Model Checking a Lazy Concurrent List-Based Set Algorithm

ZHANG Shaojie, LIU Yang
National University of Singapore

The Fourth IEEE International Conference on Secure Software Integration and Reliability Improvement

Agenda

- Introduction
- Background
- Our approach
  - Overview
  - Linearizability definition
  - Modeling language
  - Linearizability as refinement relation
- Experiment
- Conclusion & Future Work

Introduction

- Concurrent objects are notoriously hard to design correctly.
  - Esp. Lock-free & wait-free ones.
- Linearizability is an accepted correctness criterion for shared objects.
  - A shared object is *linearizable* if each operation on the object can be understood as occurring instantaneously at some point, (a.k.a. *linearization point)*
- Formal verification or proof of linearizability rely on the knowledge of linearization points
  - Expert knowledge
  - Linearization points are hard to be statically determined

Introduction

- Verify linearizaibility against lazy concurrent list-based set algorithm
  - Proposed by Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N, Scherer III, and Nir Shavit in 2005.
  - Martin Vechev, Eran Yahav, and Greta Yorsh described a variation with weaker validation condition in 2009.
- Why choose this algorithm?
  - Highly concurrent, non-fixed linearization points.
  - Complexity: non-deterministic target location
  - Manipulates dynamic allocated memory heavily & Need a garbage collector

Agenda

- Introduction
- Background
- Our Approach
  - Overview
  - Linearizability definition
  - Modeling language
  - Linearizability as refinement relation
- Experiment
- Conclusion & Future Work

Concurrent List-based Set

- Set interface
  - Unordered collection of items
  - No duplicates
  - Methods
    - bool add(int x): put x in set; if succeeds, return true
    - bool remove(int x) take x out of set
    - bool contains(int x) tests if x in set

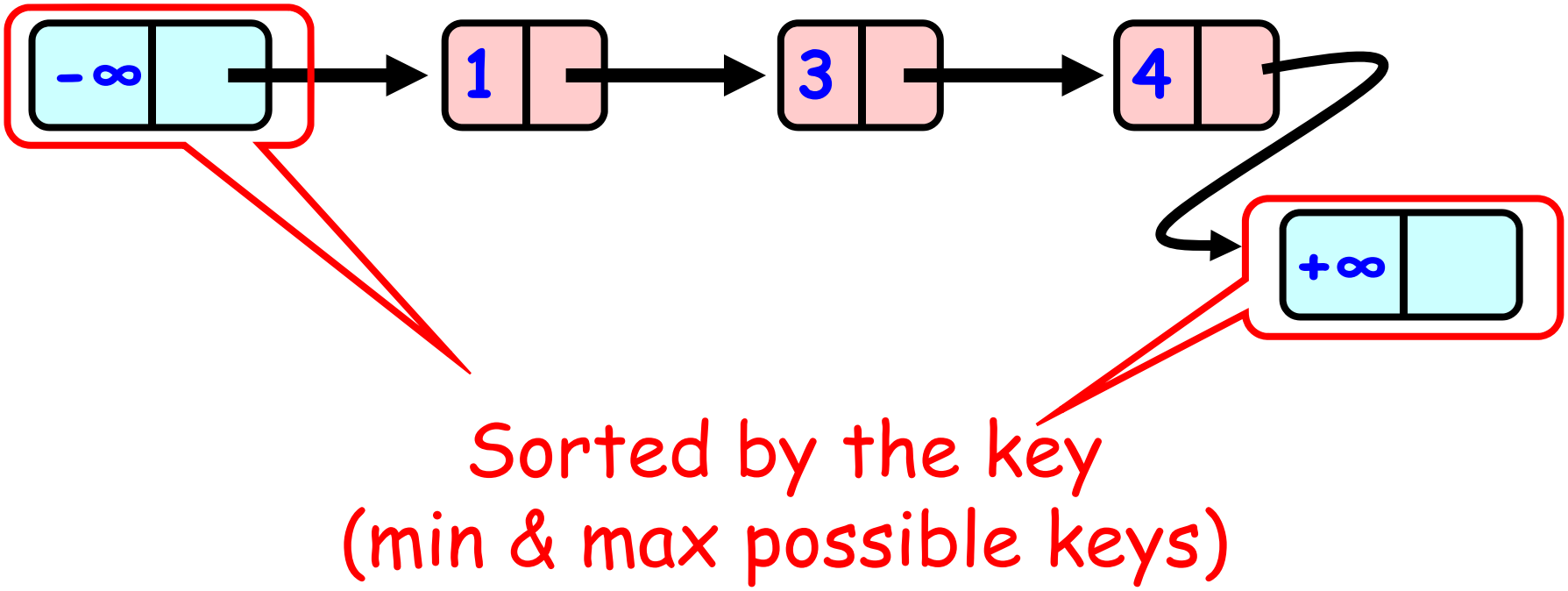
Concurrent List-based Set

- Set as a single-linked sorted list
- List node

```
public class Node {
 public int key;     // item of interest
 public Node next;   // Reference to next node
 public bool marked; //Indicate this node is about to be
removed
}
```

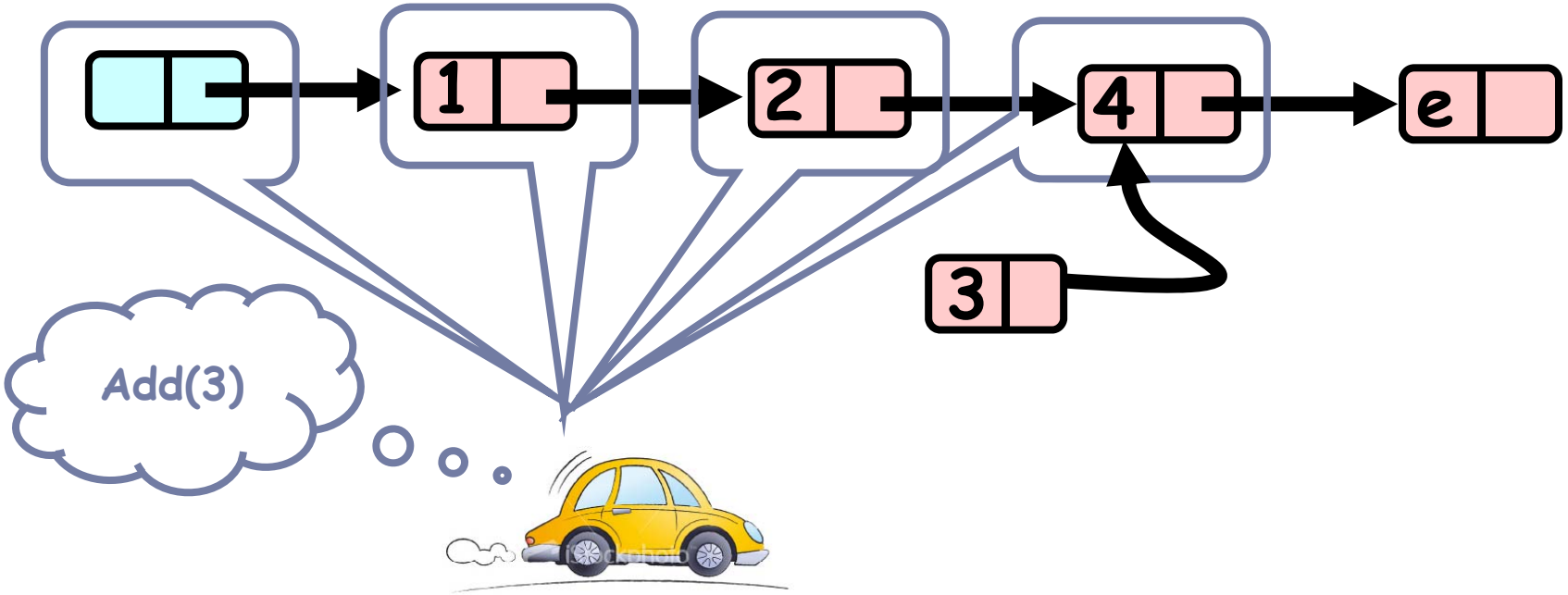
Concurrent List-based Set

- The sentinel nodes can only be compared, not modified.



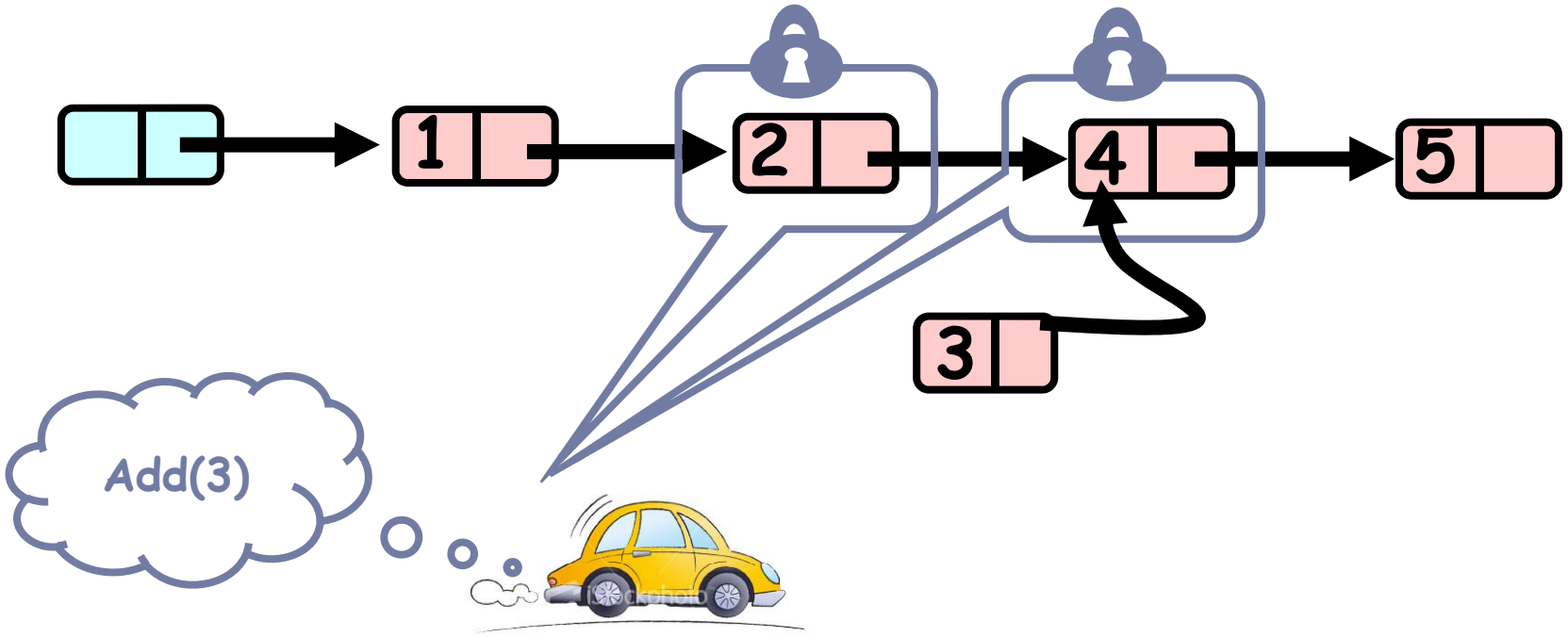
Concurrent List-based Set

- Optimistic locking scheme
  - Traverse without Locking



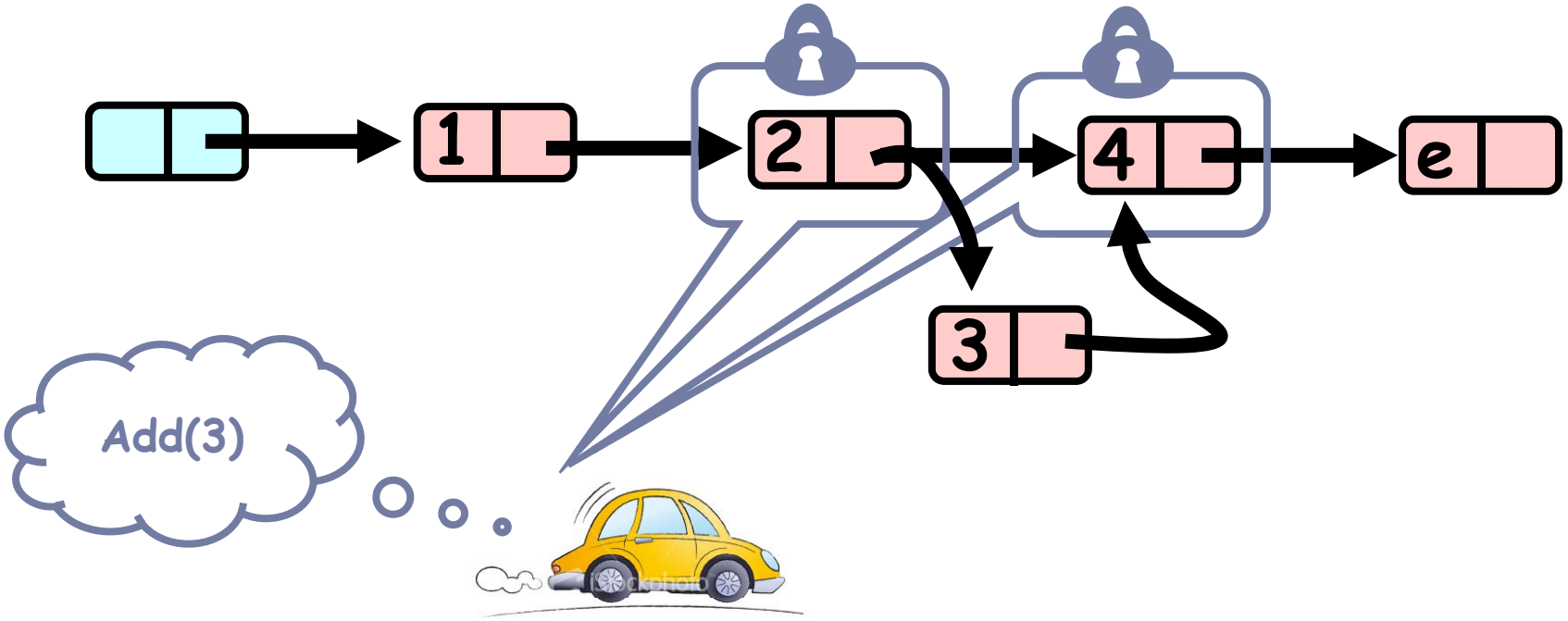
Concurrent List-based Set

- Optimistic locking scheme
  - Lock the target node and its predecessor



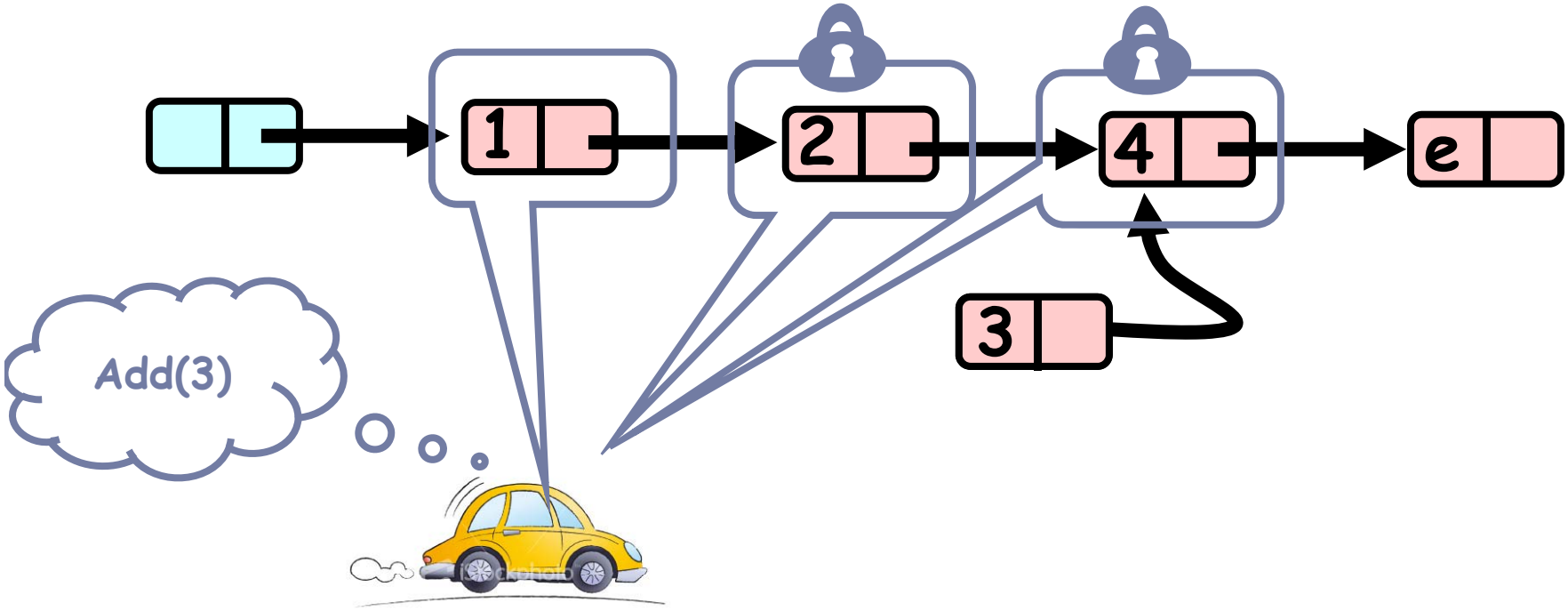
Concurrent List-based Set

- Optimistic locking scheme
  - Validation
    - Node 2 is not marked true
    - Node 4 still successor to Node 2



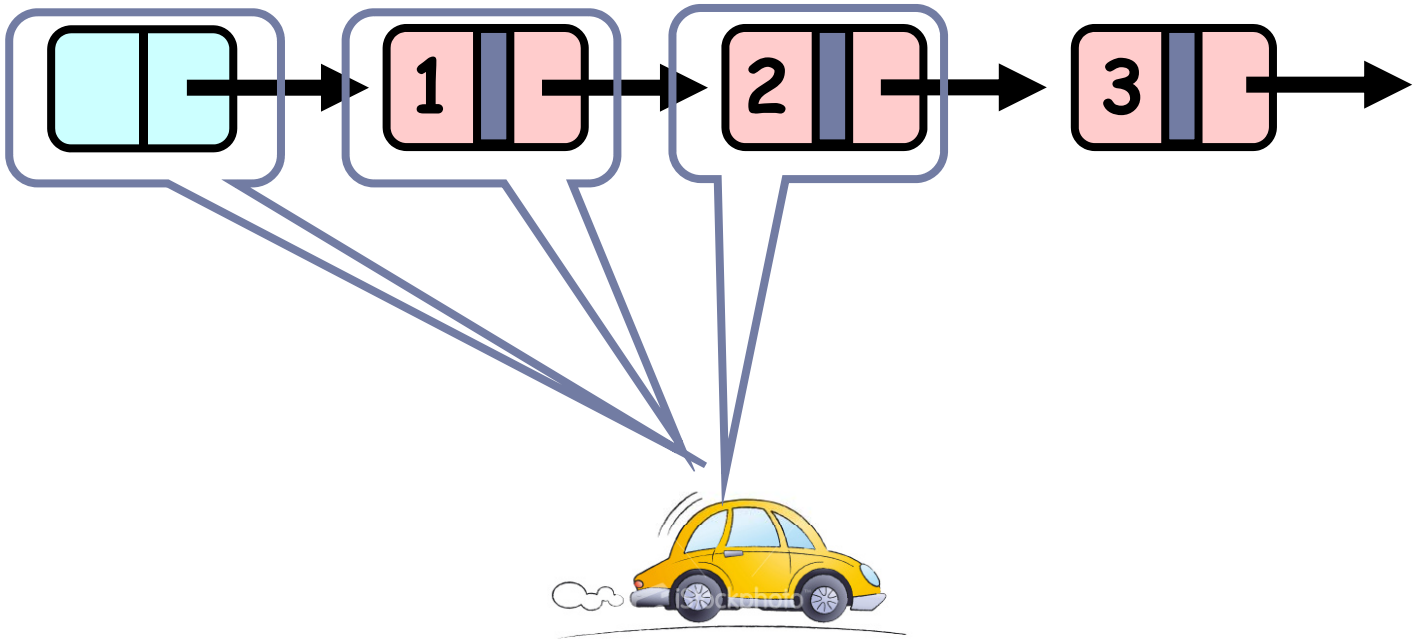
Concurrent List-based Set

- Optimistic locking scheme
  - Validation



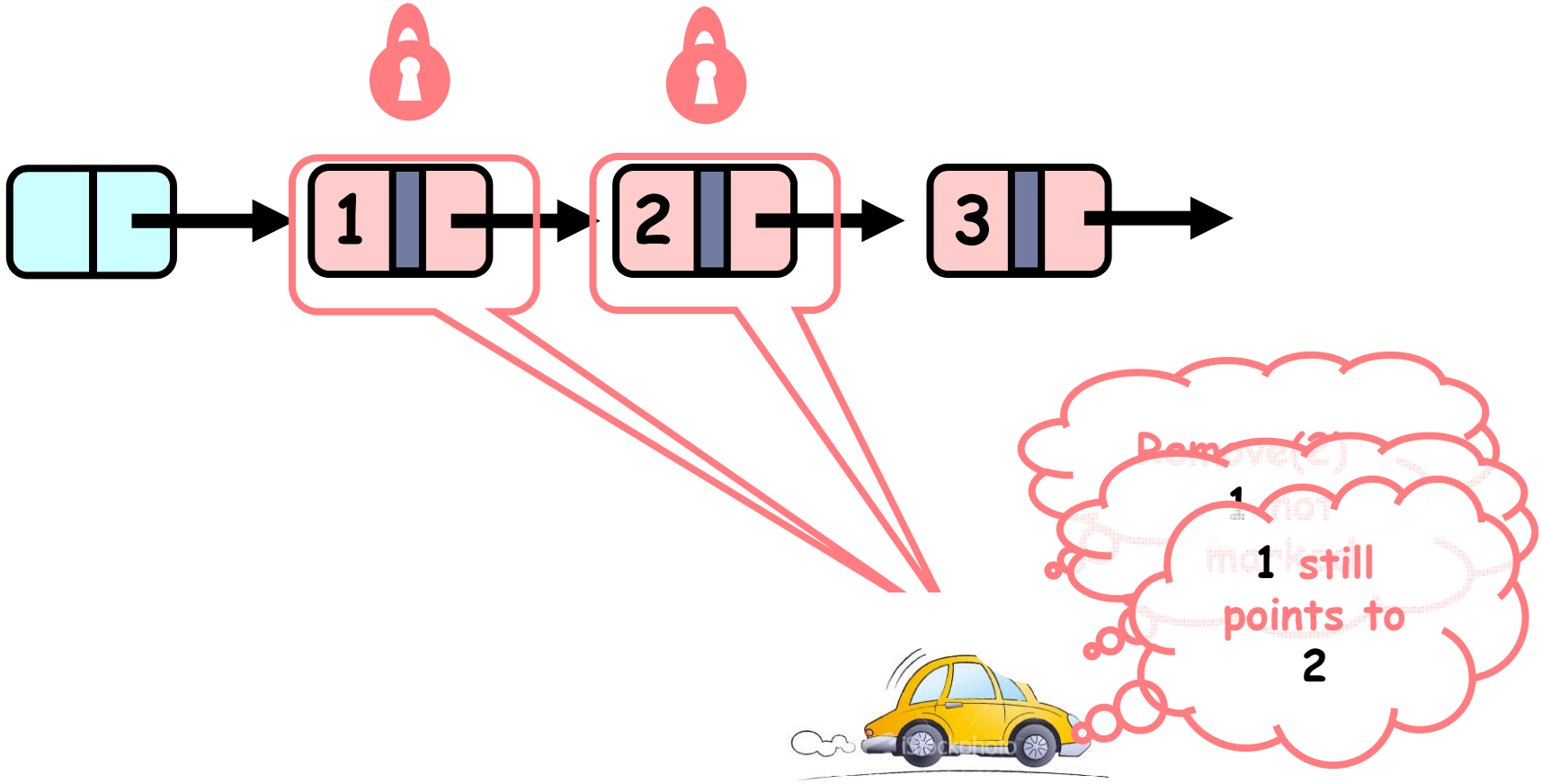
Concurrent List-based Set

- Remove



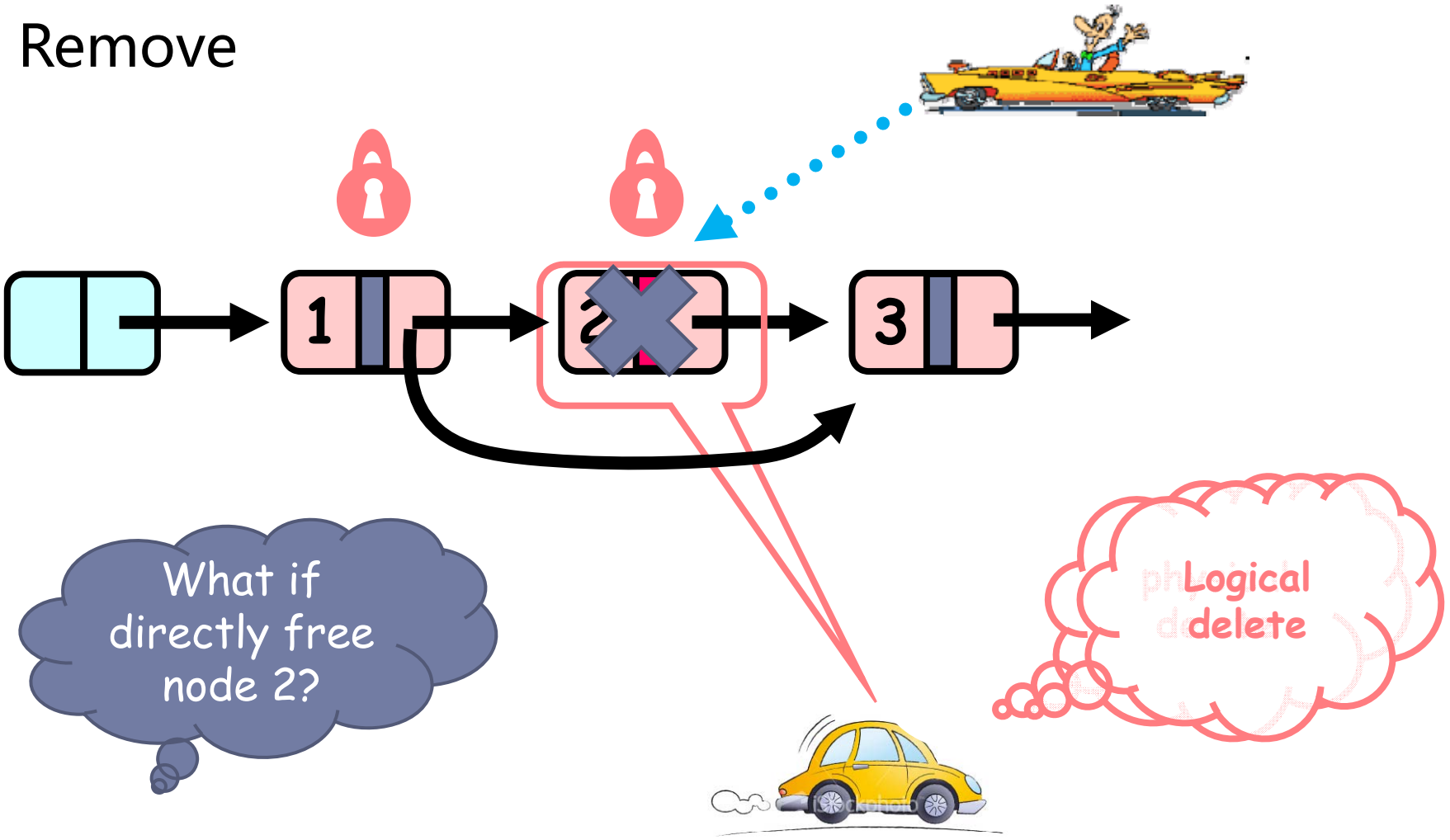
Concurrent List-based Set

- Remove



Concurrent List-based Set

- Remove



Agenda

- Introduction
- Background
- Our Approach
  - Overview
  - Linearizability definition
  - Modeling language
  - Linearizability as refinement relation
- Experimental Result
- Conclusion & Future Work

Overview of Our Approach

- The definition of linearizability is cast to trace refinement relation.
  - Fully automatically
  - Without the knowledge of linearization points
- Modeling language: CSP#(Communicating sequential programs)
  - Event-based; LTS-based semantics
- Tool: PAT(Process Analysis Toolkit)
  - A toolkit for automatically analyzing event-based concurrent systems including refinement checking

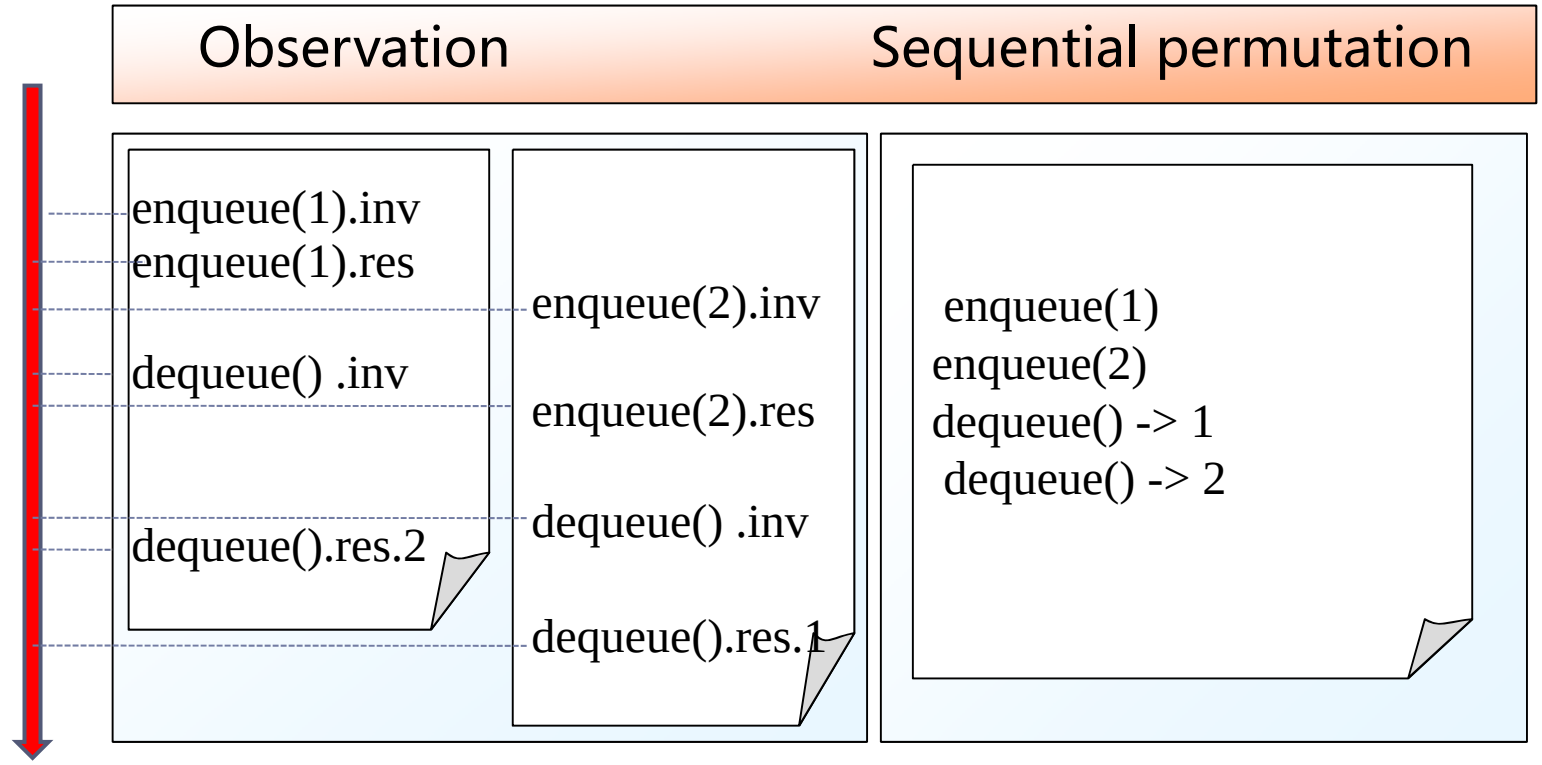
Overview of Our Approach

- Dynamic memory allocation
  - Pre-allocate a bounded array as a private memory space
- Garbage collection
  - Reference counting algorithm

Linearizability Manifesto

- Each operation could "take effect" instantaneously between invocation and response
- Correlate every concurrent execution with a consistent sequential atomic execution of the operations.
  - Preserve real-time order
  - Respect the sequential specification of the object

Linearizability Example



Modeling language

- CSP#

  Communicating Sequential Processes with shared variables, low-level programming constructs and user defined data structures.

- Grammar

$$
\begin{aligned}
P ::= & \ Stop \mid Skip \\
\mid & \ e\{program\} \rightarrow P \\
\mid & \ P \setminus X \\
\mid & \ P_1;\ P_2 \\
\mid & \ P_1 \ \square \ P_2 \\
\mid & \ if(b)\ \{P_1\}\ else\ \{P_2\} \\
\mid & \ P_1 \ ||| \ P_2 \\
\mid & \ case\{b1 : P_1\ b2 : P_2\ \cdots;\ default : P\} \\
\mid & \ atomic\{P\} \\
e ::= & \ name(.expression)*
\end{aligned}
$$

Linearizability as Refinement Relations

- Theorem

  Suppose *Lsp* is a linearizable specification LTS model for a shared object *o*, consider *Lim* that implements object *o*, then Traces of *Lim* are linearizable iff *Lim* refines *Lsp*.

- 1st-Step:

  Define the linearizable specification model

- Specify each operation *op* of a shared object *o* on a process p_i using three atomic steps:
  - the invocation action $inv(op)_i$
  - the linearization action $lin(op)_i$ (Invisible event)
  - the response action $res(op, resp)_i$.

Linearizability as refinement relations

```
//Specification
var<Set> s;

Sys = |||i:{0..N-1}@(P(i, 0));

P(i, j) = ifa(j < Q){( (Add(i, j)[] Remove(i, j)[]Contains(i, j)))};

Add(i, j) = []x:{MIN..MAX}@
            (
                add_inv.i.x -> tau{s.Add(i,x)}-> add_res.i.x.(s.GetAddData())-> P(i, j+1)
            );

Remove(i, j) = []x:{MIN..MAX}@
            (
                rm_inv.i.x -> tau{s.Remove(i,x)}-> rm_res.i.x.(s.GetRemoveData())-> P(i, j+1)
            );
Contains(i, j) = []x:{MIN..MAX}@
            (
                ct_inv.i.x -> tau{s.Contains(i,x)} -> ct_res.i.x.(s.GetContainData())->P(i, j+1)
            );
```

Linearizability as Refinement Relations

- 2nd -Step:
  Consider the implementation of object o.
- The visible events of *impl* are also those $inv(op)_i$'s and $res(op, resp)_i$'s.
- Memory management operations are encapsulated as methods in the inner library of PAT.

Linearizability as Refinement Relations

- Memory allocation

  var<EntryList> l = new EntryList(M, MIN, MAX);

  - Reference Counting Garbage Collector
  - Always keep the number of references to each list node
  - Collector runs when the reference of some list node becomes zero

```
public class Node {
 public int key;
 public Node next;
 public bool marked;
 public int reference;
//the number of variables pointing to this node
}
```

Linearizability as Refinement Relations

- Reference Counting Garbage Collector
  - Whenever a pointer variable to a list node is modified, update the *reference*

```
Predecessor = Current
```

```
Assign(Predecessor, Current)
{
    ...
    IncreaseReference(Current)
    DecreaseReference(Predecessor)
}
```

  - Don’ t consider the nodes of which reference is zero during the checking

Agenda

- Introduction
- Background
- Our Approach
  - Overview
  - Linearizability definition
  - Modeling language
  - Linearizability as refinement relation
- Experiment
- Conclusion & Future Work

Experimental result

- Testbed is a server with 2.813 GHz Intel Xeon 64-bit CPU and 32 GB memory.

The maximum number of inserted keys

The number of processes

The number of operations each process performs

Set			Result	
#Proc	#Key	#Operation	#States	Time(sec)
2	1	∞	265904	37.06
2		1		
2		1		
2		2		
3	1	1		
3	2	1	-	-

- '—' means infeasible.
- '∞' means unbounded number.

This model is built inside PAT, http://pat.comp.nus.edu.sg

Optimization

- Function details about dynamic memory allocation and reference-counting garbage collection are hided in the embedded library of PAT.
  - No intermediate states during the function execution are generated.
- Manually combine sequences of local actions into one atomic block

Agenda

- Introduction
- Background
- Our Approach
  - Overview
  - Linearizability definition
  - Modeling language
  - Linearizability as refinement relation
- Experiment
- Conclusion & Future Work

Conclusion

- Verify linearizability using trace refinement relation
- Show that refinement checking algorithm behind PAT allows verifying linearizability against concurrent objects
  - Without the knowledge of linearization points
  - Fully automatically
- Show that PAT provides a fairly convenient and efficient way to define new data types and complex functions in a programming language
  - Leaves the model clean
  - Avoid augmenting because of the runtime environment

On-going and future work

- Deal with infamous state explosion problem
  - Symmetry reduction (in progress)
  - Partial order reduction
  - Combine various state space reduction techniques and parameterized refinement checking for infinite number of processes

The End

Thank you!
Q&A