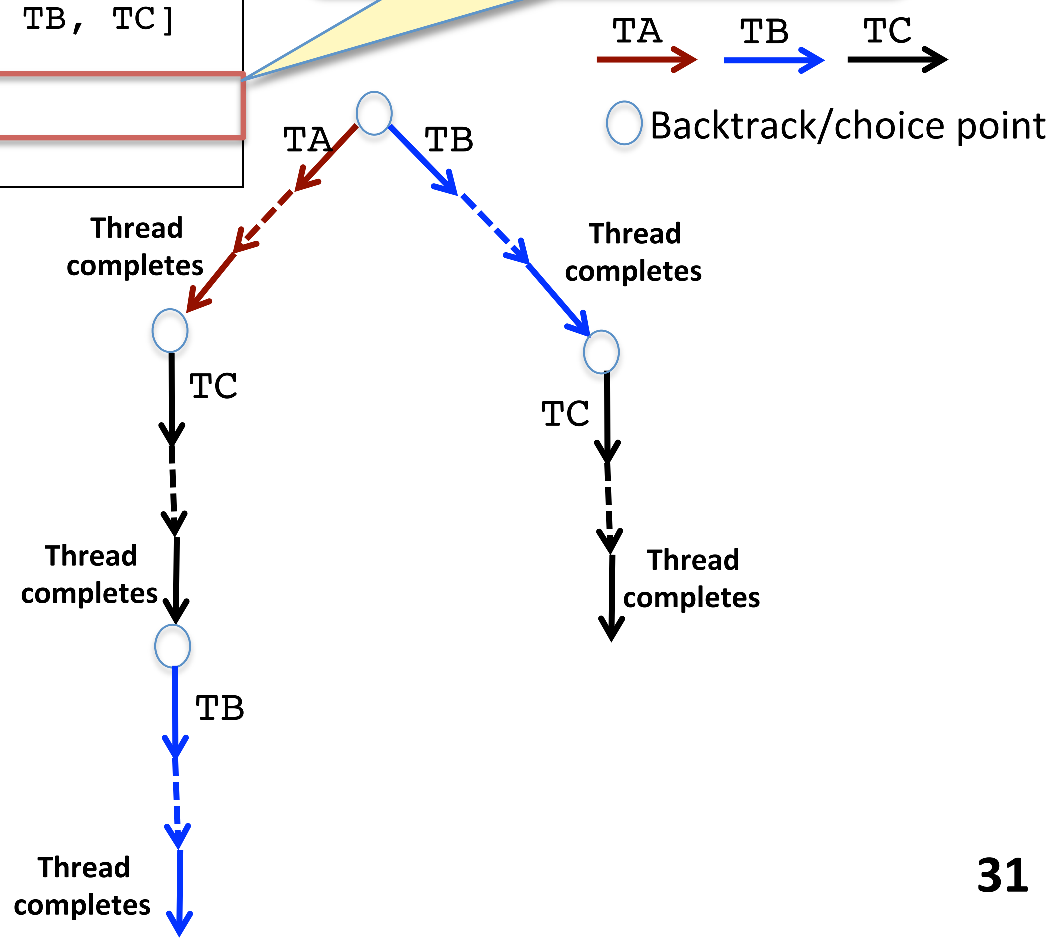
Pick a different thread when backtracked



First test: Run each thread sequentially until completion (No interleaving)

```
// Test in Concurrit DSL

TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

LOOP UNTIL TA, TB, TC COMPLETE {

  BACKTRACK HERE WITH T IN [TA, TB, TC]

  RUN T UNTIL COMPLETES
}
```

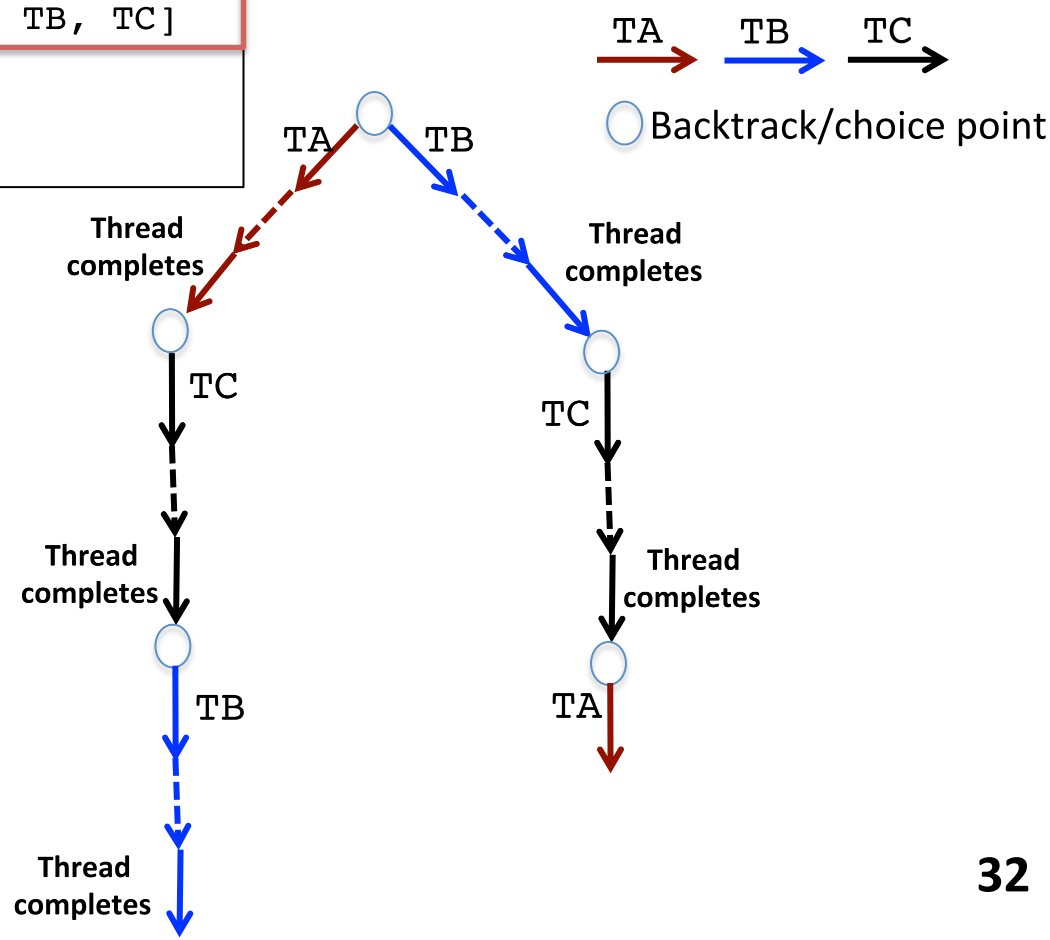
Run selected thread until it completes



First test: Run each thread sequentially until completion (No interleaving)

```
// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3:    BACKTRACK HERE WITH T IN [TA, TB, TC]

4:    RUN T UNTIL COMPLETES
   }
```

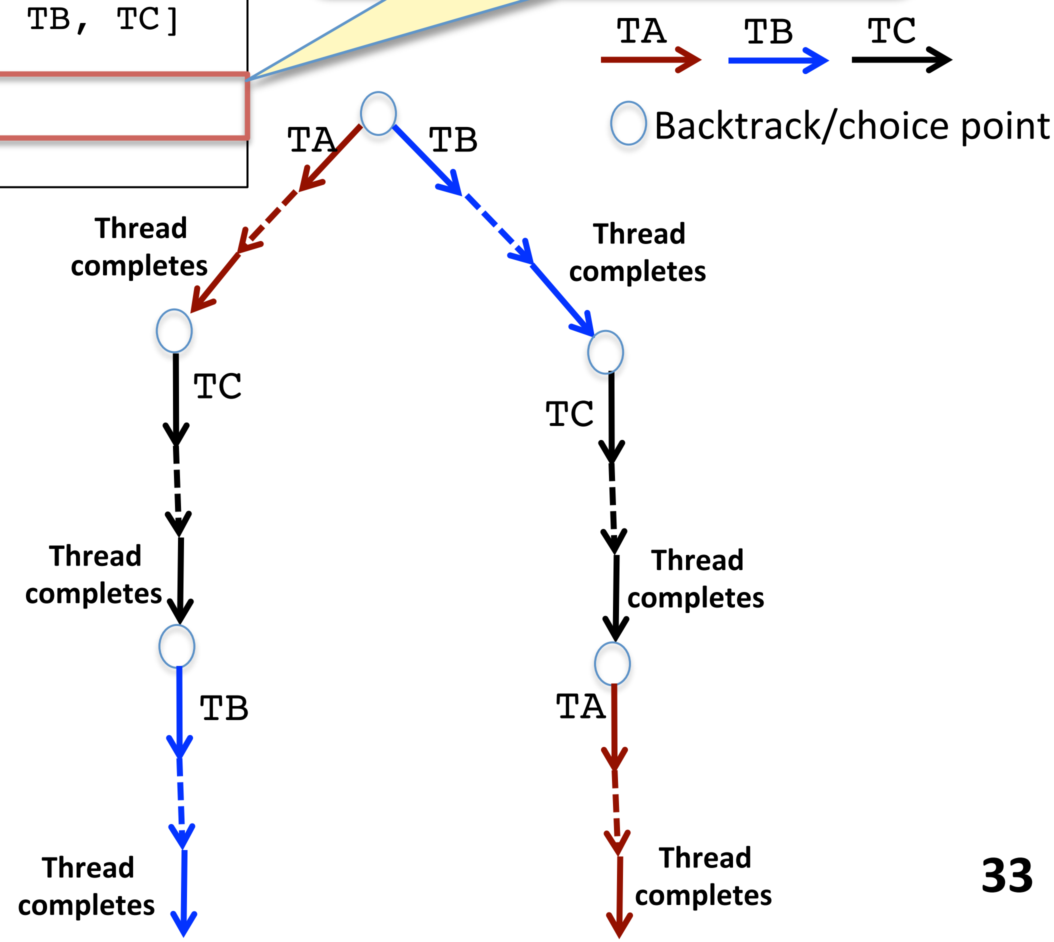
Pick a different thread when backtracked



First test: Run each thread sequentially until completion (No interleaving)

```
// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3:   BACKTRACK HERE WITH T IN [TA, TB, TC]

4:   RUN T UNTIL COMPLETES
   }
```

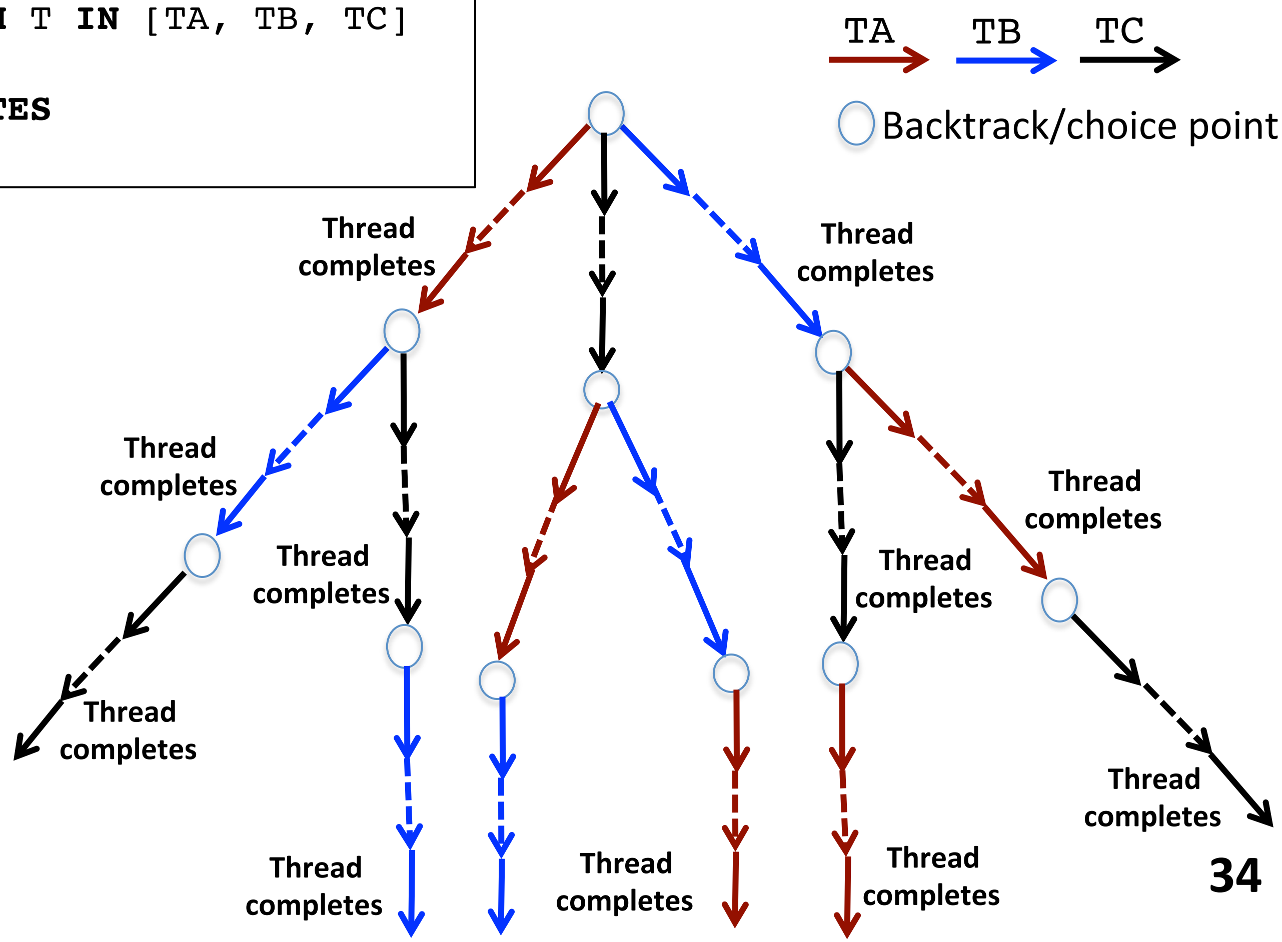
Run selected thread until it completes



First test: Run each thread sequentially until completion (No interleaving)

```
// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3:    BACKTRACK HERE WITH T IN [TA, TB, TC]

4:    RUN T UNTIL COMPLETES
   }
```

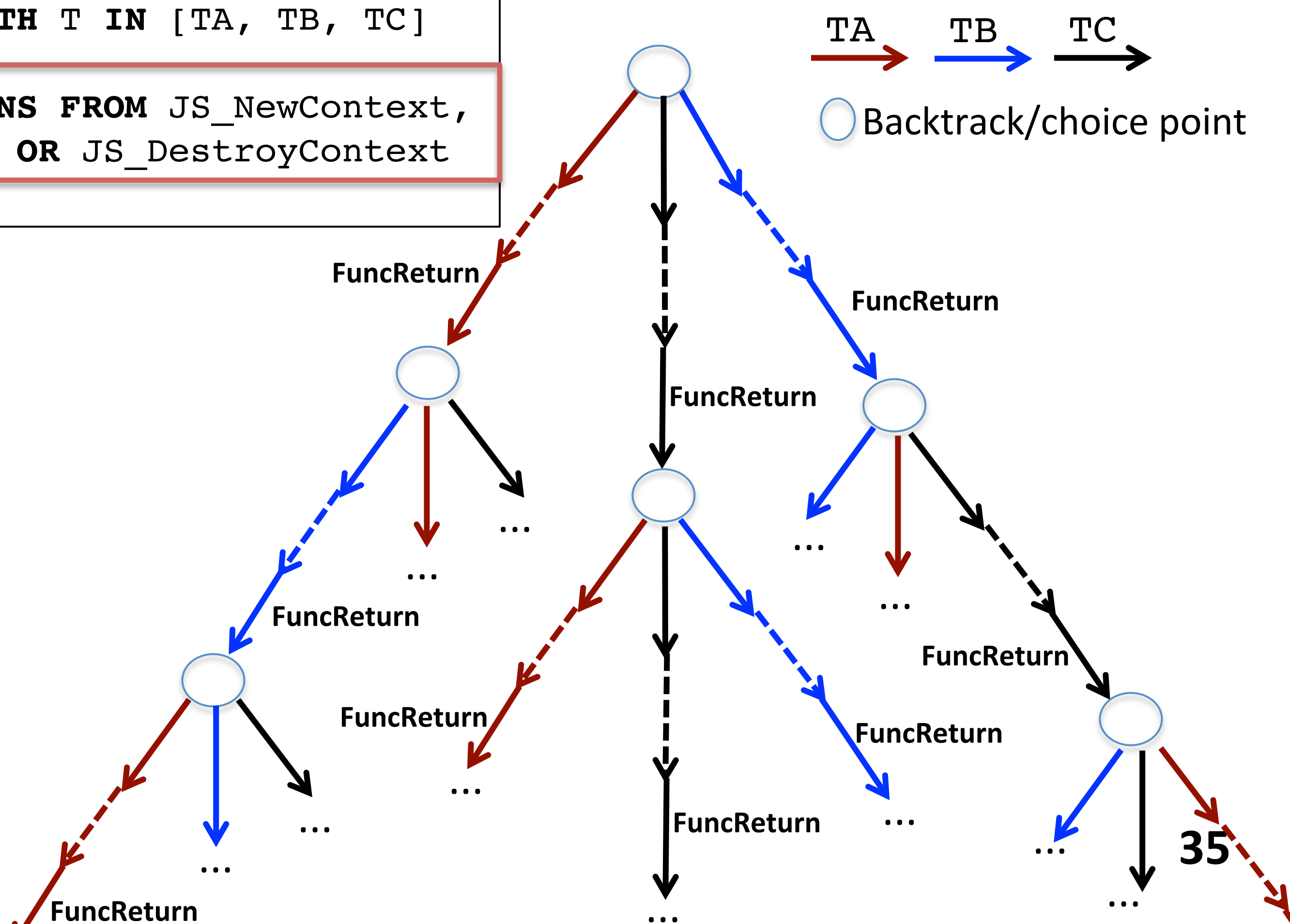
Result:
6 schedules
No assertion failure!



Second test: Run each thread sequentially until it returns from function

```
// Test in Concurrit DSL
1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()
2: LOOP UNTIL TA, TB, TC COMPLETE {
3:     BACKTRACK HERE WITH T IN [TA, TB, TC]
4:     RUN T UNTIL RETURNS FROM JS_NewContext,
        JS_BeginRequest, OR JS_DestroyContext
}
```

Result:
< 50 schedules
No assertion failure!



Outline

- Bug report for Mozilla SpiderMonkey
- Write tests in Concurrit DSL to generate buggy schedule
  - **Simple schedules**
    - Few schedules **BUT** not manifesting bug
  - **All schedules**
    - Manifests bug **BUT** too many schedules
  - **Target buggy schedule in bug report**
    - Few schedules **AND** manifests bug

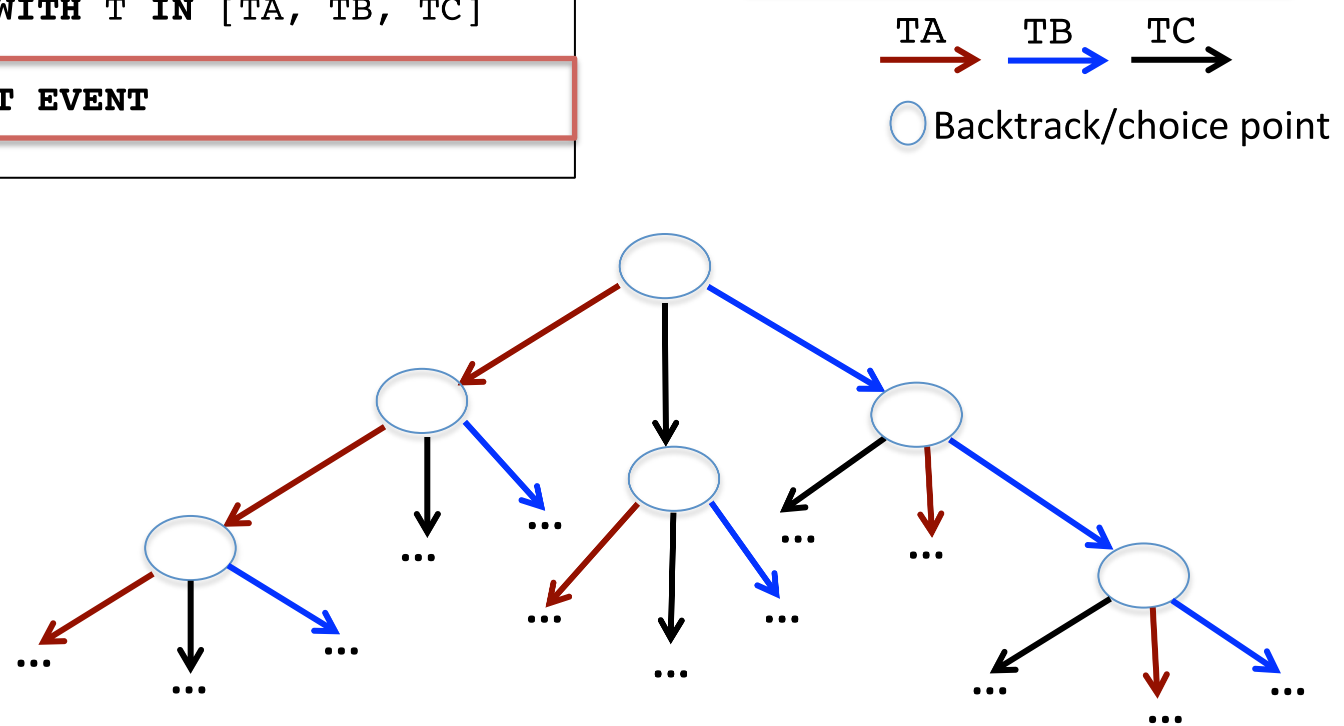
First test: Run each thread sequentially until completion (No interleaving)

```
// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3:    BACKTRACK HERE WITH T IN [TA, TB, TC]

4:    RUN T UNTIL COMPLETES
}
```

Generate all thread schedules

```
// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3:     BACKTRACK HERE WITH T IN [TA, TB, TC]

4:     RUN T UNTIL NEXT EVENT
}
```

Result:
> 100,000 schedules
Assertion failure
after a night!



What is different from (traditional) model checking?

1. **Cannot control/instrument everything!**
   - Must tolerate uncontrolled non-determinism
   - Backtrack-and-replay-prefix may fail

2. **Localize the search**
   - To particular functions, operations, states, ...

BUT: Can express traditional model checking algorithms

- If every operation can be controlled
- Feasible for small programs, data structures, ...

Outline

- Bug report for Mozilla SpiderMonkey
- Write tests in Concurrit DSL to generate buggy schedule
  - **Simple schedules**
    - Few schedules **BUT** not manifesting bug
  - **All schedules**
    - Manifests bug **BUT** too many schedules
  - **Target buggy schedule in bug report**
    - Few schedules **AND** manifests bug

Possible buggy schedule from bug report

Igor Bukanov 2009-03-09 17:47:12 PDT Comment 5 [reply] [-] [reply] [-]

```
At least one problem that I can see from the code is that js_GC does the check:

if (rt->state != JSRTS_UP && gckind != GC_LAST_CONTEXT)
    return;

outside the GC lock. Now suppose there are 3 threads, A, B, C. Threads A and B
calls js_DestroyContext and thread C calls js_NewContext.

First thread A rem
the last one so th
latter skips the a

B                                                                  m the
r                                                                   runs
th

At this stage the                                                   that
runs the GC and re                                                 ate is
DOWN.

Now the thread C enters the picture. It discovers under the GC lock in
js_NewContext that the newly allocated context is the first one. Since
rt->state is DOWN, it releases the GC lock and starts the first context
initialization procedure. That procedure includes the allocation of the initial
atoms and it will happen when the thread A runs the GC. This may lead precisely
to the first stack trace from the comment 4.
```

Thread C

Threads A, B

```
static void * testfunc(void *ignore

    JSContext *cx = JS_NewContext(rt, 0x1000);
    if (cx) {
        JS_BeginRequest(cx);
        JS_DestroyContext(cx);
    }

    return NULL;
}
```

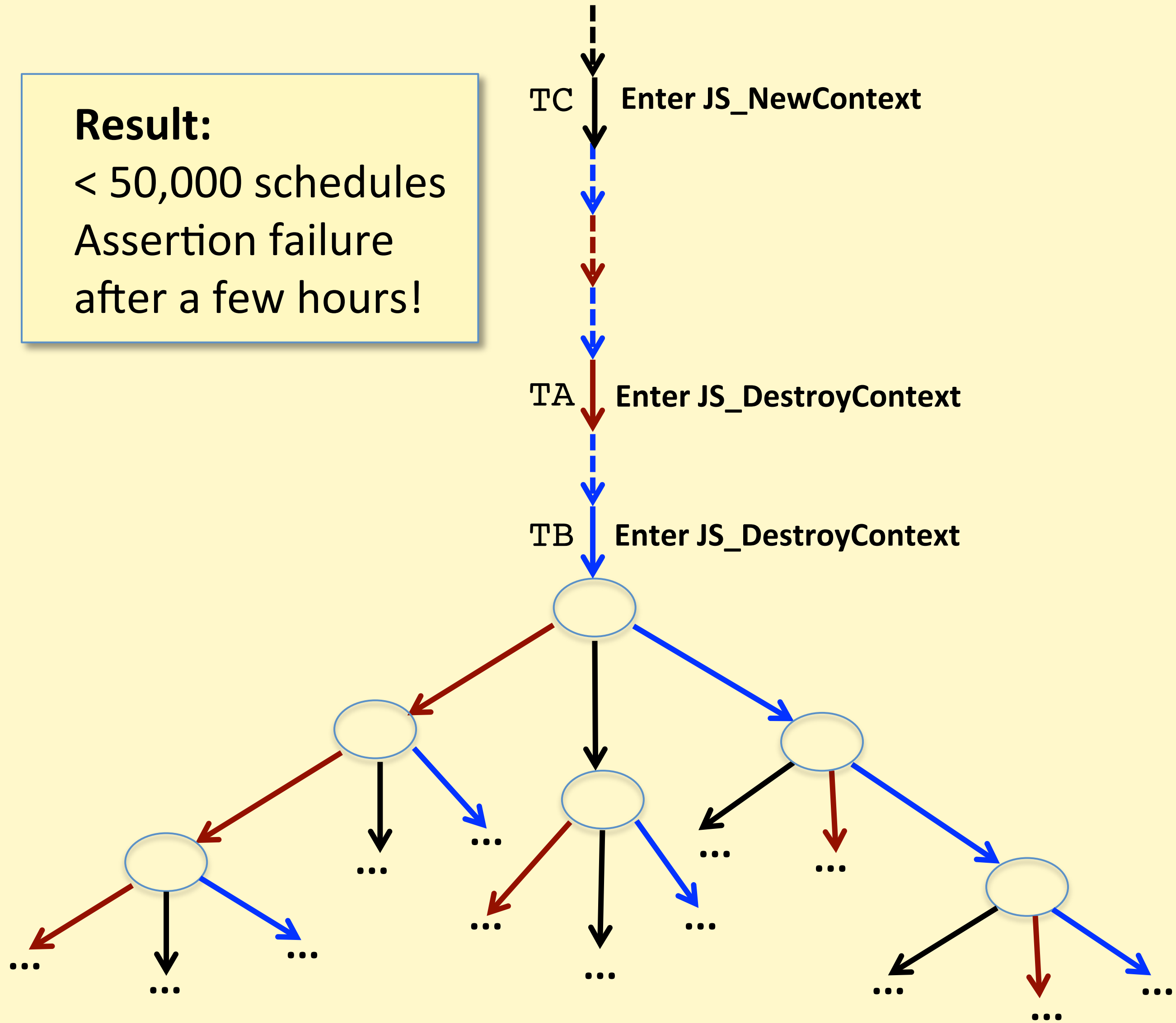
Generate all thread schedules

```
// Test in Concurrit DSL

TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

LOOP UNTIL TA, TB, TC COMPLETE {

    BACKTRACK HERE WITH T IN [TA, TB, TC]

    RUN T UNTIL NEXT EVENT
}
```

Exploiting programmer's insights about bug

```
// Test in Concurr
TC = WAIT_FOR_TI
TA = WAIT_FOR_DI
TB = WAIT_FOR_DI
LOOP UNTIL TA, '
    BACKTRACK HEI
    RUN T UNTIL I
}
```

Result:
< 50,000 schedules
Assertion failure
after a few hours!



What is different from (traditional) model checking?

1. **Cannot control/instrument everything!**
   - Must tolerate uncontrolled non-determinism
   - Backtrack-and-replay-prefix may fail

2. **Localize the search**
   - To particular functions, operations, states, ...

BUT: Can express traditional model checking algorithms
- If every operation can be controlled
- Feasible for small programs, data structures, ...

Possible buggy schedule from bug report

- Shared variables involved in the bug:
  - **rt->state, rt->gcLock, rt->gcThread**
- Reschedule threads when accessing them.

outside the GC lock. Now suppose there are 3 threads, A, B, C. Threads A and B calls js_DestroyContext and thread C calls js_NewContext.

First thread A removes its context from the runtime list. That context is not the last one so thread does not touch rt->state and eventually calls js_GC. The latter skips the above check and tries to to take the GC lock.

Before this moment the thread B takes the lock, removes its context from the runtime list, discovers that it is the last, sets rt->state to LANDING, runs the-last-context-cleanup, runs the GC and then sets rt->state to DOWN.

At this stage the thread A gets the GC lock, setup itself as the thread that runs the GC and releases the GC lock to proceed with the GC when rt->state is DOWN.

Now the thread C enters the picture. It discovers under the GC lock in js_NewContext that the newly allocated context is the first one. Since rt->state is DOWN, it releases the GC lock and starts the first context initialization procedure. That procedure includes the allocation of the initial atoms and it will happen when the thread A runs the GC. This may lead precisely to the first stack trace from the comment 4.

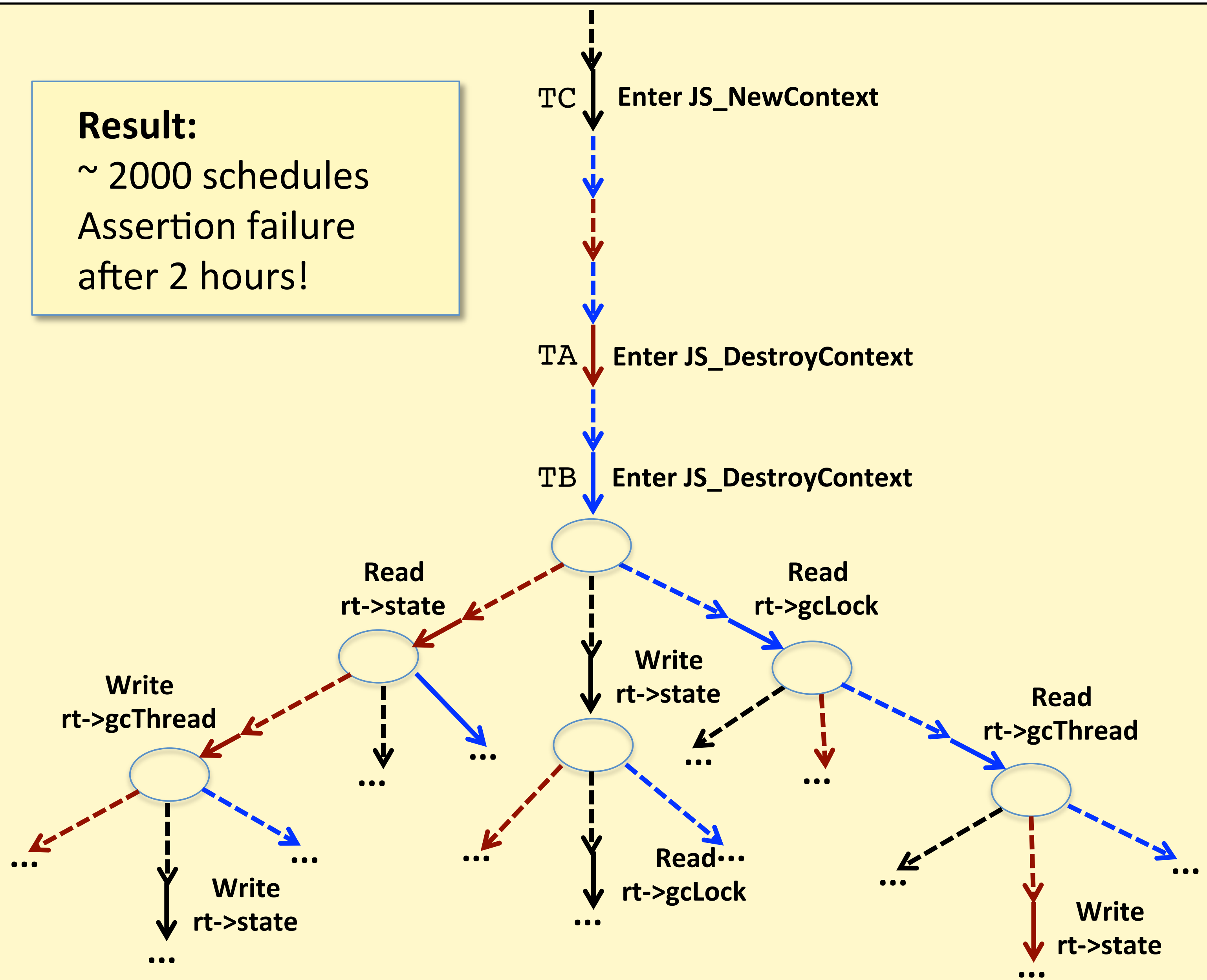
Exploiting programmer's insights about bug

```
// Test in Concurrit DSL

TC = WAIT_FOR_THREAD(ENTERS JS_NewContext)

TA = WAIT_FOR_DISTINCT_THREAD(ENTERS JS_DestroyContext)

TB = WAIT_FOR_DISTINCT_THREAD(ENTERS JS_DestroyContext)

LOOP UNTIL TA, TB, TC COMPLETE {

   BACKTRACK HERE WITH T IN [TA, TB, TC]

   RUN T UNTIL NEXT EVENT
}
```

Exploiting programmer's insights about bug

```
// Test in Co
1: TC = WAIT_
2: TA = WAIT_
3: TB = WAIT_
4: LOOP UNTIL
5:     BACKTRA
6:     RUN T U
    }
```

Result:
~ 2000 schedules
Assertion failure
after 2 hours!



Possible buggy schedule from bug report

Igor Bukanov 2009-03-09 17:47:12 PDT Comment 5 [reply] [-] [reply] [-]

At least one problem that I can see from the code is that js_GC does the check:

```
if (rt->state != JSRTS_UP && gckind != GC_LAST_CONTEXT)
    return;
```

outside the GC lock. Now suppose there are
calls js_DestroyContext and thread C calls

Setup

First thread A removes its context from the runtime list. That context is not the last one so thread does not touch rt->state and eventually calls js_GC. The latter skips the above check and tries to take the GC lock.

Before this moment the thread B takes the
runtime list, discovers that it is the last
the-last-context-cleanup, runs the GC and

Fixed, known schedule for threads A and B

At this stage the thread A gets the GC lock, setup itself as the thread that runs the GC and releases the GC lock to proceed with the GC when rt->state is DOWN.

Now the thread C enters the picture. It dis
js_NewContext that the newly allocated cont
rt->state is DOWN, it releases the GC lock
initialization procedure. That procedure in
atoms and it will happen when the thread A
to the first stack trace from the comment 4

Unknown schedule for A and C

Final test

```
// Test in Concurrit DSL

TC = WAIT_FOR_THREAD(
                 ENTERS JS_NewContext)

TA = WAIT_FOR_DISTINCT_THREAD(
                 ENTERS JS_DestroyContext)

TB = WAIT_FOR_DISTINCT_THREAD(
                 ENTERS JS_DestroyContext)

RUN TA UNTIL READS &rt->state IN js_GC

RUN TB UNTIL COMPLETES

RUN TA UNTIL WRITES &rt->gcThread IN js_GC

LOOP UNTIL TA, TC COMPLETE {

   BACKTRACK HERE WITH T IN [TA, TC]

   RUN T UNTIL READS OR WRITES MEMORY
}
```

Igor Bukanov 2009-03-09 17:47:12 PDT Comment

```
At least one problem that I can see from the code is that js

if (rt->state != JSRTS_UP && gckind != GC_LAST_CONTEXT)
    return;

outside the GC lock. Now
calls js_DestroyContext 

First thread A removes its context from the runtime list. Th
the last one so thread does not touch rt->state and eventual
latter skips the above c

Before this moment the t
runtime list, discovers
the-last-context-cleanup

At this stage the thread A gets the GC lock, setup itself as
runs the GC and releases the GC lock to proceed with the GC
DOWN.

Now the thread C enters
js_NewContext that the n
rt->state is DOWN, it re
initialization procedure
atoms and it will happen
to the first stack trace from the comment 4.
```

Setup

Fixed, known schedule for threads A and B

Unknown schedule for A and C

Final test

Igor Bukanov 2009-03-09 17:47:12 PDT Comment 5 [reply] [-] [reply] [-]

At least one problem that I can see from the code is that js_GC does the check:

```
if (rt->state != JSRTS_UP && gckind != GC_LAST_CONTEXT)
    return;
```

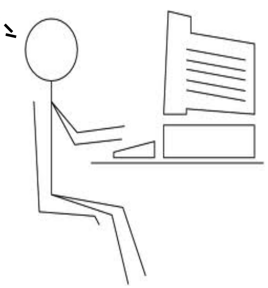
outside the GC lock. Now suppose there are 3 threads, A, B, C. Threads A and B calls js_DestroyContext and thread C calls js_NewContext.

First thread A removes its context from the runtime list. That context is not the last one so thread does not touch rt->state and eventually calls js_GC. The latter skips the above check and tries to to take the GC lock.

Before this moment the thread B takes the lock, removes its context from the runtime list, discovers that it is the last, sets rt->state to LANDING, runs the-last-context-cleanup, runs the GC and then sets rt->state to DOWN.

At this stage the thread A gets the GC lock, setup itself as the thread that runs the GC and releases the GC lock to proceed with the GC when rt->state is DOWN.

Now the thread C enters the picture. It discovers under the GC lock in js_NewContext that the newly allocated context is the first one. Since rt->state is DOWN, it releases the GC lock and starts the first context initialization procedure. That procedure includes the allocation of the initial atoms and it will happen when the thread A runs the GC. This may lead precisely to the first stack trace from the comment 4.



Software Under Test

. . . . . .

. . . . . .

+

```
// Test in Concurrit DSL

TC = WAIT_FOR_THREAD(
                ENTERS JS_NewContext)

TA = WAIT_FOR_DISTINCT_THREAD(
                ENTERS JS_DestroyContext)

TB = WAIT_FOR_DISTINCT_THREAD(
                ENTERS JS_DestroyContext)

RUN TA UNTIL READS &rt->state IN js_GC

RUN TB UNTIL COMPLETES

RUN TA UNTIL WRITES &rt->gcThread IN js_GC

LOOP UNTIL TA, TC COMPLETE {

   BACKTRACK HERE WITH T IN [TA, TC]

   RUN T UNTIL READS OR WRITES MEMORY
}
```

Triggers assertion failure in < 30 thread schedules

(Add to regression test suit)

Implementation/Evaluation

- **Implementation**: DSL embedded in C++
  - Prototype: *http://code.google.com/p/concurrit/*
- Wrote concise tests for (real/manually-inserted) bugs in well-known benchmarks
  - Reproducing bugs
    using < 20 lines of DSL code, after < 30 schedules
  - **Inspect:** bbuf, bzip2, pbzip2, pfscan
  - **PARSEC:** dedup, streamcluster
  - **RADBench:** SpiderMonkey 1/2, Mozilla NSPR 1/2/3
    - **Ongoing:** Apache httpd, Chromium, Memcached
- Can write various model checking algorithms (next slide)

Default search policies

```
EXPLORE_ALL_SCHEDULES(THREADS) {
  LOOP UNTIL ALL THREADS COMPLETE {
    BACKTRACK HERE WITH T IN THREADS
    RUN T UNTIL NEXT EVENT
  }
}
```

```
EXPLORE_TWO_CONTEXT_BOUNDED_SCHEDULES(THREADS) {
  BACKTRACK HERE WITH T1 IN THREADS
  BACKTRACK HERE LOOP NONDETERMINISTICALLY {
    RUN T1 UNTIL NEXT EVENT
  }

  BACKTRACK HERE WITH T2 IN [THREADS EXCEPT T1]
  BACKTRACK HERE LOOP NONDETERMINISTICALLY {
    RUN T2 UNTIL NEXT EVENT
  }

  EXPLORE_THREADS_UNTIL_COMPLETION(THREADS)
}
```

```
EXPLORE_THREADS_UNTIL_COMPLETION(THREADS) {
  LOOP UNTIL ALL THREADS COMPLETE {
    BACKTRACK HERE WITH T IN THREADS
    RUN T UNTIL COMPLETION
  }
}
```

Positioning Concurrit: Usage scenarios

Insert sleeps:
Explore one schedule

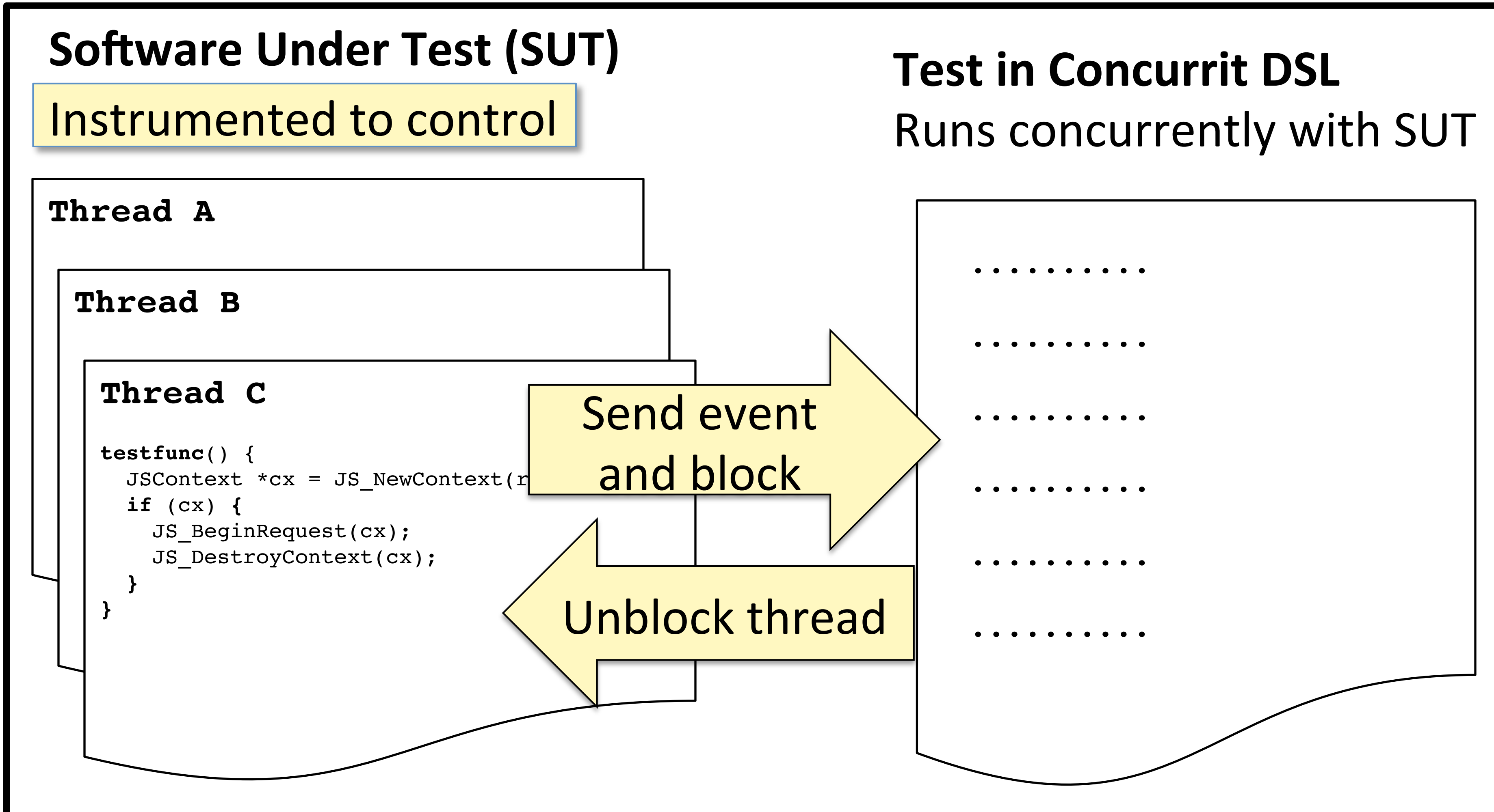
Model checking:
Explore all schedules

Concurrit

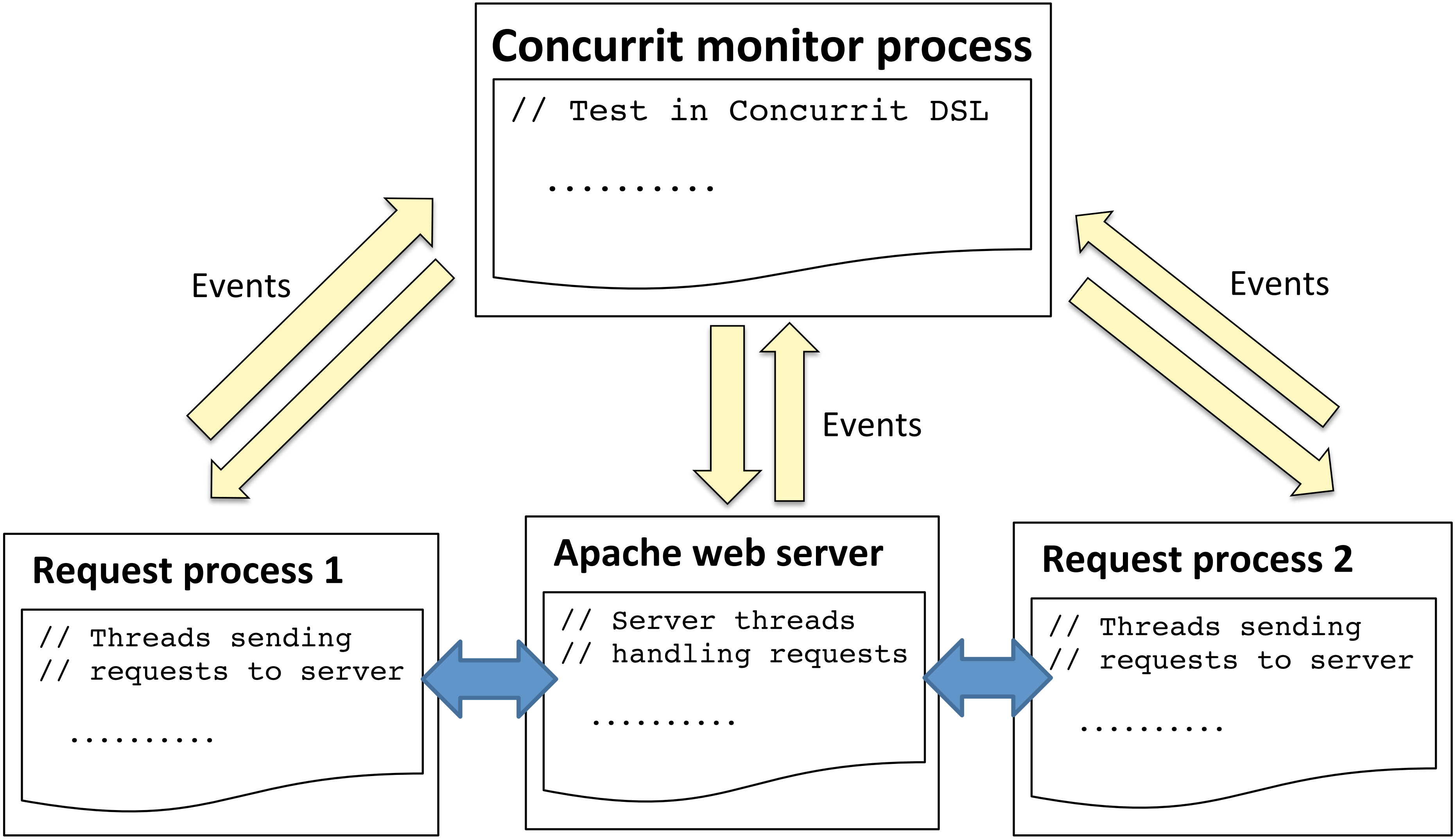
Control user-defined events
- Portable, testing library
- Manual instrumentation
- Generate exact/perfect schedule

Control all operations
- Exhaustive testing tool
- Automated instrumentation
- Generate all schedules

Unit-testing programs with Concurrit



Ongoing work: Integration testing
Controlling multi-process/distributed applications



Approaches to controlling thread schedules

Test run: A set of executions of the test driver.

Success: At least one execution in the run hits the bug.