Commutativity Race Detection
Concepts, Algorithms and Open Problems

Martin Vechev
Department of Computer Science
ETH Zurich

ETH Zürich
Concurrency is Everywhere
Concurrency and Interference

Concurrency and Interference go together

Some interference is **good** (e.g., coordination)

Some interference is **bad** (e.g., causes errors)
Data Races

A generic notion of interference: data races

Data race:
two unordered operations from different threads access the same memory location, where one of the accesses is a write
Example of a Data Race

```java
Object o = new Object();

fork {
    o.talk = "Martin"
}

o.talk = "Marco"
```

1. two writes to `o.talk`
2. not ordered by program
Android Data Races
Scalable Race Detection for Android Applications, ACM OOPSLA 2015

Races cause memory leaks, crashes, null dereferences, bad UI

http://www.eventtracer.org/android
Dynamic Race Detection
20+ years of algorithms and optimizations

Lots of work on the subject:
improve asymptotic complexity, parallelization, sampling, hardware, optimizations, ...

**Guarantees:** proves program free of races for a given input state
Dynamic Race Detection
20+ years of algorithms and optimizations

Lots of work on the subject:
- improve asymptotic complexity
- parallelization
- sampling
- hardware
- optimizations, ...

Guarantees: proves program free of races for a given input state

Recommended reading:

**FastTrack**, ACM PLDI’09, S. Freund, C. Flanagan
**EventRacer**, ACM OOPSLA’13, V. Raychev, M. Sridharan, M. V.

MIT’s Cilk Race Detector for SP-graphs, ACM SPAA’97, M. Feng, C. Leiserson
**Race Detection for 2D partial orders**, ACM SPAA’15, D. Dimitrov, M. V., V. Sarkar

Practices

Theory
Trend:
Interference shifts to interfaces

Increasingly, low-level functionality captured inside high level objects
The gap

Classic data race detection
Read-write notion of conflict
Many optimizations

Interference at the interface

Current detectors ineffectively in handling real-world programs
Wanted

New kind of race detection
Higher level notion of conflict
New optimizations

Interference at the interface

How do we bridge the gap in an elegant and clean manner?
Commutativity Race Detection, ACM PLDI’14
Dimitar Dimitrov, Veselin Raychev, Eric Koskinen, M.V.
Commutativity Race

two high-level operations do not commute

and are not ordered by the program

capture

interference between high-level operations
ConcurrentHashMap data = new ConcurrentHashMap();

for (String key : keys)
    fork {
        data.put(key, Value.compute(key));
    }

new Array[data.size()];
ConcurrentHashMap data = new ConcurrentHashMap();
for (String key : keys)
    fork {
        data.put(key, Value.compute(key));
    }
joinAll
new Array [data.size()];

1. put and size do not commute
2. are ordered by the program
Commutativity Race Detection

- commutativity specification
- structural representation
- happens-before
- commutativity race detector
Commutativity Race Detection

- commutativity specification
- structural representation
- happens-before
- commutativity race detector
**Hashmap commutativity spec**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>put(k_1,v_1)/r_1 \bowtie size()/n_2</code></td>
<td><code>(v_1 = nil \land r_1 = nil) \lor (v_1 \neq nil \land r_1 \neq nil)</code></td>
</tr>
<tr>
<td><code>put(k_1,v_1)/r_1 \bowtie put(k_2,v_2)/r_2</code></td>
<td><code>k_1 \neq k_2 \lor (v_1 = r_1 \land v_2 = r_2)</code></td>
</tr>
<tr>
<td><code>put(k_1,v_1)/r_1 \bowtie get(k_2)/v_2</code></td>
<td><code>k_1 \neq k_2 \lor v_1 = r_1</code></td>
</tr>
<tr>
<td><code>size()/n_1 \bowtie size()/n_2</code></td>
<td><code>true</code></td>
</tr>
<tr>
<td><code>size()/n_1 \bowtie get(k_2)/v_2</code></td>
<td><code>true</code></td>
</tr>
<tr>
<td><code>get(k_1)/v_1 \bowtie get(k_2)/v_2</code></td>
<td><code>true</code></td>
</tr>
</tbody>
</table>
**Hashmap commutativity spec**

\[
\begin{align*}
\text{put}(k_1,v_1)/r_1 \bowtie \text{size()}/n_2 & \iff (v_1 = \text{nil} \land r_1 = \text{nil}) \lor (v_1 \neq \text{nil} \land r_1 \neq \text{nil}) \\
\text{put}(k_1,v_1)/r_1 \bowtie \text{put}(k_2,v_2)/r_2 & \iff k_1 \neq k_2 \lor (v_1 = r_1 \land v_2 = r_2) \\
\text{put}(k_1,v_1)/r_1 \bowtie \text{get}(k_2)/v_2 & \iff k_1 \neq k_2 \lor v_1 = r_1 \\
\text{size()}/n_1 \bowtie \text{size()}/n_2 & \iff \text{true} \\
\text{size()}/n_1 \bowtie \text{get}(k_2)/v_2 & \iff \text{true} \\
\text{get}(k_1)/v_1 \bowtie \text{get}(k_2)/v_2 & \iff \text{true}
\end{align*}
\]
Hashmap commutativity spec

\[
\begin{align*}
\text{put}(k_1,v_1)/r_1 \bowtie \text{size}()/n_2 \iff (v_1 = \text{nil} \land r_1 = \text{nil}) \lor (v_1 \neq \text{nil} \land r_1 \neq \text{nil}) \\
\text{put}(k_1,v_1)/r_1 \bowtie \text{put}(k_2,v_2)/r_2 \iff k_1 \neq k_2 \lor (v_1 = r_1 \land v_2 = r_2) \\
\text{put}\left(\text{'a', 'tom'}\right)/\text{nil} \not\bowtie \text{size}()/1 \\
\text{size}()/n_1 \bowtie \text{size}()/n_2 \iff \text{true} \\
\text{put}\left(\text{'b', 'lol'}\right)/\text{cat} \not\bowtie \text{size}()/1 \\
\text{get}(k_1)/v_1 \bowtie \text{get}(k_2)/v_2 \iff \text{true}
\end{align*}
\]
Commutativity Specifications can be large and complex

Example: ArrayList (part of the spec)

| $t_1 = s_1.remove_at(i_1)$ | $s_2.add_at(i_2, v_2)$ | $(i_1 < i_2 \land s_1[i_2] = v_2) \lor$
|                           |                        | $(i_1 = i_2 \land s_1[i_1] = v_2) \lor$
|                           |                        | $(i_1 > i_2 \land s_1[i_1 - 1] = s_1[i_1])$
| $t_2 = s_2.get(i_2)$      | $(i_1 < i_2 \land s_1[i_2] = s_1[i_2 + 1]) \lor$
|                           | $(i_1 = i_2 \land s_1[i_1] = s_1[i_2 + 1]) \lor$
| $t_1 > i_2$               | $i_1 > i_2$            |

| $t_2 = s_2.indexOf(v_2)$  | $\neg((\exists i : s_1[i] = v_2) \lor$
|                           | $(\exists i < i_1 : s_1[i] = v_2) \lor$
|                           | $(\neg(\exists i < i_1 : s_1[i] = v_2) \land s_1[i_1] = v_2 \land i_1 < |s_1| - 1 \land s_1[i_1 + 1] = v_2)$
| $t_2 = s_2.lastIndexOf(v_2)$ | $\neg((\exists i : s_1[i] = v_2) \lor$
|                           | $(\exists i < i_1 : s_1[i] = v_2) \land \neg(\exists i > i_1 : s_1[i] = v_2))$
| $t_2 = s_2.remove_at(i_2)$ | $(i_1 < i_2 \land s_1[i_2] = s_1[i_2 + 1]) \lor$
|                           | $(i_1 = i_2 \land s_1[i_1] = s_1[i_2 + 1]) \lor$
|                           | $(|s_1| - 1 > i_1 > i_2 \land s_1[i_1] = s_1[i_1 + 1])$
| $s_2.remove_at(i_2)$      | $(i_1 < i_2 \land s_1[i_2] = s_1[i_2 + 1]) \lor$
|                           | $(i_1 = i_2 \land s_1[i_1] = s_1[i_2 + 1]) \lor$
|                           | $(|s_1| - 1 > i_1 > i_2 \land s_1[i_1] = s_1[i_1 + 1])$
| $t_2 = s_2.set(i_2, v_2)$ | $(i_1 < i_2 \land s_1[i_2] = s_1[i_2 + 1] = v_2) \lor$
|                           | $(i_1 = i_2 \land s_1[i_1] = s_1[i_2 + 1] = v_2) \lor$
| $i_1 > i_2$               | $i_1 > i_2$

$s_2.set(i_2, v_2)$

$(i_1 < i_2 \land s_1[i_2] = s_1[i_2 + 1] = v_2) \lor$

$(i_1 = i_2 \land s_1[i_1] = s_1[i_2 + 1] = v_2) \lor$

$i_1 > i_2$

“Verification of Semantic Commutativity Conditions and Inverse Operations on Linked Data Structures”, Deokhwan Kim & Martin Rinard, ACM PLDI’11
Commutativity Race Detection
Naïve Algorithm

Initially $\text{Seen} = \varepsilon$

Then during execution, for each operation $\text{op}$ do:

```python
foreach b ∈ Seen {
    if not (\text{op} \bowtie b) and (\text{op} \mid\mid b)
        report 'commutativity race'
}
\text{Seen} = \text{Seen} \cdot \text{op}
```
Commutativity Race Detection
Naïve Algorithm

Initially $\text{Seen} = \varepsilon$

Then during execution, for each operation $\text{op}$ do:

```c
foreach \texttt{b} \in \texttt{Seen} \{ \\
    \text{if not (} \texttt{op} \bowtie \texttt{b} \text{) and } (\texttt{op} \mid\mid \texttt{b}) \\
    \text{report 'commutativity race'} \\
\}
\text{Seen} = \text{Seen} \cdot \text{op}
```

Commut? \quad Ordered?
Commutativity Race Detection
Naïve Algorithm

Initially $\text{Seen} = \varepsilon$

Then during execution, for each operation $\text{op}$ do:

```plaintext
foreach $b \in \text{Seen}$ {
    if not($\text{op} \bowtie b$) and ($\text{op} || b$)
        report 'commutativity race'
}
\text{Seen} = \text{Seen} \cdot \text{op}
```

Two problems:

1. Extremely inefficient: worst-case $O(\text{Seen})$ time per-operation
2. Does not explore commutativity sharing between operations
Commutativity Race Detection
Naïve Algorithm

For trace: \( \text{put}(k_1, v_1)/\text{nil} \cdot \text{put}(k_2, v_2)/\text{nil} \cdot \text{put}(k_3, v_3)/\text{nil} \cdot \text{size}() / 3 \)

Upon encountering \( \text{size}() / 3 \), we will perform 3 checks:

- \( \text{put}(k_1, v_1)/\text{nil} \)
- \( \text{put}(k_2, v_2)/\text{nil} \)
- \( \text{put}(k_3, v_3)/\text{nil} \)

Yet, what matters is whether a concurrent \text{resize} has occurred

Wanted: leverage sharing between observed operations
Commutativity Race Detection

\[
\begin{align*}
    \text{put}(k_1,v_1)/r_1 \bowtie \text{size}()/n \Leftrightarrow & \quad (v_1 = \text{nil} \land r_1 = \text{nil}) \lor (v_1 \neq \text{nil} \land r_1 \neq \text{nil}) \\
    \text{put}(k_1,v_1) \bowtie \text{get}(k_2)/v_2 & \Rightarrow (v_1 \neq \text{nil} \land v_2 = r_2) \\
    k_1 \neq k_2 & \Leftrightarrow (v_1 = r_1 \land v_2 = r_2) \\
    \end{align*}
\]
Micro-ops representation

operations

\( \text{put('a','tom')/nil} \quad \not\quad \text{size()}/1 \)
Micro-ops representation

operations

put(‘a’,‘tom’)/nil ∩ size()/1

μ

w:‘a’ -> μ

resize

μ

size

μ-operations

conflict
From logic to $\mu$-operations

\[
\begin{align*}
\text{put}(k_1,v_1)/r_1 \bowtie&\text{size()}/n_2 \iff (v_1 = \text{nil} \land r_1 = \text{nil}) \lor (v_1 \neq \text{nil} \land r_1 \neq \text{nil}) \\
\text{put}(k_1,v_1)/r_1 \bowtie&\text{put}(k_2,v_2)/r_2 \iff k_1 \neq k_2 \lor (v_1 = r_1 \land v_2 = r_2) \\
\text{put}(k_1,v_1)/r_1 \bowtie&\text{get}(k_2)/v_2 \iff k_1 \neq k_2 \lor v_1 = r_1 \\
\text{size()}/n_1 \bowtie&\text{size()}/n_2 \iff \text{true} \\
\text{size()}/n_1 \bowtie&\text{get}(k_2)/v_2 \iff \text{true} \\
\text{get}(k_1)/v_1 \bowtie&\text{get}(k_2)/v_2 \iff \text{true} \\
\end{align*}
\]

VS.

\[
\begin{align*}
\text{put}(\text{‘a’},\text{‘tiger’})/\text{nil} \quad &\text{size()}/1 \\
\mu \downarrow &\text{resize} \\
\text{w:‘a’} &\text{conflict} \\
\mu \downarrow &\text{size}
\end{align*}
\]
From logic to μ-operations

\[
\begin{align*}
\text{put}(k_1,v_1)/r_1 \Join \text{size}()/n_2 & \iff (v_1 = \text{nil} \land r_1 = \text{nil}) \lor (v_1 \neq \text{nil} \land r_1 \neq \text{nil}) \\
\text{put}(k_1,v_1)/r_1 \Join \text{put}(k_2,v_2)/r_2 & \iff k_1 \neq k_2 \lor (v_1 = r_1 \land v_2 = r_2) \\
\text{put}(k_1,v_1)/r_1 \Join \text{get}(k_2)/v_2 & \iff k_1 \neq k_2 \lor v_1 = r_1 \\
\text{size}()/n_1 \Join \text{size}()/n_2 & \iff \text{true} \\
\text{size}()/n_1 \Join \text{get}(k_2)/v_2 & \iff \text{true} \\
\text{get}(k_1)/v_1 \Join \text{get}(k_2)/v_2 & \iff \text{true}
\end{align*}
\]
Commutativity Compiler

Benefits

size() / 3

put(k1, v1)/nil

put(k2, v2)/nil

put(k3, v3)/nil

conflict on operations
conflict on μ-operations

size

resize

w:k1

w:k2

w:k3
Commutativity Race Detection

put(k1,v1)/r1 ⋈ size()/n2 ⇔ (v1 = nil ∧ r1 = nil) ∨ (v1 ≠ nil ∧ r1 ≠ nil)

put(k1,v1)/r1 ⋈ put(k2,v2)/r2 ⋈ get(k2)/v2

size()/n1 ⋈ size()/n2

⟺ (v1 = nil ∧ r1 = nil) ∨ (v1 ≠ nil ∧ r1 ≠ nil)

v1 = r1

k1 ≠ k2 ∨ (v1 = r1 ∧ v2 = r2)
Commutativity Race Detection

Key idea: associate vector clocks with $\mu$-operations
Asymptotic complexity

$O(n)$ conflict checks per encountered operation

Can we do better?
Extended Logical Fragment (ECL)

A logical fragment which ensures $O(1)$ checks per operation
Extended Logical Fragment (ECL)

\[ S ::= V_1 \neq V_2 \mid S \land S \mid \text{true} \mid \text{false} \]

Example:

\[ \text{put}(k_1,v_1)/r_1 \bowtie \text{get}(k_2)/v_2 \iff k_1 \neq k_2 \lor v_1 = r_1 \]

Limited relationship between variables of different methods.
For instance, \( V_1 = V_2 \) not allowed
Extended Logical Fragment (ECL)

\[ S ::= V_1 \neq V_2 \mid S \land S \mid \text{true} \mid \text{false} \]

\[ B ::= P_{V_1} \mid P_{V_2} \mid \neg B \mid B \land B \mid B \lor B \mid \text{true} \mid \text{false} \]

Any predicate over \( V_1 \)

Any predicate over \( V_2 \)
Extended Logical Fragment (ECL)

\[ S ::= V_1 \neq V_2 \mid S \land S \mid \text{true} \mid \text{false} \]

\[ B ::= P_{v_1} \mid P_{v_2} \mid \neg B \mid B \land B \mid B \lor B \mid \text{true} \mid \text{false} \]

Example:

\[ \text{put}(k_1, v_1)/r_1 \bowtie \text{get}(k_2)/v_2 \iff k_1 \neq k_2 \lor v_1 = r_1 \]
Extended Logical Fragment (ECL)

\[ S ::= V_1 \neq V_2 \mid S \land S \mid \text{true} \mid \text{false} \]

\[ B ::= P_{V_1} \mid P_{V_2} \mid \neg B \mid B \land B \mid B \lor B \mid \text{true} \mid \text{false} \]

\[ X ::= S \mid B \mid X \land X \mid X \lor B \]

Not \( X \lor X \)
Extended Logical Fragment (ECL)

\[ S ::= V_1 \neq V_2 \mid S \land S \mid \text{true} \mid \text{false} \]

\[ B ::= P_{V_1} \mid P_{V_2} \mid \neg B \mid B \land B \mid B \lor B \mid \text{true} \mid \text{false} \]

\[ X ::= S \mid B \mid X \land X \mid X \lor B \]

Example:

\[ \text{put}(k_1, v_1)/r_1 \bowtie \text{get}(k_2)/v_2 \iff k_1 \neq k_2 \lor v_1 = r_1 \]

- Comes from \( S \)
- Comes from \( B \)

Not \( X \lor X \)
Extended Logical Fragment (ECL)

\[
S ::= V_1 \neq V_2 \mid S \land S \mid \text{true} \mid \text{false}
\]

\[
B ::= P_{V_1} \mid P_{V_2} \mid \neg B \mid B \land B \mid B \lor B \mid \text{true} \mid \text{false}
\]

\[
X ::= S \mid B \mid X \land X \mid X \lor B
\]

ECL ensures $O(1)$ time per operation

Extends SIMPLE from:

Exploiting the commutativity lattice, Milind Kulkarni, Donald Nguyen, Dimitrios Prountzos, Xin Sui, Keshav Pingali, ACM PLDI 2011
Classic Read-Write Race Detection...

...is now a special case:

\[
\begin{align*}
\text{read}() / r_1 & \bowtie \text{read}() / r_2 & \iff & \text{true} \\
\text{read}() / r_1 & \bowtie \text{write}(v_2) & \iff & \text{false} \\
\text{write}(v_1) & \bowtie \text{write}(v_2) & \iff & \text{false}
\end{align*}
\]

Fits inside ECL
What about this?

...a more precise spec:

\[
\begin{align*}
\text{read}() / r_1 & \bowtie \text{read}() / r_2 & \iff & \text{true} \\
\text{read}() / r_1 & \bowtie \text{write}(v_2) & \iff & r_1 = v_2 \\
\text{write}(v_1) & \bowtie \text{write}(v_2) & \iff & v_1 = v_2
\end{align*}
\]

\textbf{NO!}

ECL does not allow = between variables of different operations
Commutativity race detection

Commutativity specification

structural representation

happens-before

commutativity race detector
Experimental Results

Implemented a commutativity detector for Java

Found both correctness and performance bugs in large-scale apps (e.g., H2 Database)
Commutativity race detection

Parametric Concurrency Analysis Framework
Generalizes classic read-write race detection
Enables concurrency analysis for clients of high-level abstractions
Logical fragment ensures $O(1)$ complexity
More Applications

Software Defined Networks

Partial Order Reduction

Optimistic Concurrency Control

Eventual Consistency
Open Problems

commutativity specification

structural representation

happens-before

commutativity race detector

\[ \text{put}(k_1,v_1)/r_1 \bowtie \text{size}() / n \iff ( v_1 = \text{nil} \land r_1 = \text{nil} ) \lor ( v_1 \neq \text{nil} \land r_1 \neq \text{nil} ) \]

\[ \text{put}(k_1,v_1)/r_1 \bowtie \text{size}() / n \iff ( v_1 = \text{nil} \land r_1 = \text{nil} ) \lor ( v_1 \neq \text{nil} \land r_1 \neq \text{nil} ) \]

\[ k_1 \neq k_2 \iff ( v_1 = r_1 \land v_2 = r_2 ) \]

\[ k_1 \neq k_2 \iff ( v_1 = r_1 ) \]

[0,0,0]
[1,2,0]
[0,1,0,3]
[4,2,1,0]

new {}
fork
put
join
size
Question 1: Can we learn the commutativity spec?

Specification of a Map ADT → Automatic Inference

\[ \begin{align*}
\text{put}(k_1, v_1)/r_1 & \bowtie \text{size}()/n_2 \quad \Leftrightarrow \quad (v_1 = \text{nil} \land r_1 = \text{nil}) \lor (v_1 \neq \text{nil} \land r_1 \neq \text{nil}) \\
\text{put}(k_1, v_1)/r_1 & \bowtie \text{put}(k_2, v_2)/r_2 \quad \Leftrightarrow \quad k_1 \neq k_2 \lor (v_1 = r_1 \land v_2 = r_2) \\
\text{put}(k_1, v_1)/r_1 & \bowtie \text{get}(k_2)/v_2 \quad \Leftrightarrow \quad k_1 \neq k_2 \lor v_1 = r_1 \\
\text{size}()/n_1 & \bowtie \text{size}()/n_2 \quad \Leftrightarrow \quad \text{true} \\
\text{size}()/n_1 & \bowtie \text{get}(k_2)/v_2 \quad \Leftrightarrow \quad \text{true} \\
\text{get}(k_1)/v_2 & \bowtie \text{get}(k_2)/v_2 \quad \Leftrightarrow \quad \text{true}
\end{align*} \]
Question 1:
Can we learn the commutativity spec?

Learning Commutativity Specifications, CAV’15
Black Box Learning of specifications from examples

Many open problems: state, quantifiers, minimization
Open Problems

can we learn it?

commutativity specification

structural representation

happens-before

commutativity race detector

put(k₁,v₁)/r₁ ⋈ size() \ni (v₁ = nil ∧ r₁ = nil) ∨ (v₁ ≠ nil ∧ r₁ ≠ nil)

put(k₁,v₁)/r₁

put(k₁,v₁)/r₁

size()

[0,0,0]

[1,2,0]

[0,1,0,3]

[4,2,1,0]
Question 2:
Logic Expressivity vs. Asymptotic Complexity

Recall that ECL allows for $O(1)$ checks per operation:

\[
S ::= V_1 \neq V_2 \mid S \land S \mid \text{true} \mid \text{false}
\]
\[
B ::= P_{V_1} \mid P_{V_2} \mid \neg B \mid B \land B \mid B \lor B \mid \text{true} \mid \text{false}
\]
\[
X ::= S \mid B \mid X \land X \mid X \lor B
\]

What is the richest logical fragment which guarantees $O(1)$?

What about $O(\log N)$?
Open Problems

can we learn it?

richest fragment to obtain $O(1)$ time?

put($k_1,v_1$)/$r_1 \bowtie$ size()/$n$ $\iff$ ($v_1 = \text{nil} \land r_1 = \text{nil}$) $\lor$ ($v_1 \neq \text{nil} \land r_1 \neq \text{nil}$)

put($k_1,v_1$)/$r_1 \bowtie$ put($k_2,v_2$)/$r_2$ $\iff$ $k_1 \neq k_2$

$\iff$ ($v_1 = r_1 \land v_2 = r_2$)

commutativity specification

structural representation

happens-before

commutativity race detector

new {}
fork
put
join
size
Question 3: Commutativity Compiler Optimizations

Obtaining a succinct micro-operation representation can be tricky.

New compiler optimizations which depend on the logical fragment and on the property we are checking.
Open Problems

- can we learn it?
- richest fragment to obtain $O(1)$ time?
- compiler optimizations exploiting fragment

commutativity specification

structural representation

happens-before

commutativity race detector
Question 4: Impossibility questions

Can a read-write race detector precisely detect commutativity races?

At what cost?

What is the “consensus-like” hierarchy of concurrency analyzers?
Open Problems

can we learn it?

richest fragment to obtain $O(1)$ time?

impossibility questions

commutativity specification

structural representation

can we learn it?

richest fragment to obtain $O(1)$ time?

impossibility questions

commutativity specification

structural representation

happens-before

commutativity race detector

[0,0,0]
[1,2,0]
[0,1,0,3]
[4,2,1,0]

new {}
fork
put
join
size
Open Problems

- Can we learn it?
- Richest fragment to obtain $O(1)$ time?
- Compiler optimizations exploiting fragment
- Impossibility questions

**Commutativity Specification**
- \( \text{put}(k_1,v_1)/r_1 \bowtie \text{size()}/n \Leftrightarrow (v_1 = \text{nil} \land r_1 = \text{nil}) \lor (v_1 \neq \text{nil} \land r_1 \neq \text{nil}) \)

**Happens-Before**
- \([0,0,0,0], [1,2,0,0], [0,1,0,3], [4,2,1,0]\)

**Structural Representation**

**Commutativity Race Detector**
- Other analyzers (e.g., atomicity)
- Different instantiations (e.g., VCs, SP-graphs)
Commutativity race detection

Parametric Concurrency Analysis Framework
Generalizes classic read-write race detection
Enables concurrency analysis for clients of high-level abstractions
Logical fragment ensures O(1) complexity
Many open problems