Controlled Re-execution for Disjunctive Predicates

EE 382N Distributed Systems

Term Project

Fall 2015

Elie Antoun

Karim Serhan

Motivation

Mo•ti•va•tion
/ˌmōdəˈvāSH(ə)n/

The driving force by which humans achieve their goals

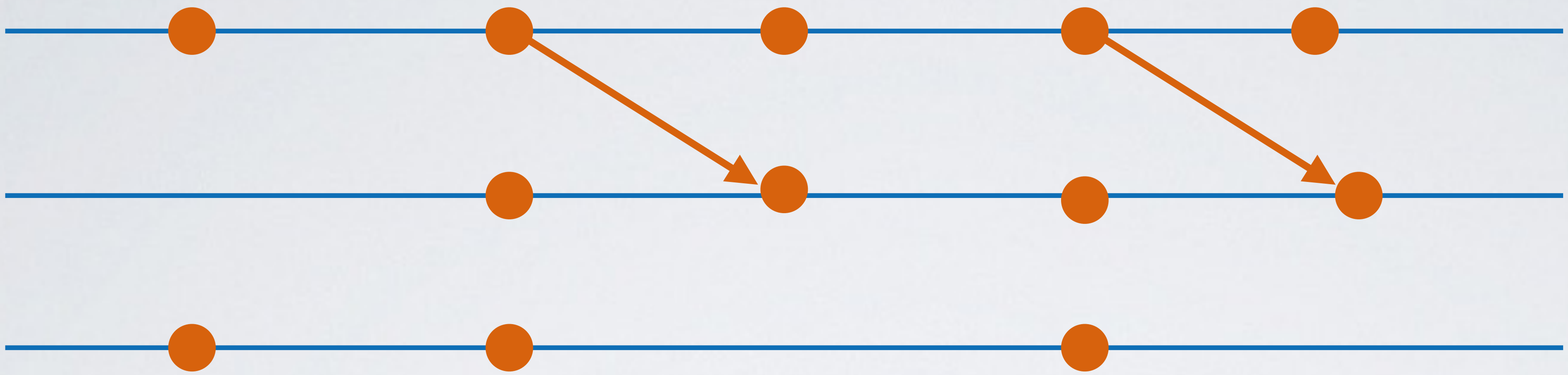
Non-deterministic nature of distributed computations

Present bugs are difficult to reproduce

Proliferation of distributed computations

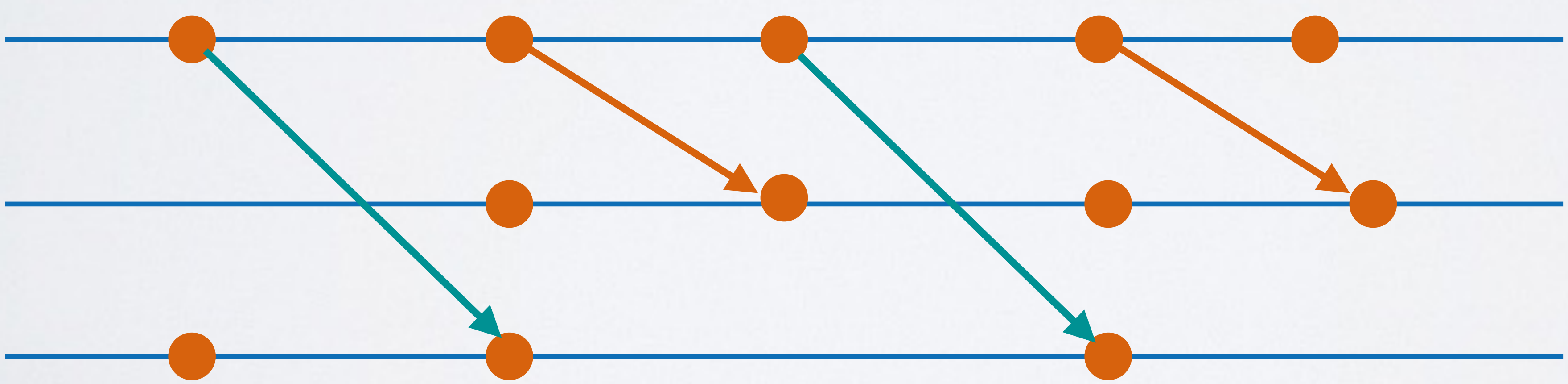
Need to control a distributed computation

Control Re-execution



Uncontrolled Execution

Messages between events

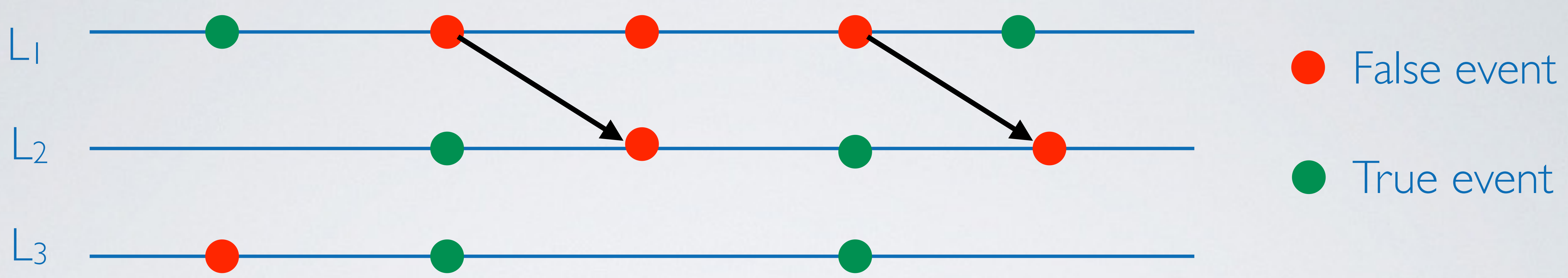


Controlled Execution

Messages between events

Synchronization messages

Disjunctive Predicates



Global predicate B is a disjunction of local predicates on processes

$$B = L_1 \vee L_2 \vee L_3$$

Problem Statement

Given a distributed computations and a predicate B, we aim to control the execution by adding synchronization messages to the computation so that B is always satisfied.

B = at least one process is alive at any time $\Rightarrow$ B = alive(P_1) $\vee$ alive(P_2) … $\vee$ alive(P_n)

B = mutual exclusion for two processes $\Rightarrow$ B = $\neg$CS(P_1) $\vee$ $\neg$CS(P_2)

Admissible Sequence

For a given computation and a predicate , an admissible sequence of events α captures some safe execution of events S , where $S \subseteq E$

Properties of admissible sequence

1. Agreement: an admissible sequence α should be consistent with the happened before relation $\rightarrow$
   $\forall\, i, j : i < j \Rightarrow \alpha_j \nrightarrow \alpha_i$

2. Boundary Condition: an admissible sequence must start with an initial event and end in a final event
   $(\alpha_0 \in E.\bot) \wedge (\alpha_{n-1} \in E.\top)$

4. Continuity: an admissible sequence must be continuous
   $\forall\, (\alpha_j, \alpha_i \in \alpha) \wedge (\alpha_j, \alpha_i \notin E.\top) \wedge (\alpha_i.succ \rightarrow \alpha_j) \Rightarrow \alpha_{i+1} = \alpha_i.succ$

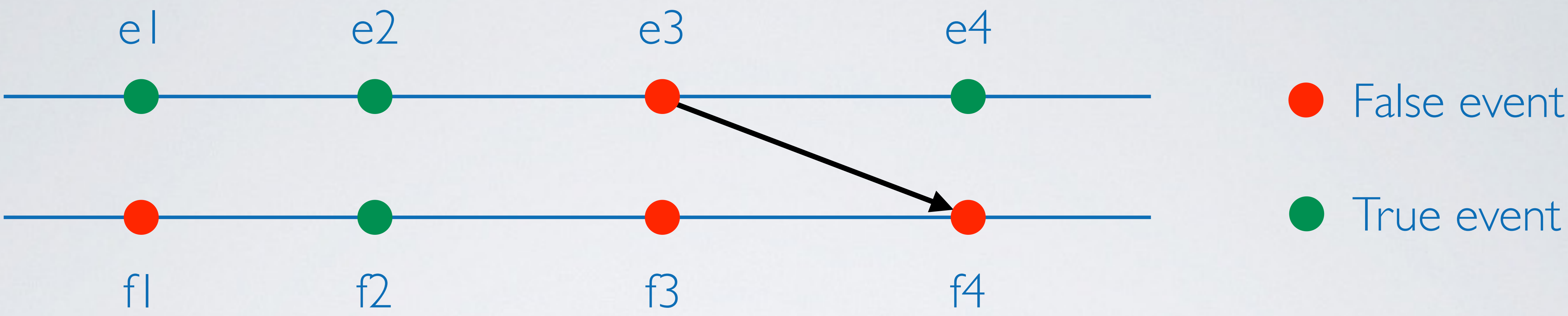
5. Safety: α must be a safe execution.
   $legal(C, E, \alpha) \wedge (\exists\, \alpha_i \in \alpha \wedge \alpha_i \in frontier(C, E)) \Rightarrow C(B) = True$

True Event Graph

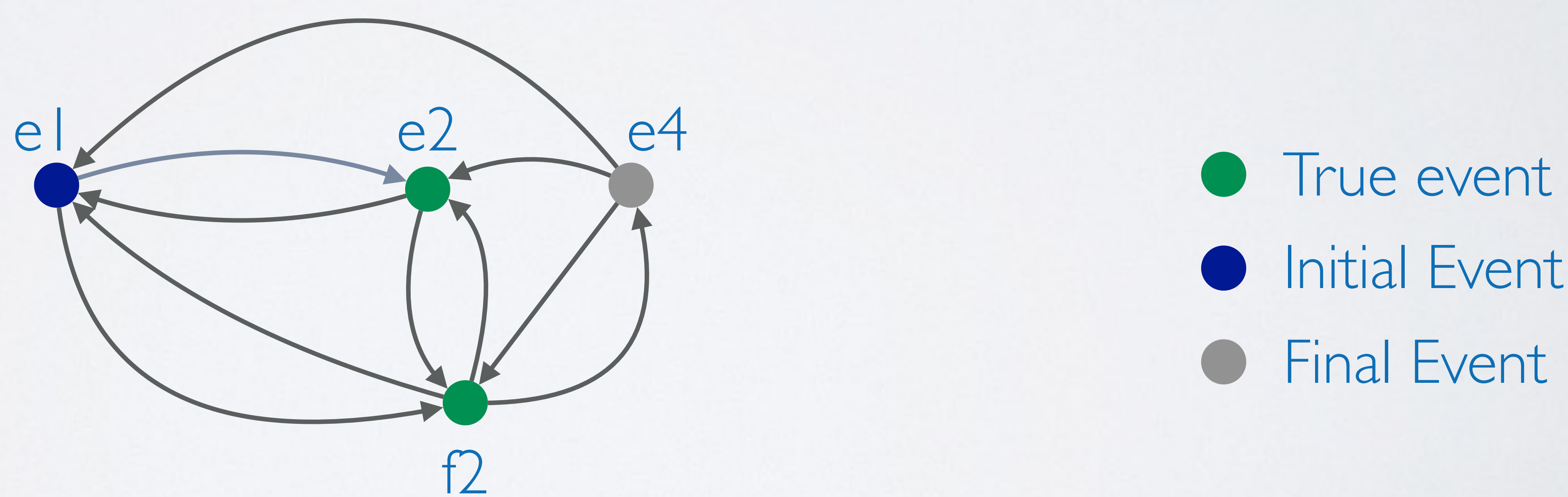
For a given computation $(E, \rightarrow)$ and a predicate B , a true event graph is defined as:

1. An event e An event e belongs to the true event graph if the local predicate for the process *e.procc* is true at the event e.
2. There exists an edge between two events e and f if e.succ $\twoheadrightarrow$ f
3. An event e in the true event graph is labeled as initial if $e \in E.\perp$
4. An event e in the true event graph is labeled as "final: if $e \in E.\top$

True Event Graph



Computation (E, →)



True Event Graph

Theorems

Theorem 1

A predicate B is controllable in a computation (E, $\rightarrow$) iff there exists an admissible sequence of events with respect to the predicate B and the computation (E, $\rightarrow$)

Theorem 2

Let G (v, E) be the true event graph corresponding to the computation (E, $\rightarrow$) and a predicate B. The shortest path in G, if it exists, corresponds to the admissible sequence of events in the computation

Theorem 3

Let G (v, E) be the true event graph corresponding to the computation (E, $\rightarrow$) and a predicate B. B is controllable if in the computation (E, $\rightarrow$) then there exists a permissible path in G.

Solution

Computation Trace

↓

True Event Graph

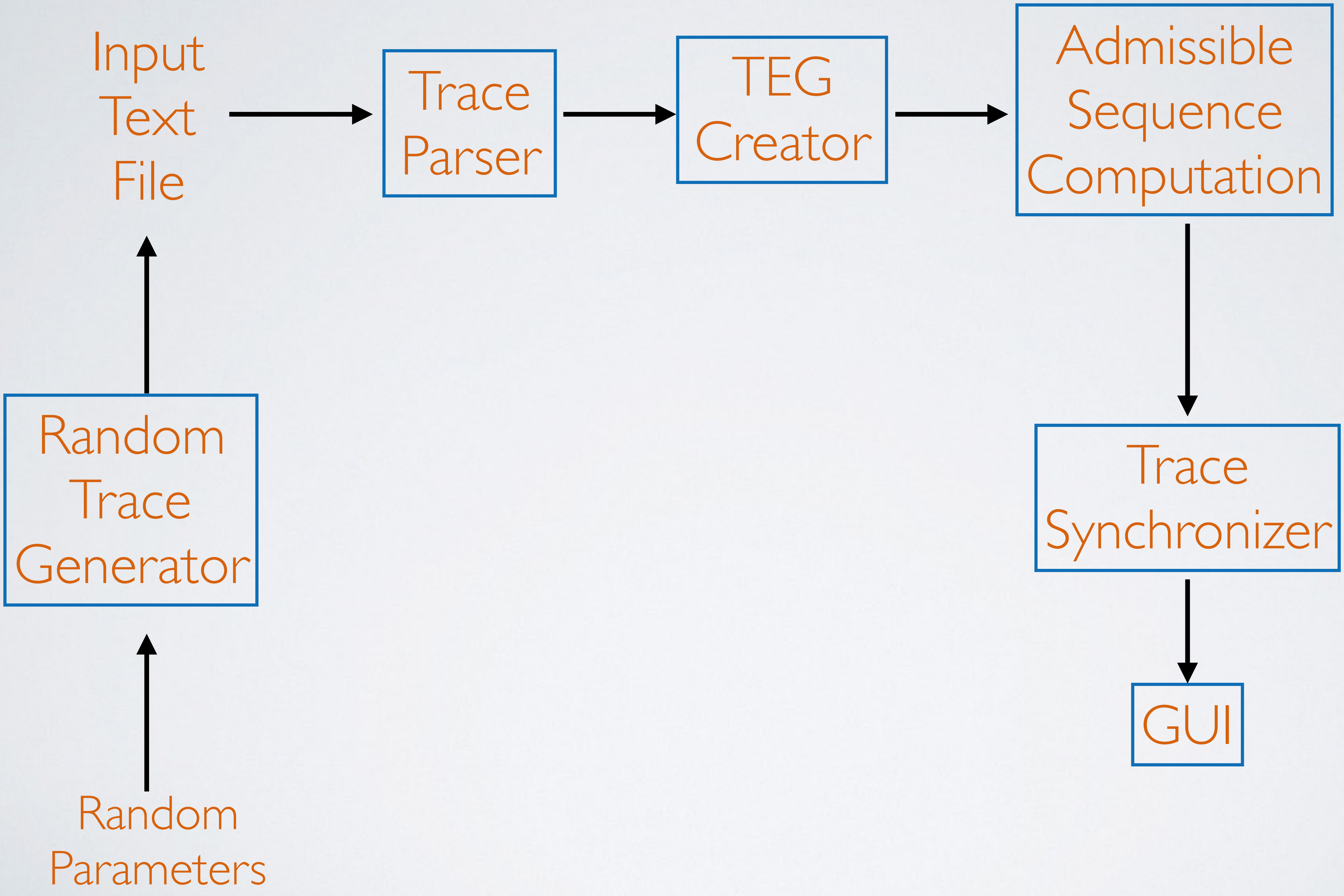
↓

Admissible Sequence

↓

Recompute Controlled Computation Trace

Java Framework



Admissible Sequence Algorithm

```
findAdmissibleSequence(trace):
 graph = constructTEG(trace)
 admissible_seq = NULL
 min_length = INFINITY
 for e in graph.initial_nodes
  for f in graph.final_nodes
   path = graph.shortest_path(e, f)
   if (length(path) < min_length)
    admissible_seq = path;
    min_length = length(admissible_seq)
   end if
  end for
 end for
 return admissible_seq
```

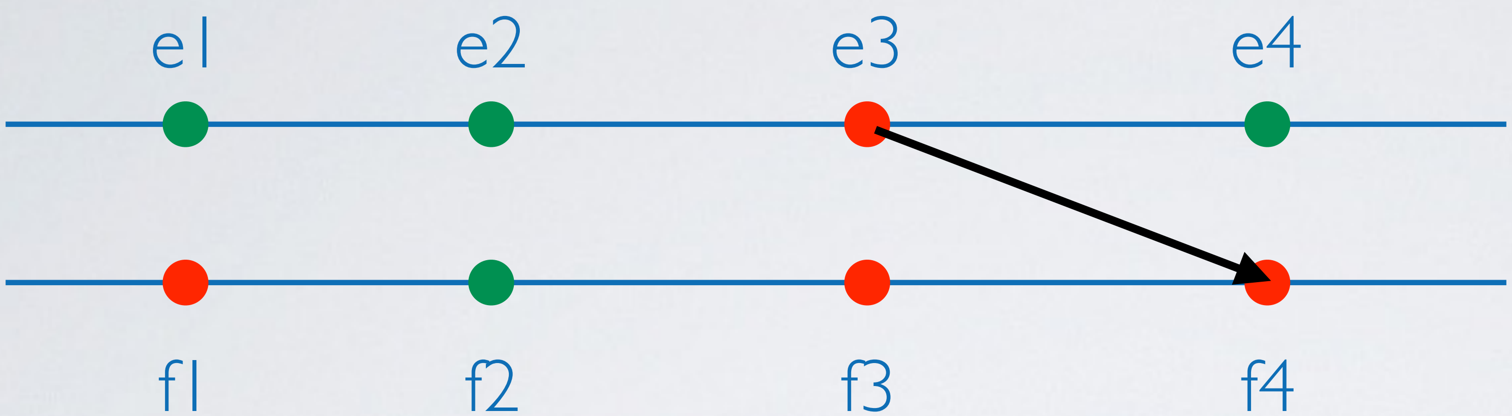
Time Complexity: $O((nm)^3)$

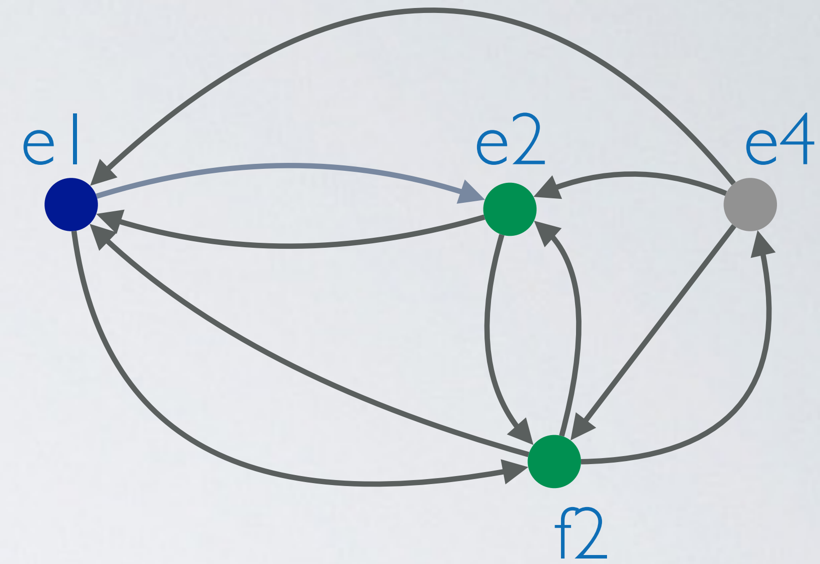
Space Complexity: $O((nm)^2)$

Trace Synchronizer Algorithm

```
addSyncrhonizationMessages(admissibleS_seq):
 // add messages from set A in section II
 for i in [0,|α|)
  for j in (i,|α|)
   if (αi.process ≠ αj.process)
    addSyncMessage(αi, αj)
   end if
  end for
 end for
 // add messages from set B in section II
 for i in [0, |α|-1)
  if (αi.process ≠ αi+1.process)
   addSyncMessage(αi+1, αi.next)
  end if
 end for
```

True Event Graph

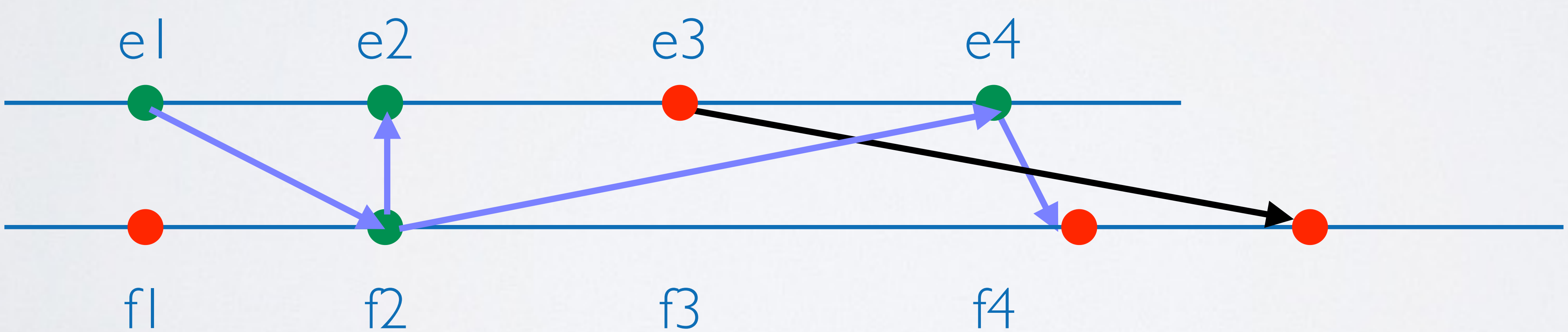




Original Computation

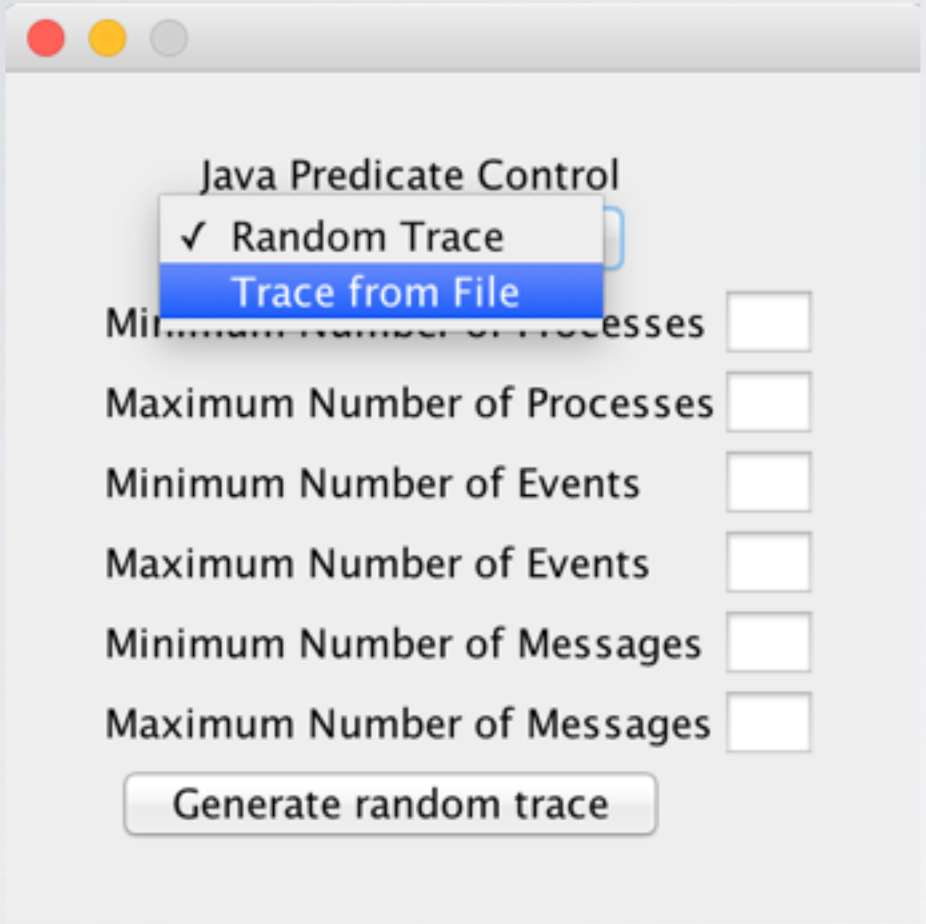
TEG

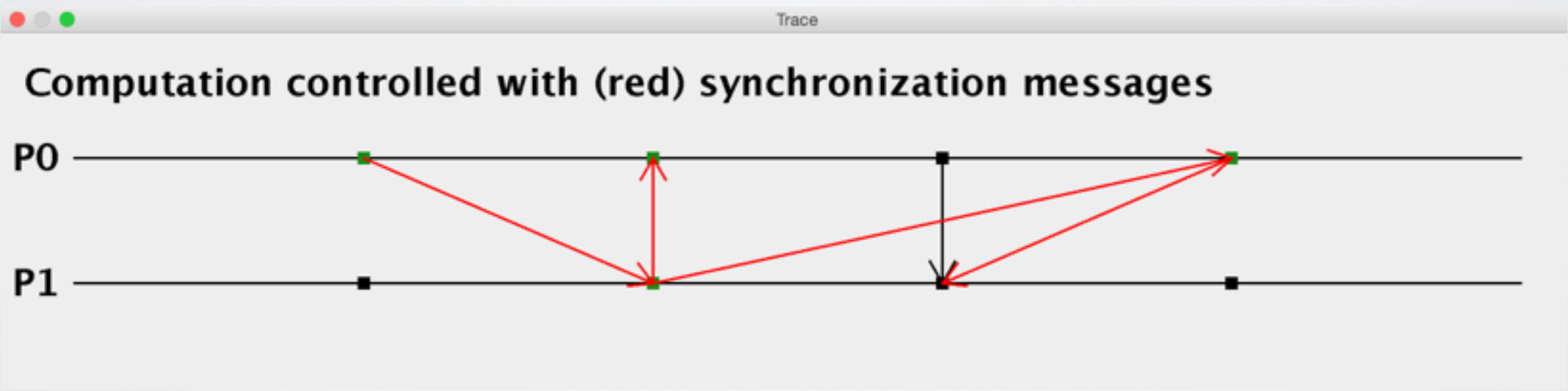
Admissible Sequence: e1, f2, e4



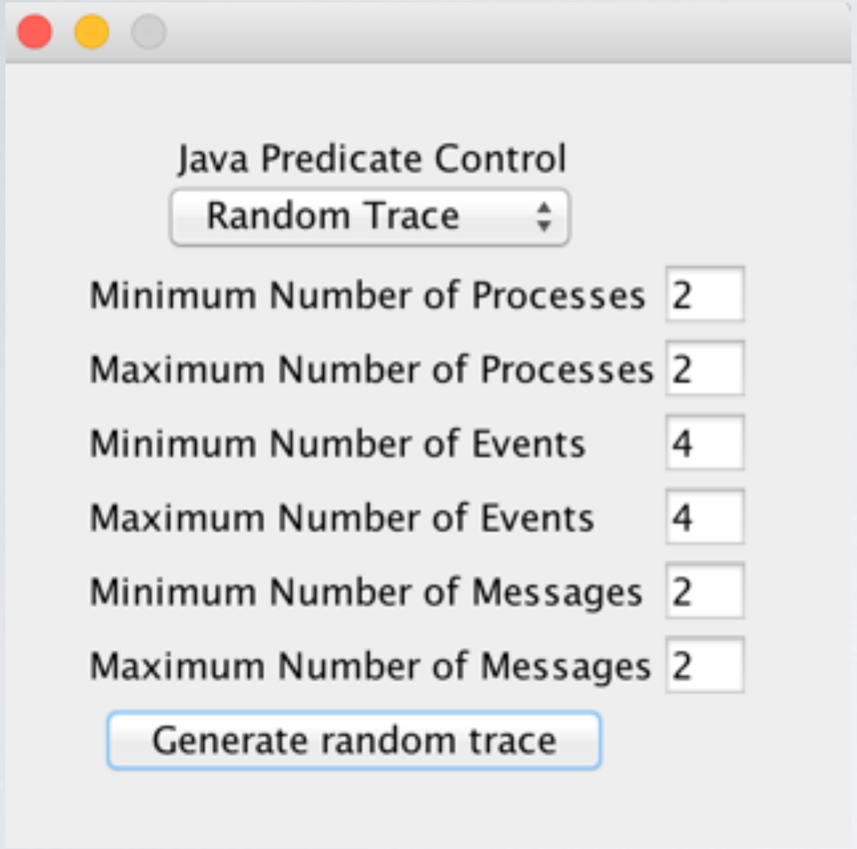
Controlled Computation

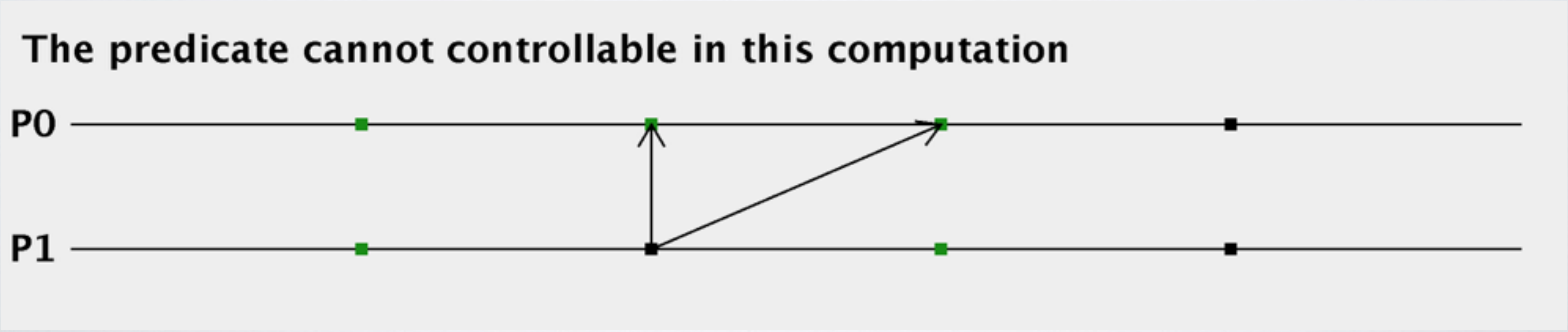
Java Framework



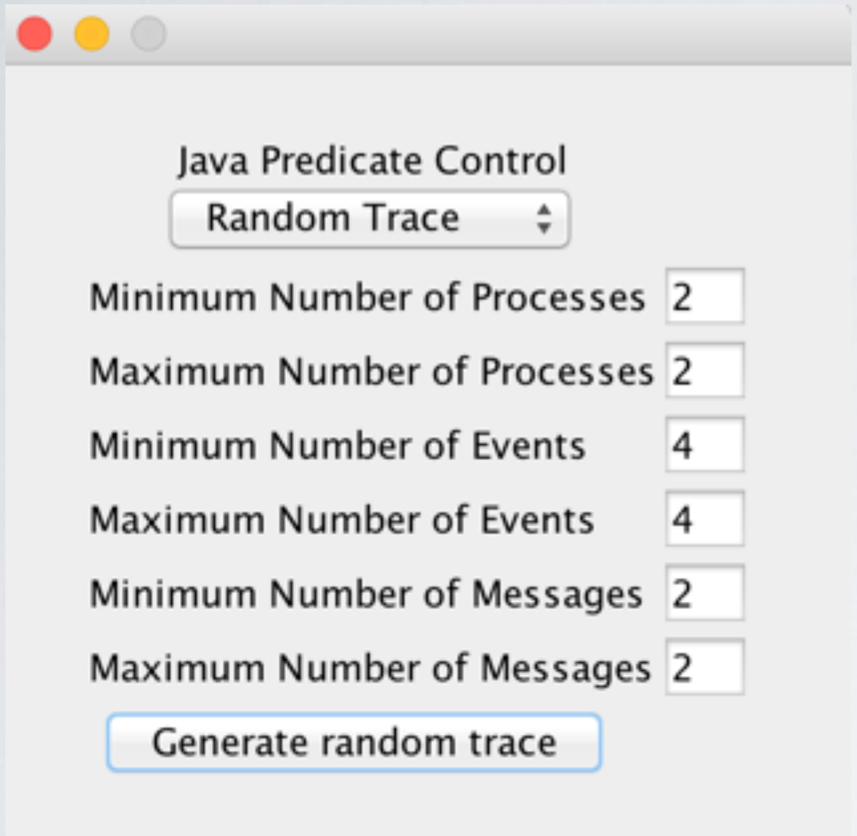


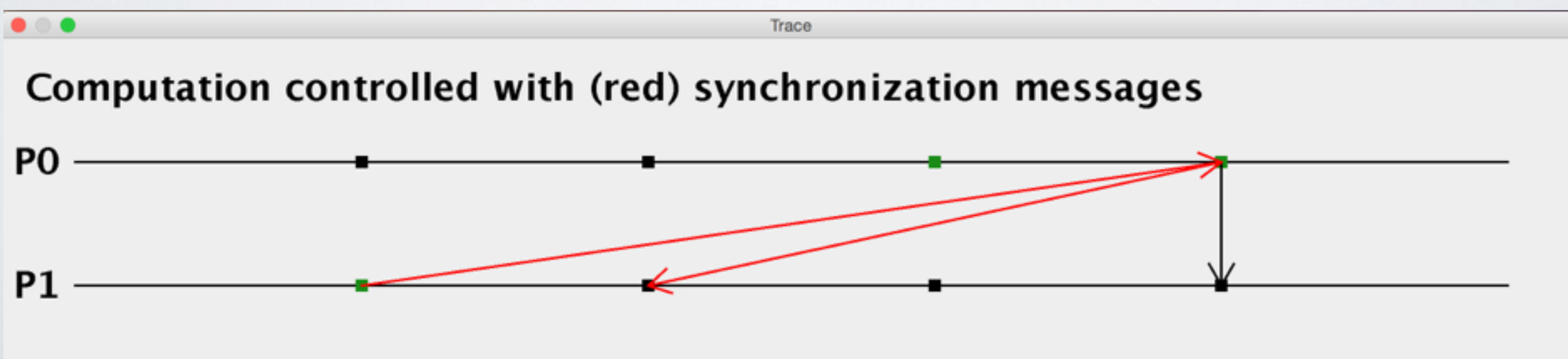
Java Framework





Java Framework





Thank You!