The C/C++ Concurrency

and Effective Stateless Model Checking for C/C++ Concurrency

presented by Vladimír Štill

5th March 2018

Concurrency in C and C++

Pre C11/C++11

- no support for concurrency in the language and standard library
- no standard-defined behaviour for parallel programs

C11/C++11

- standard defines behaviour of parallel programs
- standard library defines:
  - thread manipulation: threads, futures
  - synchronization: mutexes, condition variables, **atomics**

not-atomic/unordered

- cannot be used for synchronization
- two threads can access non-atomic variable at the same time and at least one of them modifies it – *data race* $\rightarrow$ *undefined behaviour*

not-atomic/unordered

- cannot be used for synchronization
- two threads can access non-atomic variable at the same time and at least one of them modifies it – *data race* $\rightarrow$ *undefined behaviour*
- `volatile` does not change anything

C11 Variable Access Modes in Parallel Programs II

```c
#include <stdatomic.h> // <atomic> in C++
#include <pthread.h>
#include <assert.h>

_Atomic int x; // or atomic_int x;
               // std::atomic< int > x; in C++
void *worker1( void *_ ) {
    atomic_fetch_add( &x, 1 ); // atomic, synchronizes
    return NULL;               // C++: x.fetch_add( 1 );
}
int main() {
    pthread_t t1;
    pthread_create( &t1, NULL, worker1, NULL );
    ++x; // atomic, synchronizes
    pthread_join( t1, NULL );
    assert( x == 2 ); // OK
}
```

atomic access

- for variables declared with `atomic_*` type or with `_Atomic` qualifier
- by default all access is fully atomic
  - all accesses to all atomic variables have a single total order in each run
  - i.e. interleaving semantics

atomic access

- for variables declared with `atomic_*` type or with `_Atomic` qualifier
- by default all access is fully atomic
  - all accesses to all atomic variables have a single total order in each run
  - i.e. interleaving semantics
- atomic ordering can be relaxed

Memory Model

- semantics of memory access
  - of hardware: `x86`/`x86_64`, ARM, POWER
  - of programming language: C11/C++11, Java, C#
- what primitives/instructions are supported (fences, CAS, RMW...)
- what synchronization is guaranteed
- what reordering can be observed

C11/C++11 Memory Model

- designed to allow high-performance even of very relaxed hardware (POWER/ARM)
- different levels of synchronization: **memory order**

```
x.fetch_add( 1, std::memory_order_relaxed ); // C++
```

- ranges from *almost no guarantees* to *interleaving semantics*

Memory Orders I

- **relaxed** – only guarantees that operations *on the single location* are ordered
  - safe counters
  - termination indication (from signal handler, . . .)
  - can be reordered with other operations

Memory Orders II

- **release** – for write operations
- **acquire** – for read operations
- together ensure all previous writes will be visible when a release write is observed by an acquire read (simplified)
  - *previous* is anything happening control-flow-before in the writing thread

```
int x;
std::atomic< bool > f;

void t1() {
  x = 42;
  f.store( true, std::memory_order_release );
}
void t2() {
  while ( !f.load( std::memory_order_acquire ) ) { }
  assert( x == 42 ); // OK
}
```

Memory Orders III

- **acquire+release** – for read-modify-write/compare-and-swap operations
- **sequential consistency** – all SC operations are in a total order
  - the default
  - atomics in other programming languages are usually SC

Memory Orders III

- **acquire+release** – for read-modify-write/compare-and-swap operations
- **sequential consistency** – all SC operations are in a total order
  - the default
  - atomics in other programming languages are usually SC
- SC can be very expensive
- not all data structures need SC
  - acquire/release usually sufficient for queues
  - relaxed for just counting/flagging objects
- mutexes synchronize as SC

Effective Stateless Model Checking for C/C++ Concurrency

Michalis Kokologiannakis

Ori Lahav

Konstantinos Sagonas

Viktor Vafeiadis

Prelude: The Two Main Analysis Approaches

Operational-Semantics Based Analysis

- how program executes on an abstract machine?
- interleavings
- instruction reordering simulation
- tools: DIVINE, CBMC, Nidhugg, ...

Axiomatic-Semantic Based Analysis

- where is a read allowed to take value from?
- what reordering is allowed?
- executions graphs
- tools: Herd, RCMC, ...

Stateless Model Checking

- for parallel programs, often in real-world languages
- originally based on operational (interleaving) semantics
- explores state space
- does not store closed set

Stateless Model Checking

- for parallel programs, often in real-world languages
- originally based on operational (interleaving) semantics
- explores state space
- does not store closed set
- cannot handle non-terminating programs
- can explore some states repeatedly

Stateless Model Checking

- for parallel programs, often in real-world languages
- originally based on operational (interleaving) semantics
- explores state space
- does not store closed set
- cannot handle non-terminating programs
- can explore some states repeatedly

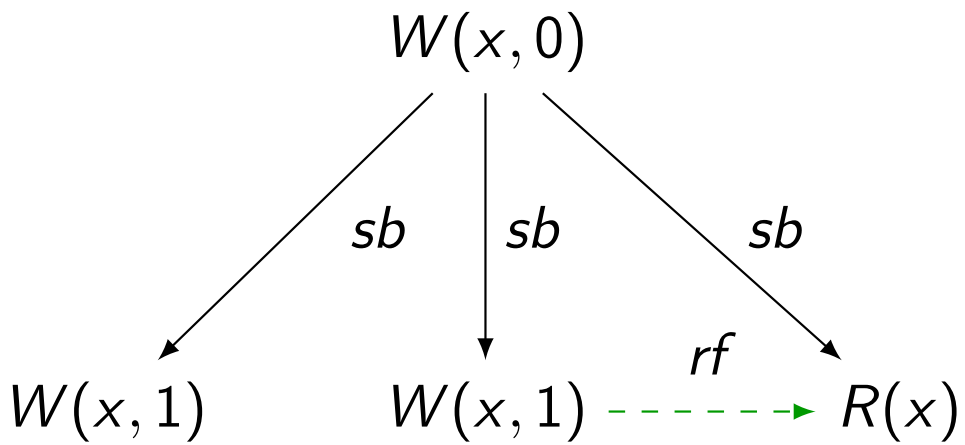
Dynamic Partial Order Reduction

- eliminates some traces, reduces redundant exploration
- usually based on some notion of equivalence of traces
- there are optimal techniques for certain equivalences

Execution Graphs I

the technique from the paper is based on execution graphs, not interleaving & DPOR

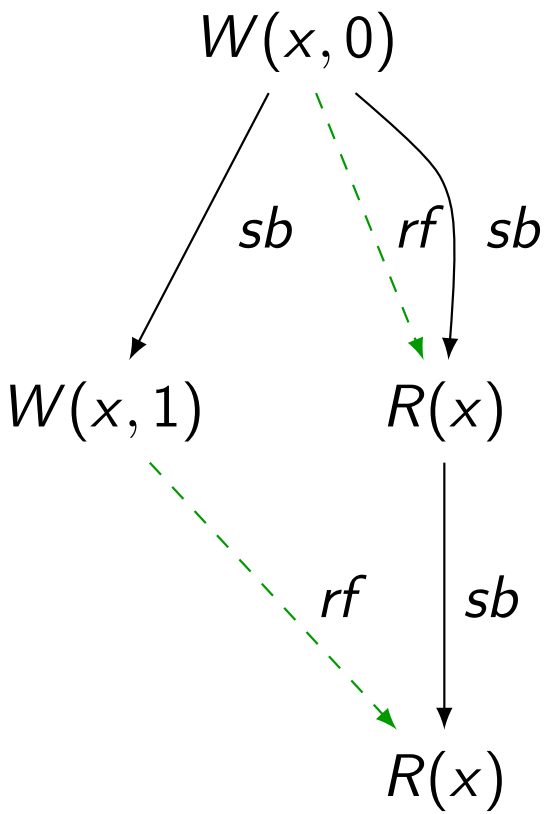
- records memory operations
- values of writes
- origin of value for reads
- dependencies

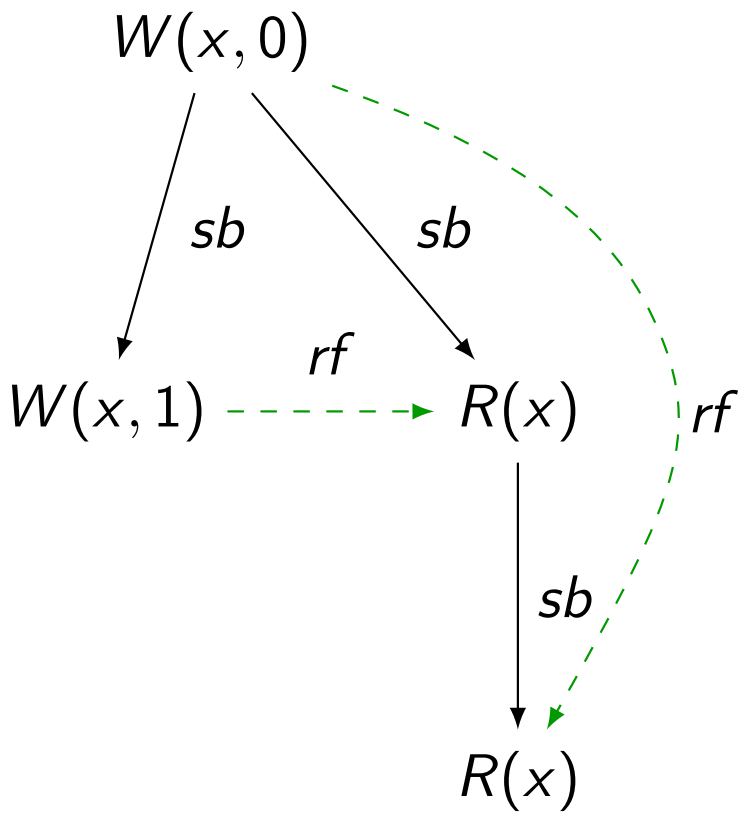


- sb = sequenced before; rf = read-from

Execution Graphs II

not all execution graphs are valid = *consistent*





Execution Graph Generation

```
a = x;    ||    x = 1;
b = x;    ||
```

$$W(x, 0)$$

- initialization

Execution Graph Generation

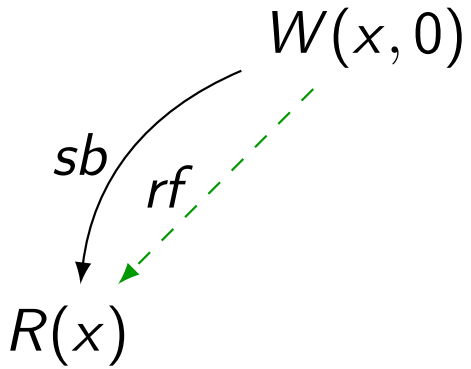
```
a = x;    ||    x = 1;
b = x;    ||
```

$$W(x, 0)$$

- in what order should actions be added?

Execution Graph Generation

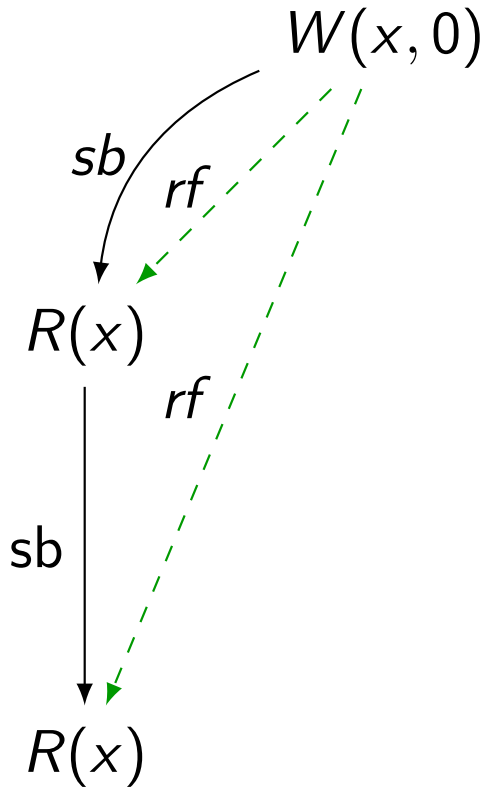
```
a = x;    ||    x = 1;
b = x;    ||
```



- add `a = x`

Execution Graph Generation

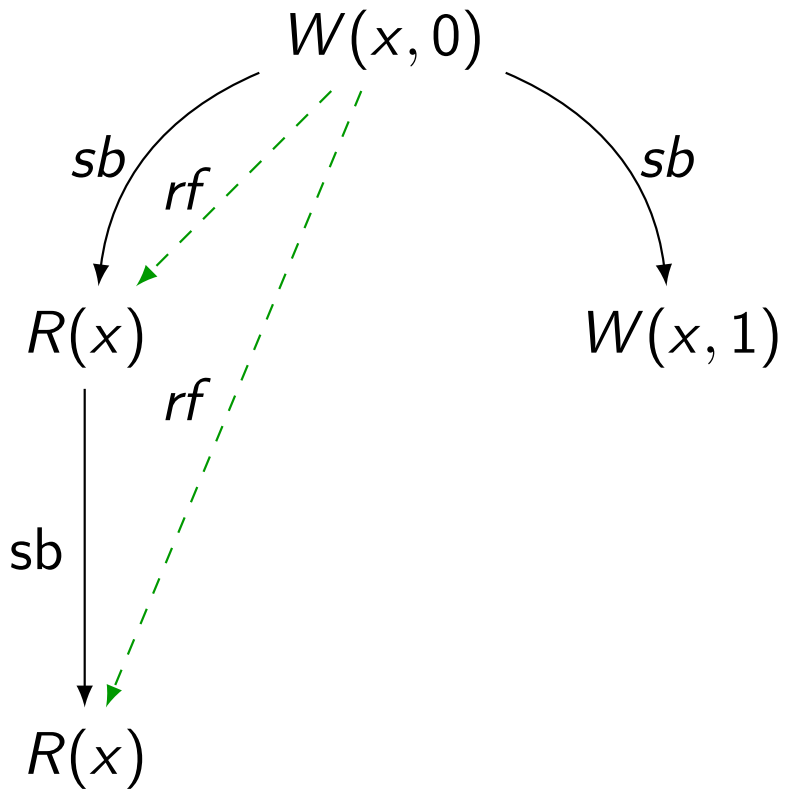
```
a = x;    ||    x = 1;
b = x;    ||
```



- add `b = x`

Execution Graph Generation

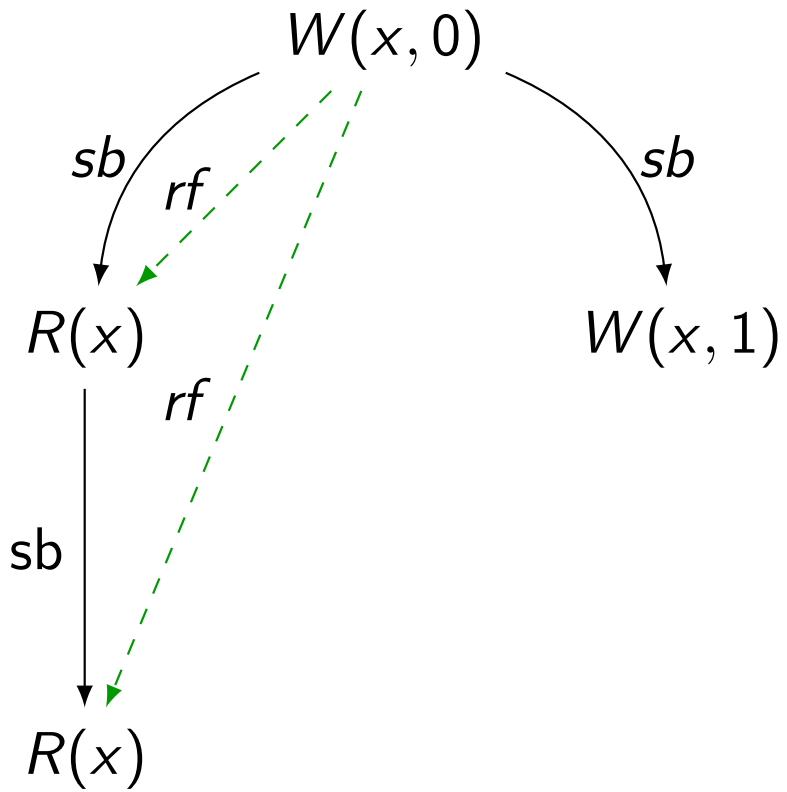
```
a = x;    ||    x = 1;
b = x;    ||
```



- add `x = 1`

Execution Graph Generation

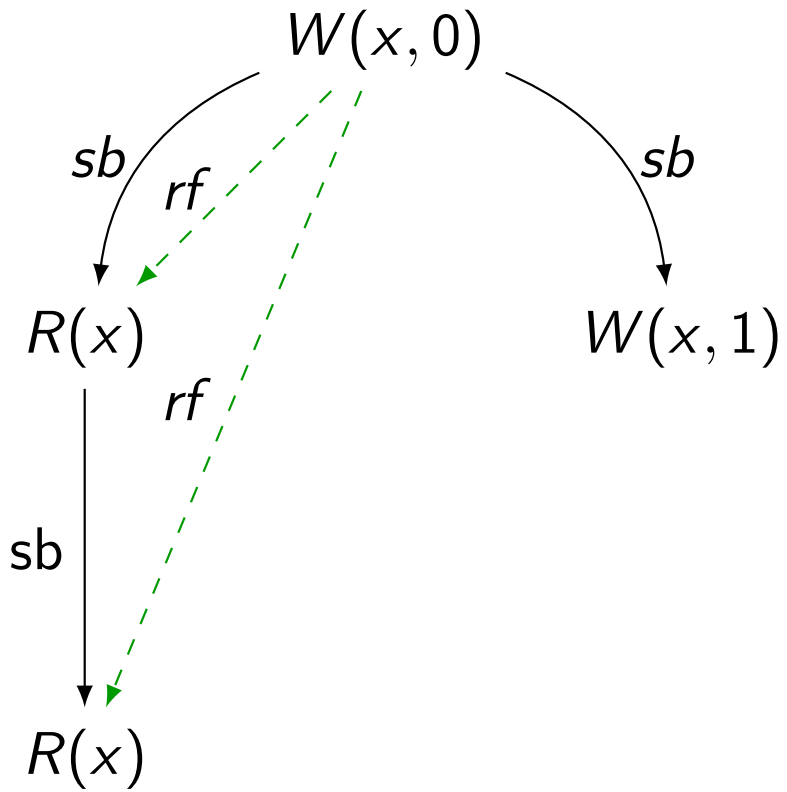
```
a = x;    ||    x = 1;
b = x;    ||
```



- explore again adding `x = 1` first?

Execution Graph Generation

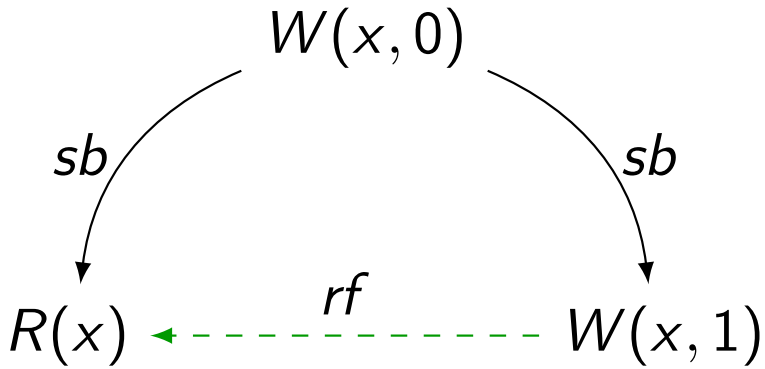
```
a = x;    ||    x = 1;
b = x;    ||
```



- no. . . revisit reads

Execution Graph Generation

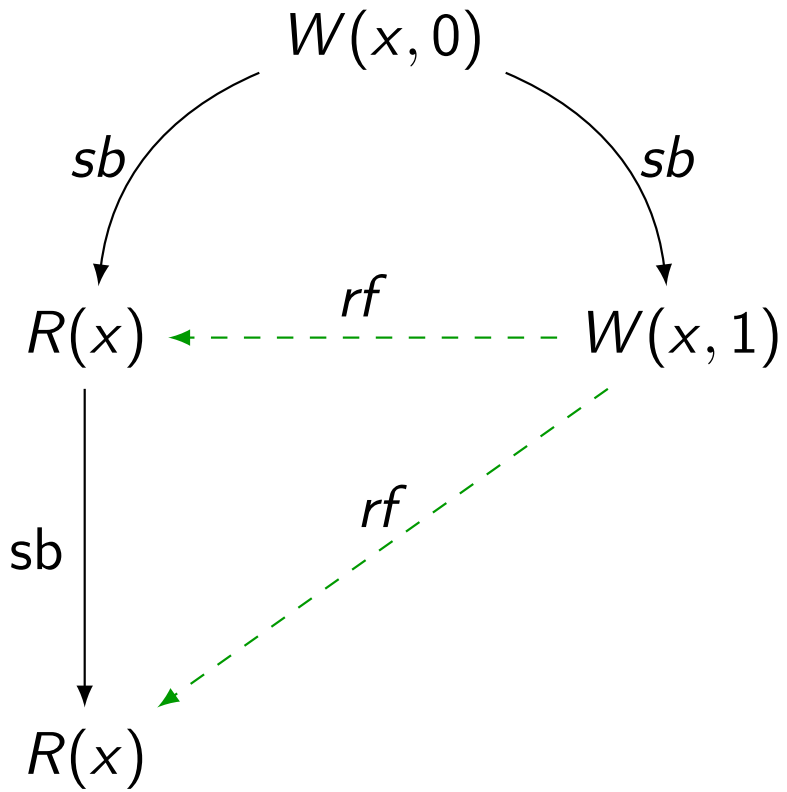
```
a = x;    ||    x = 1;
b = x;    ||
```



- option 1: revisit `a = x`

Execution Graph Generation

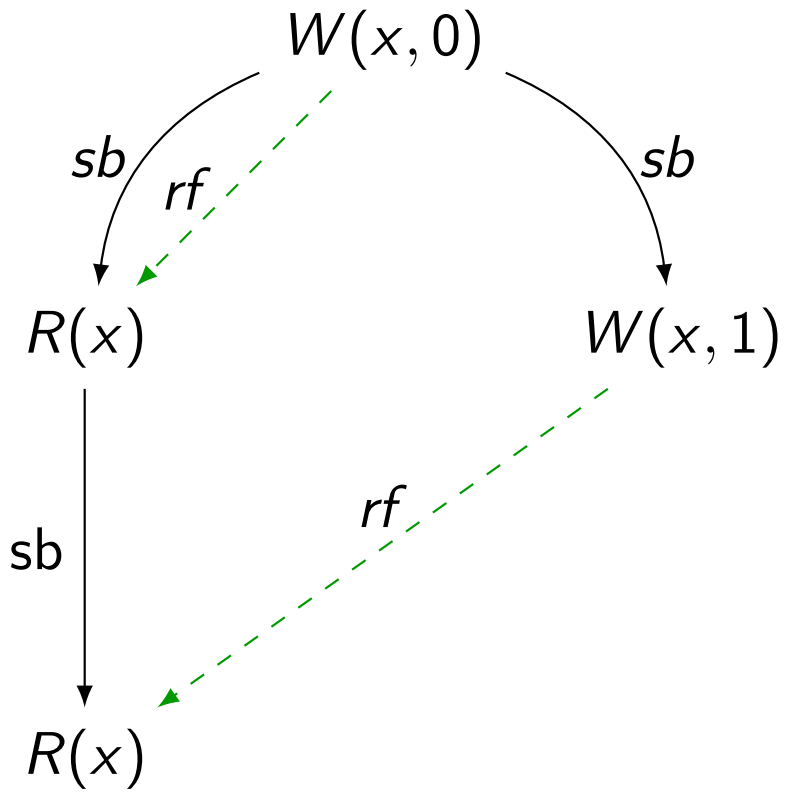
```
a = x;    ||    x = 1;
b = x;    ||
```



- option 1: revisit `a = x` + add `b = x`

Execution Graph Generation

```
a = x;    ||    x = 1;
b = x;    ||
```



- option 2: revisit `b = x`

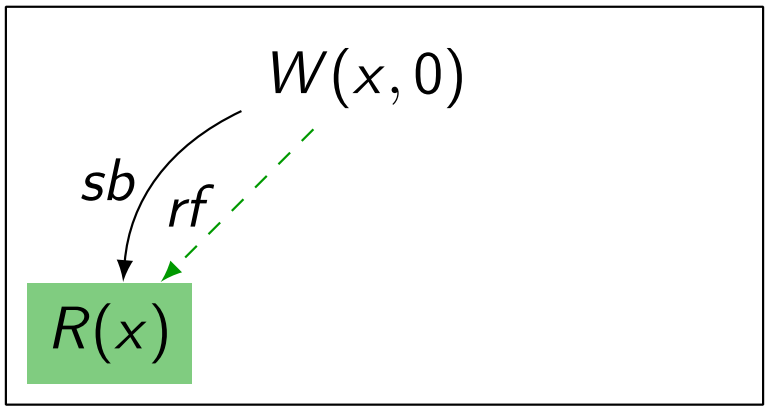
Exploration Requirements

generate all consistent graphs and

1. don't generate any inconsistent graphs
2. don't generate any graph multiple times
3. don't store generated graphs

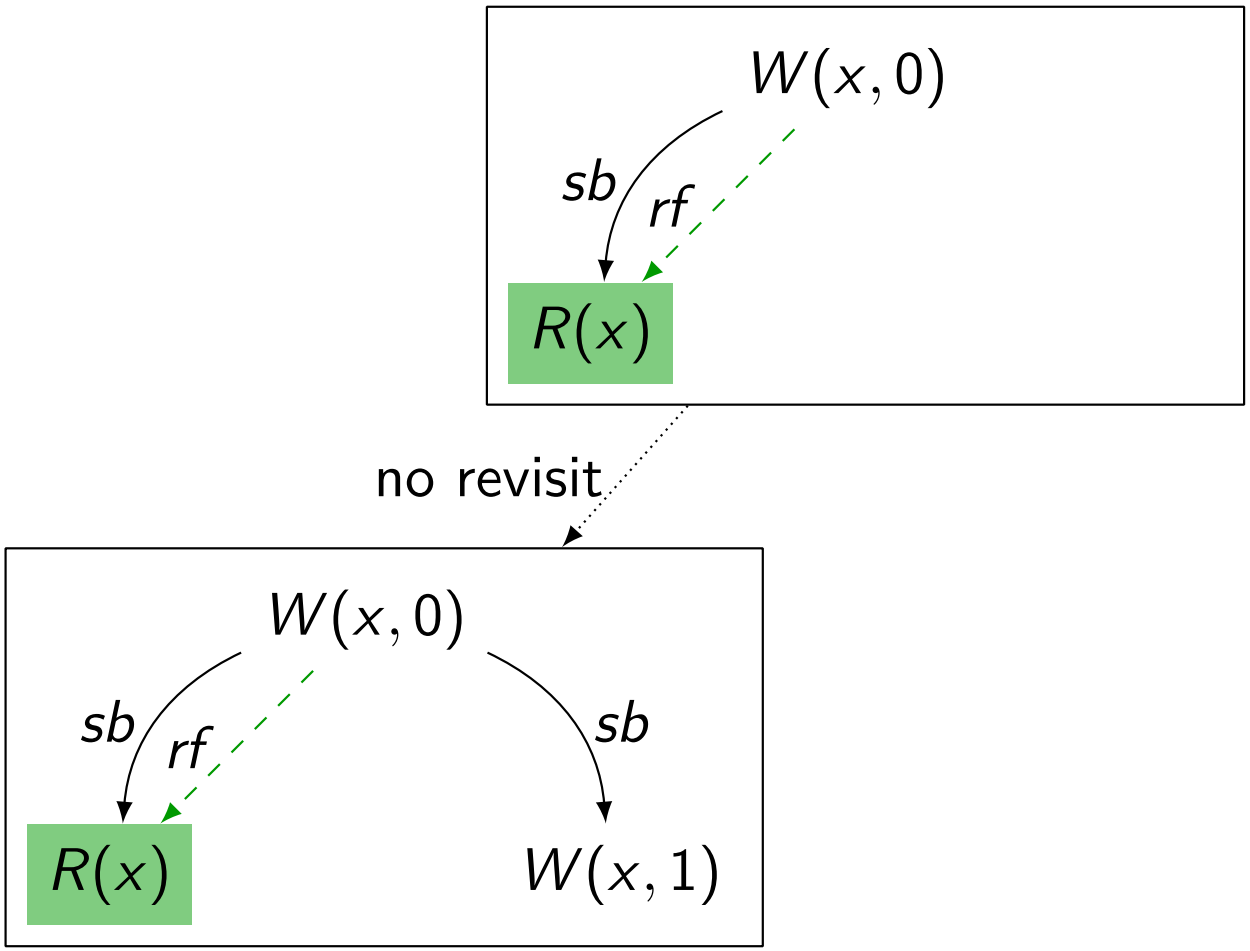
Revisitable Reads

- revisiting reads all the time causes redundant explorations
- only one instance of read needs to be revisitable



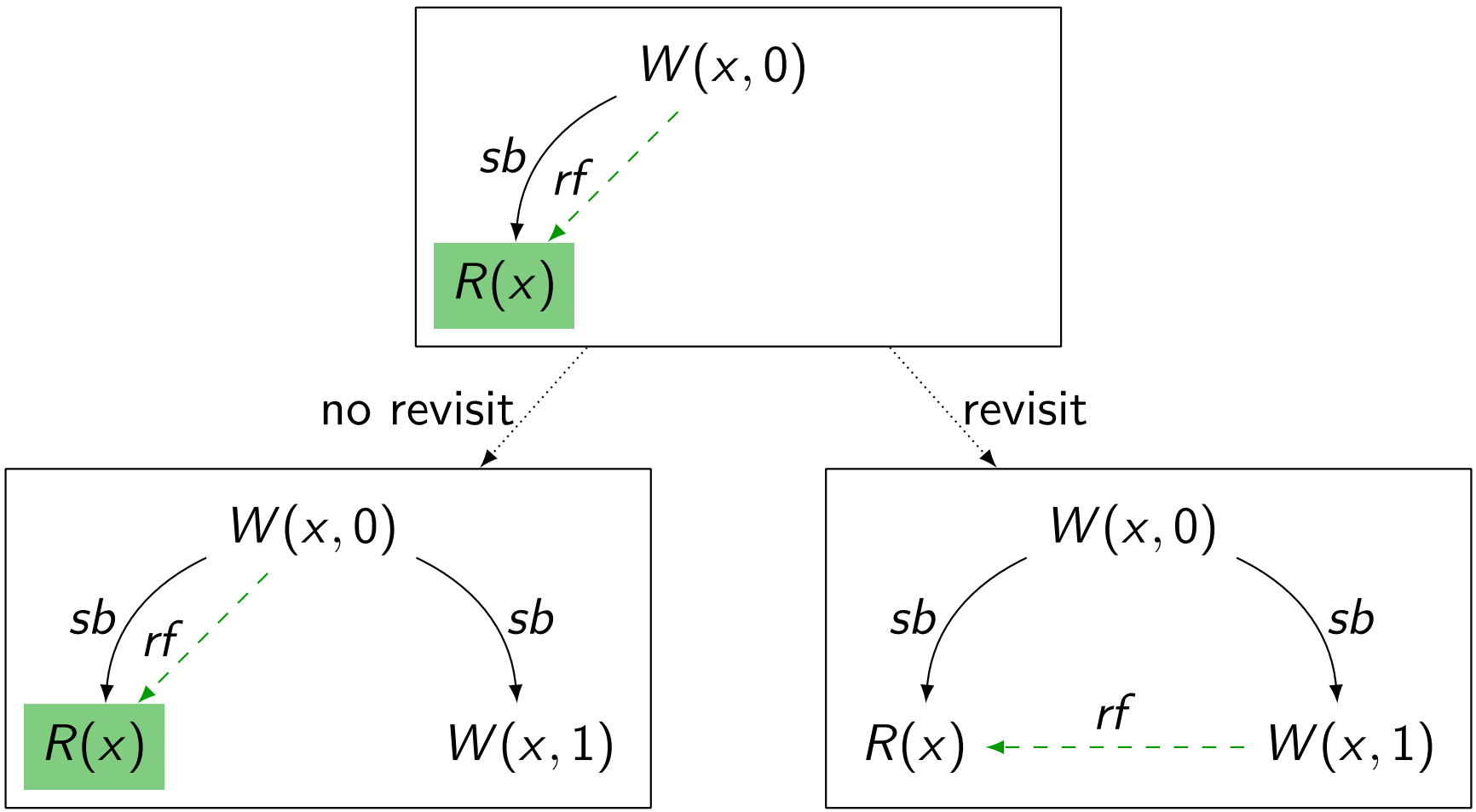
Revisitable Reads

- revisiting reads all the time causes redundant explorations
- only one instance of read needs to be revisitable



Revisitable Reads

- revisiting reads all the time causes redundant explorations
- only one instance of read needs to be revisitable



Efficient Execution Graph Generation

generate all consistent graphs and

1. don't generate any inconsistent graphs
2. don't generate any graph multiple times
3. don't store generated graphs

what is needed?

- (non)revisitable reads

Efficient Execution Graph Generation

generate all consistent graphs and

1. don't generate any inconsistent graphs
2. don't generate any graph multiple times
3. don't store generated graphs

what is needed?

- (non)revisitable reads
- prefix closed execution graphs – all prefixes of consistent EG are also consistent
  - not really for C11/C++11 standard's memory model!
  - paper uses *repaired C11 memory model*

Efficient Execution Graph Generation

generate all consistent graphs and

1. don't generate any inconsistent graphs
2. don't generate any graph multiple times
3. don't store generated graphs

what is needed?

- (non)revisitable reads
- prefix closed execution graphs – all prefixes of consistent EG are also consistent
  - not really for C11/C++11 standard's memory model!
  - paper uses *repaired C11 memory model*
  - prefix closed in absence of RMW/CAS & SC atomics
  - special handling for these (can generate duplicities)

Results

`http://plv.mpi-sws.org/rcmc/paper.pdf`

Summary

- tool RCMC
- analysis of C under the repaired C11 memory model
- stateless model checking, building execution graphs
- loop unrolling to ensure termination