Debugging Distributed Systems with Why-Across-Time Provenance

Michael Whittaker, Cristina Teodoropol, Peter Alvaro,
Joseph M. Hellerstein

Reasoning about the causes of events in a distributed system is hard

Causality

Operating Systems

R. Stockton Gaines Editor

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.

Key Words and Phrases: distributed systems, computer networks, clock synchronization, multiprocess systems

CR Categories: 4.32, 5.29

Introduction

The concept of time is fundamental to our way of thinking. It is derived from the more basic concept of the order in which events occur. We say that something happened at 3:15 if it occurred *after* our clock read 3:15 and *before* it read 3:16. The concept of the temporal ordering of events pervades our thinking about systems. For example, in an airline reservation system we specify that a request for a reservation should be granted if it is made *before* the flight is filled. However, we will see that this concept must be carefully reexamined when considering events in a distributed system.

A distributed system consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages. A network of interconnected computers, such as the ARPA net, is a distributed system. A single computer can also be viewed as a distributed system in which the central control unit, the memory units, and the input-output channels are separate processes. A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process.

We will concern ourselves primarily with systems of spatially separated computers. However, many of our remarks will apply more generally. In particular, a multiprocessing system on a single computer involves problems similar to those of a distributed system because of the unpredictable order in which certain events can occur.

In a distributed system, it is sometimes impossible to say that one of two events occurred first. The relation "happened before" is therefore only a partial ordering of the events in the system. We have found that problems often arise because people are not fully aware of this fact and its implications.

In this paper, we discuss the partial ordering defined by the "happened before" relation, and give a distributed algorithm for extending it to a consistent total ordering of all the events. This algorithm can provide a useful mechanism for implementing a distributed system. We illustrate its use with a simple method for solving synchronization problems. Unexpected, anomalous behavior can occur if the ordering obtained by this algorithm differs from that perceived by the user. This can be avoided by introducing real, physical clocks. We describe a simple method for synchronizing these clocks, and derive an upper bound on how far out of synchrony they can drift.

The Partial Ordering

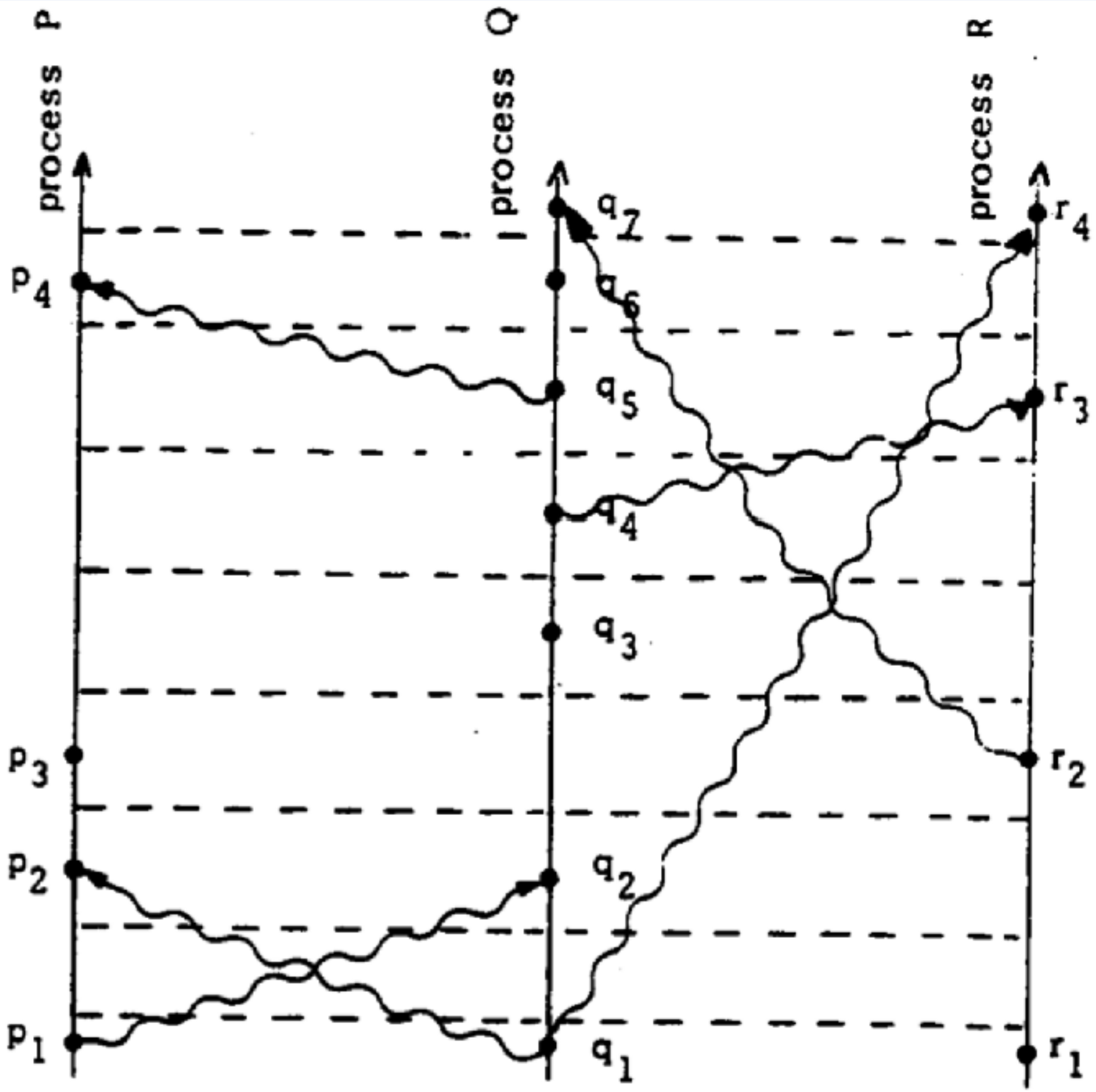
Most people would probably say that an event *a* happened before an event *b* if *a* happened at an earlier time than *b*. They might justify this definition in terms of physical theories of time. However, if a system is to meet a specification correctly, then that specification must be given in terms of events observable within the system. If the specification is in terms of physical time, then the system must contain real clocks. Even if it does contain real clocks, there is still the problem that such clocks are not perfectly accurate and do not keep precise physical time. We will therefore define the "happened before" relation without using physical clocks.

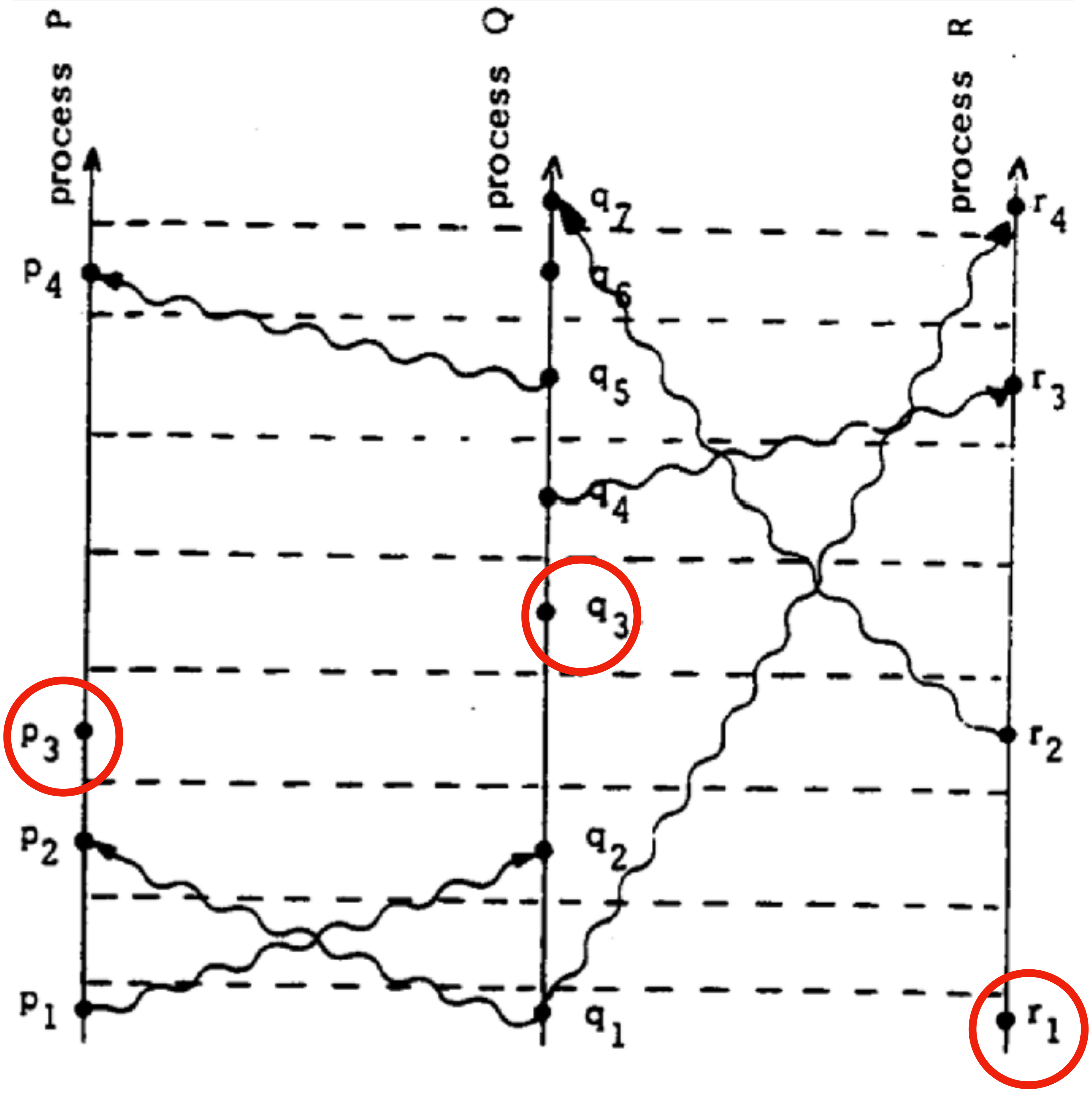
We begin by defining our system more precisely. We assume that the system is composed of a collection of processes. Each process consists of a sequence of events. Depending upon the application, the execution of a subprogram on a computer could be one event, or the execution of a single machine instruction could be one

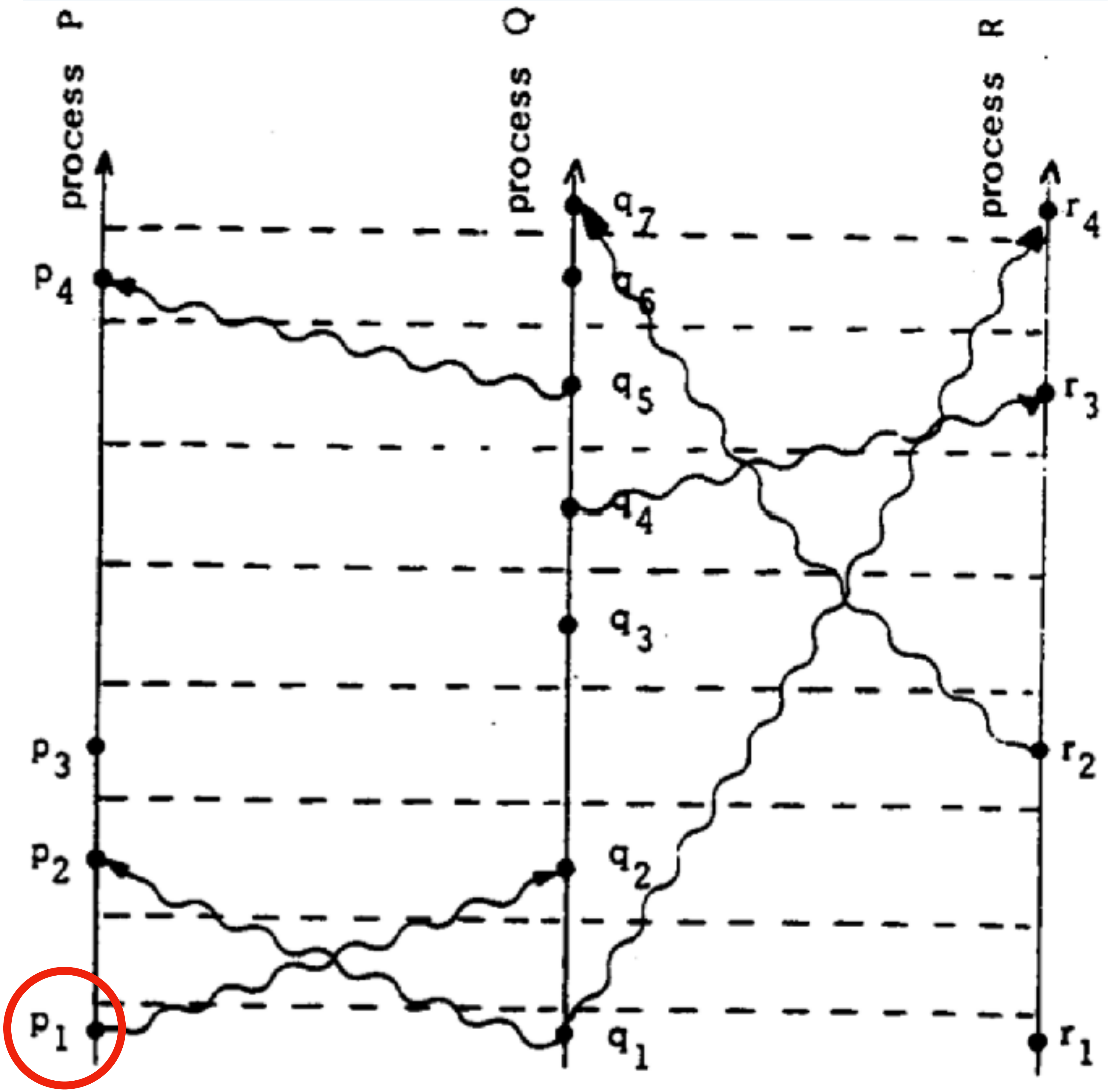
---

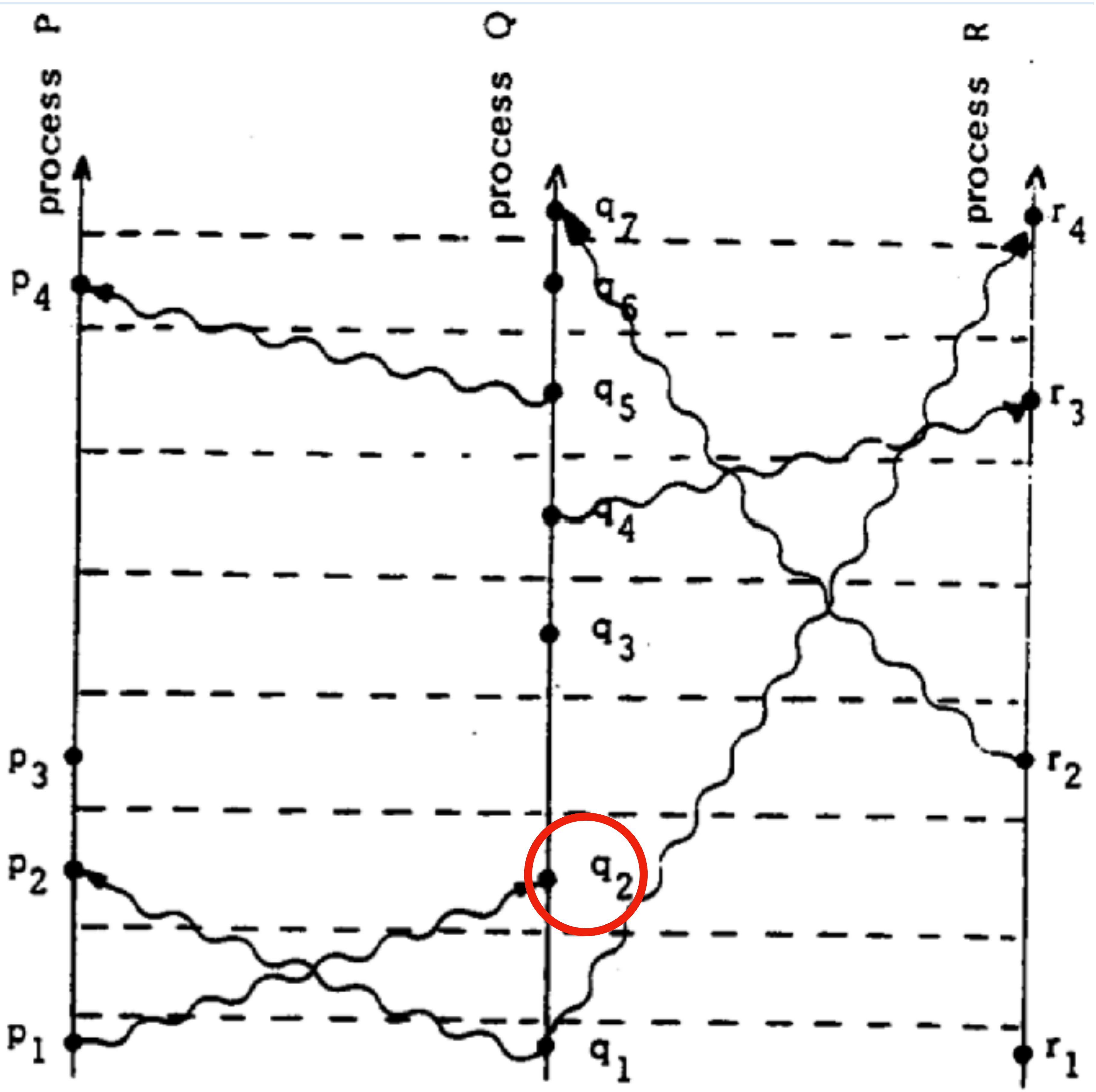
This work was supported by the Advanced Research Projects Agency of the Department of Defense and Rome Air Development Center. It was monitored by Rome Air Development Center under contract number F 30602-76-C-0094.

Author's address: Computer Science Laboratory, SRI International, 333 Ravenswood Ave., Menlo Park CA 94025.

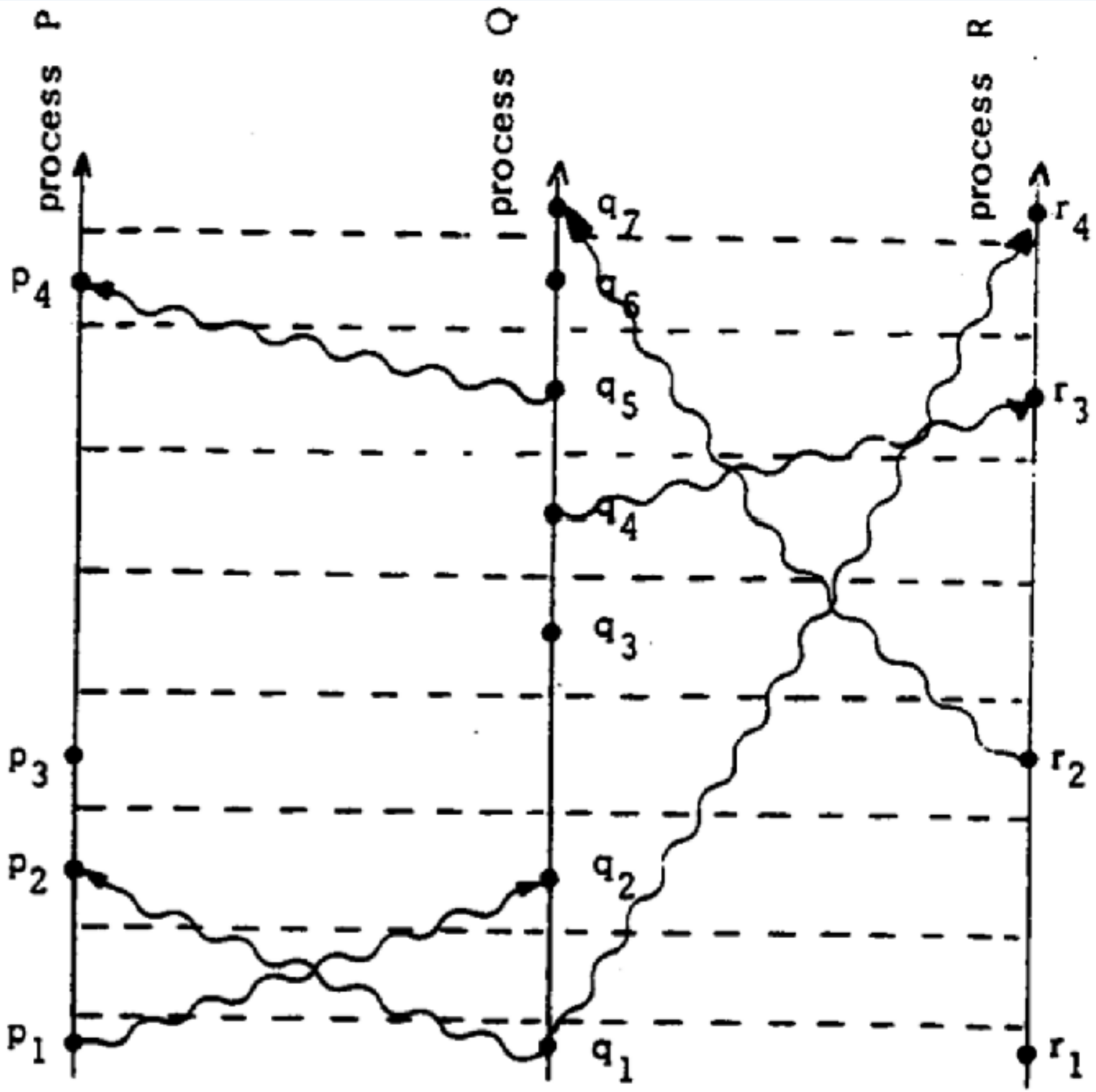

process P
process Q
process R
p_1
p_2
p_3
p_4
q_1
q_2
q_3
q_4
q_5
q_6
q_7
r_1
r_2
r_3
r_4

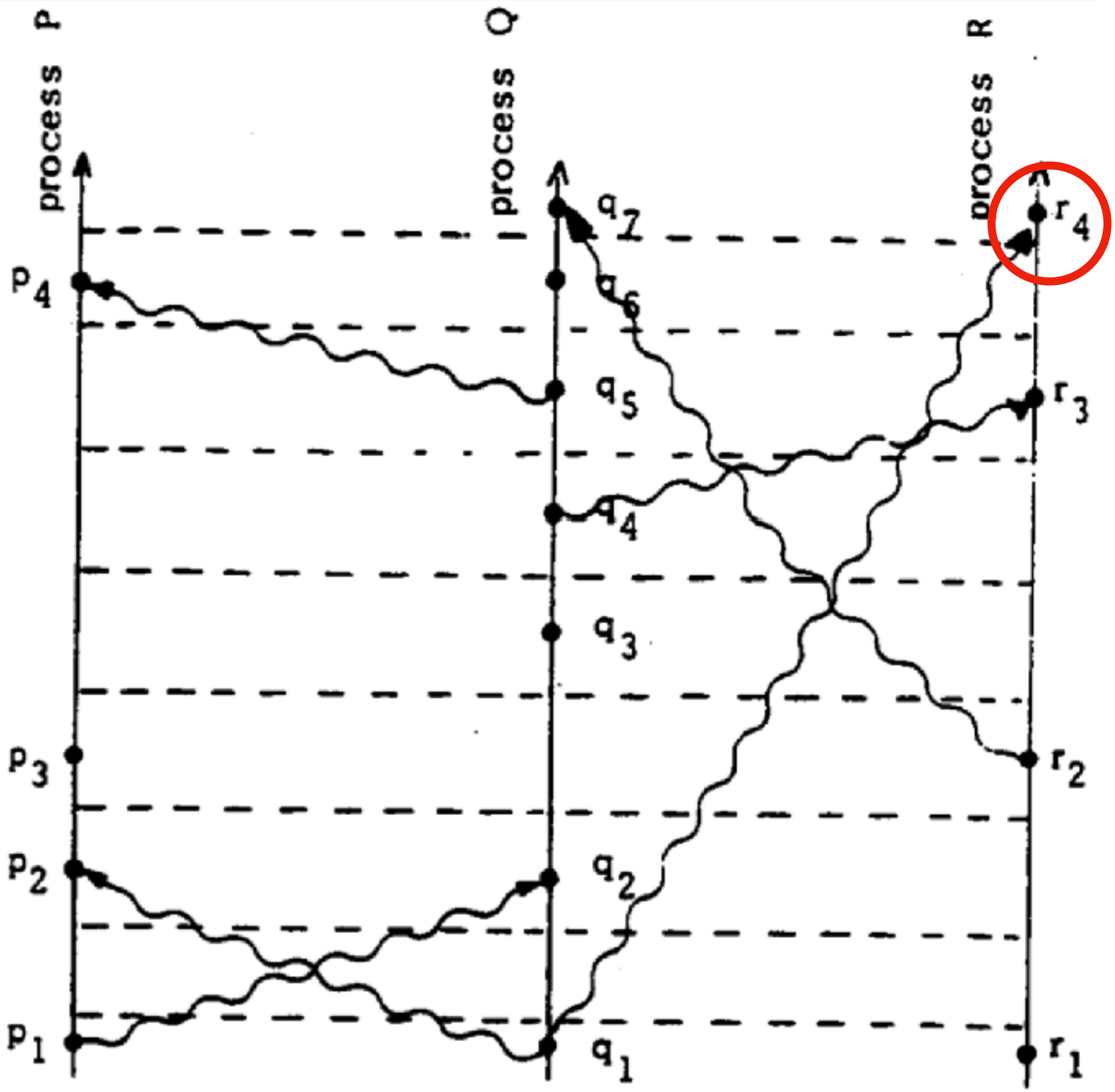

process P
process Q
process R
p1
p2
p3
p4
q1
q2
q3
q4
q5
q6
q7
r1
r2
r3
r4

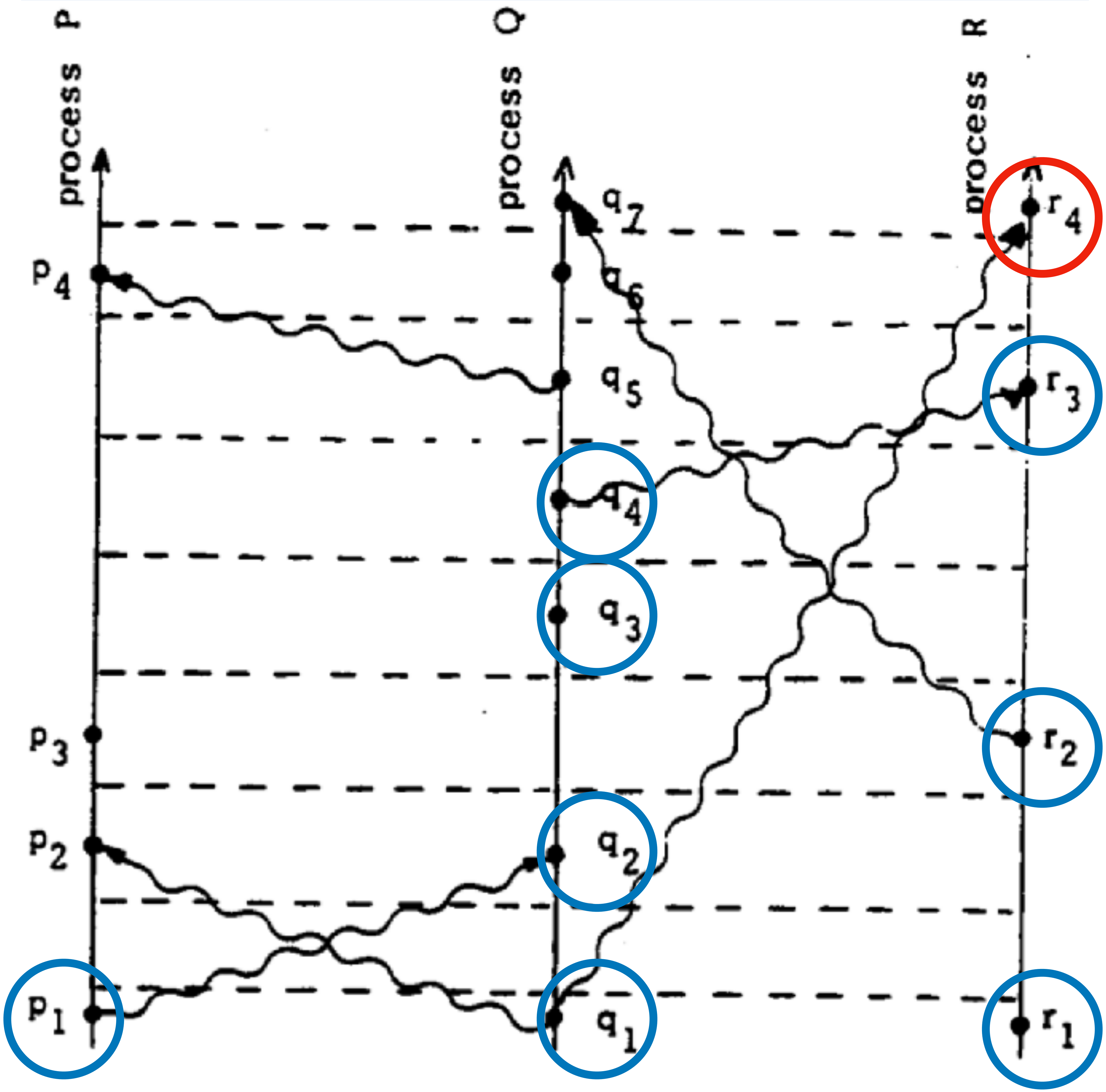

process P
process Q
process R
p_4
p_3
p_2
p_1
q_7
q_6
q_5
q_4
q_3
q_2
q_1
r_4
r_3
r_2
r_1



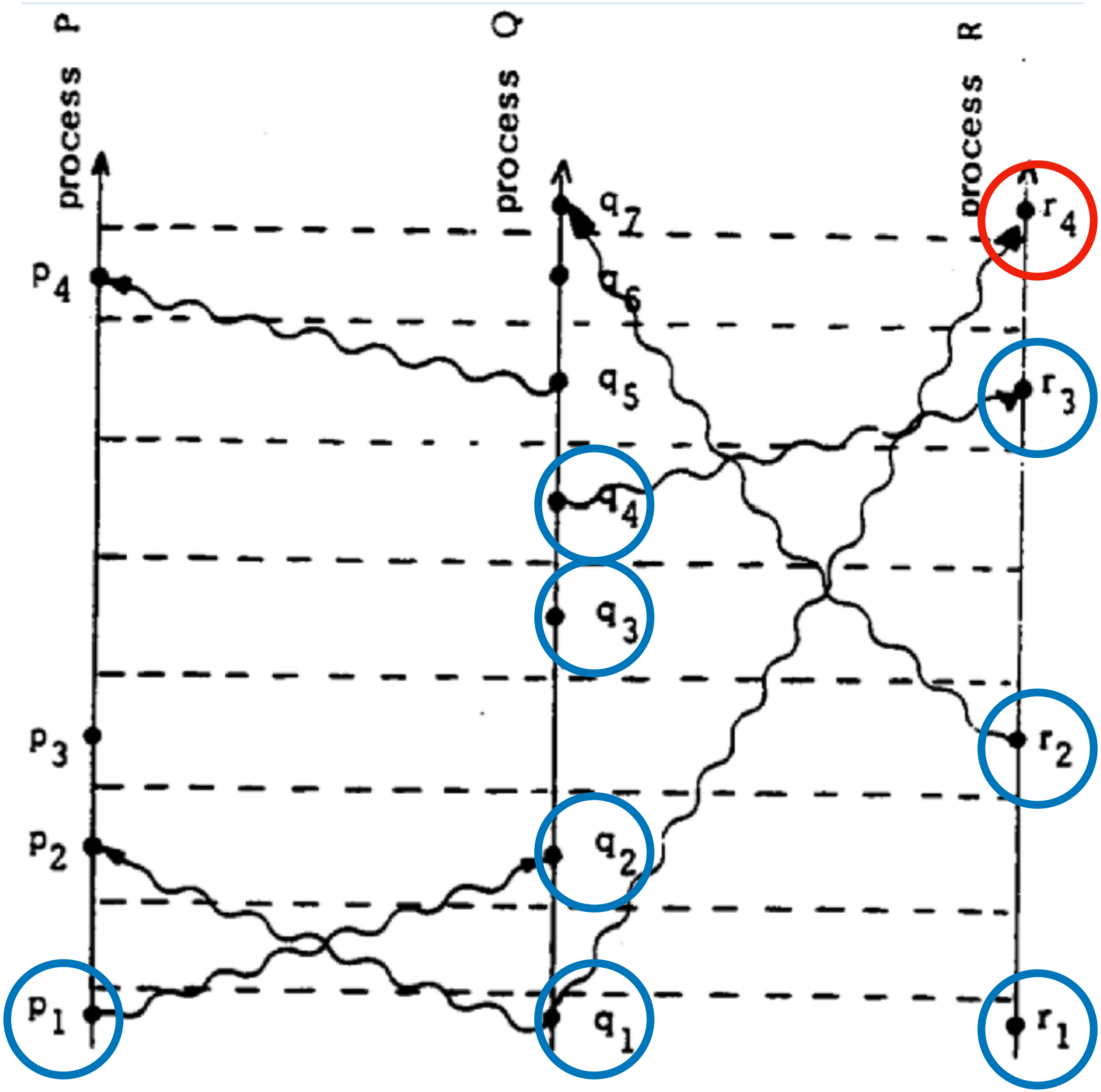
process P
process Q
process R
p1
p2
p3
p4
q1
q2
q3
q4
q5
q6
q7
r1
r2
r3
r4


process P
process Q
process R
p4
p3
p2
p1
q7
q6
q5
q4
q3
q2
q1
r4
r3
r2
r1


process P
process Q
process R
p_1
p_2
p_3
p_4
q_1
q_2
q_3
q_4
q_5
q_6
q_7
r_1
r_2
r_3
r_4


process P
process Q
process R
p_1
p_2
p_3
p_4
q_1
q_2
q_3
q_4
q_5
q_6
q_7
r_1
r_2
r_3
r_4

Causality is general-purpose
but too coarse-grained


process P
process Q
process R
p1
p2
p3
p4
q1
q2
q3
q4
q5
q6
q7
r1
r2
r3
r4

Why-Provenance

Why-Provenance

R

	x	y
r1	10	20
r2	10	40
r3	600	700

S

	y	z
s1	20	30
s2	40	50
s3	700	800

Database Instance *I*

Why-Provenance

R

	x	y
r1	10	20
r2	10	40
r3	600	700

S

	y	z
s1	20	30
s2	40	50
s3	700	800

Database Instance *I*

```
SELECT x
  FROM R, S
 WHERE R.y = S.y
```

SQL Query *Q*

Why-Provenance

R

	x	y
r1	10	20
r2	10	40
r3	600	700

S

	y	z
s1	20	30
s2	40	50
s3	700	800

Database Instance *I*

```
SELECT x
  FROM R, S
 WHERE R.y = S.y
```

SQL Query *Q*

x
10
600

Output *Q(I)*

Why-Provenance

R

	x	y
r1	10	20
r2	10	40
r3	600	700

S

	y	z
s1	20	30
s2	40	50
s3	700	800

Database Instance *I*

```
SELECT x
  FROM R, S
 WHERE R.y = S.y
```

SQL Query *Q*

x
10
600

Output Tuple *t*

Output *Q(I)*

A subinstance *I’* of *I* is a witness of *t* if *t* is in *Q(I’)*

Why-Provenance

R

	x	y
r1	10	20
r2	10	40
r3	600	700

S

	y	z
s1	20	30
s2	40	50
s3	700	800

Database Subinstance *I’*

```
SELECT x
  FROM R, S
 WHERE R.y = S.y
```

SQL Query *Q*

Why-Provenance

R

	x	y
r1	10	20
r2	10	40
r3	600	700

S

	y	z
s1	20	30
s2	40	50
s3	700	800

Database Subinstance *I’*

```
SELECT x
  FROM R, S
 WHERE R.y = S.y
```

SQL Query *Q*

x
10

Output Tuple *t*

Output *Q(I’)*

A witness *I'* of *t* is a minimal witness of *t* if no proper subinstance of *I'* is also a witness of *t*

Why-Provenance

R

	x	y
r1	10	20
r2	10	40
r3	600	700

S

	y	z
s1	20	30
s2	40	50
s3	700	800

Database Subinstance *I’*

```
SELECT x
  FROM R, S
 WHERE R.y = S.y
```

SQL Query *Q*

x
10

Output Tuple *t*

Output *Q(I’)*

Why-Provenance

R

	x	y
r1	10	20
r2	10	40
r3	600	700

S

	y	z
s1	20	30
s2	40	50
s3	700	800

Database Subinstance *I’*

```
SELECT x
  FROM R, S
 WHERE R.y = S.y
```

SQL Query *Q*

x
10

Output Tuple *t*

Output *Q(I’)*

Why-Provenance

R

	x	y
r1	10	20
r2	10	40
r3	600	700

S

	y	z
s1	20	30
s2	40	50
s3	700	800

Database Subinstance *I’*

```
SELECT x
  FROM R, S
 WHERE R.y = S.y
```

SQL Query *Q*

x
10

Output Tuple *t*

Output *Q(I’)*

The why-provenance of *t*
is the set of minimal
witnesses of *t*

Why-provenance is fine-grained but not generally applicable

Causality is general-purpose but too coarse-grained

Why-provenance is fine-grained but not generally applicable

Wat-provenance is general-purpose *and* fine-grained

Wat-provenance generalizes why-provenance from static relational databases to arbitrary state machines

Wat-provenance is to state machines what why-provenance is to relational databases

Wat-Provenance

Wat-Provenance

R

x	y
10	20
10	40
700	800

Database Instance I

Wat-Provenance

R

x	y
10	20
10	40
700	800

Database Instance I

```
SELECT x
  FROM R, S
 WHERE R.y = S.y
```

Query Q

Wat-Provenance

R

x	y
10	20
10	40
700	800

Database Instance I

```
SELECT x
  FROM R, S
 WHERE R.y = S.y
```

Query Q

x
10
700

Output Tuple t

Wat-Provenance

R

x	y
10	20
10	40
700	800

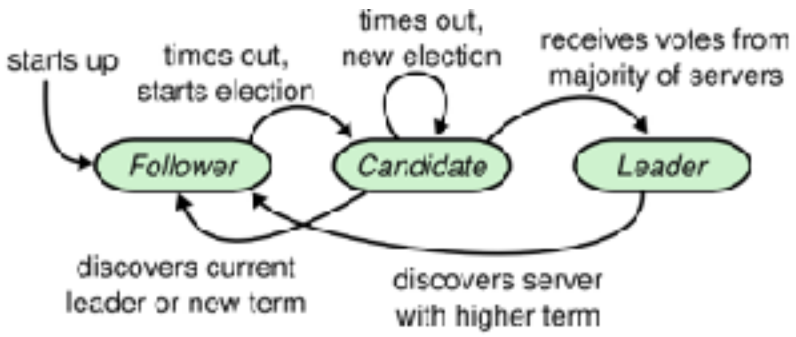
Database Instance I

```
SELECT x
  FROM R, S
 WHERE R.y = S.y
```

Query Q

x
10
700

Output Tuple t



State Machine M

Wat-Provenance

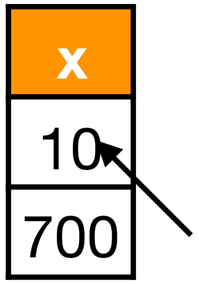
R

x	y
10	20
10	40
700	800

Database Instance I

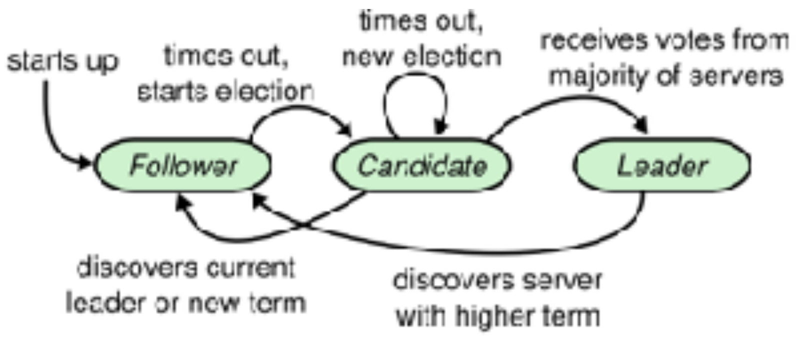
```
SELECT x
  FROM R, S
 WHERE R.y = S.y
```

Query Q



x
10
700

Output Tuple t



State Machine M

a	b	c	d

Input Trace T

Wat-Provenance

R

x	y
10	20
10	40
700	800

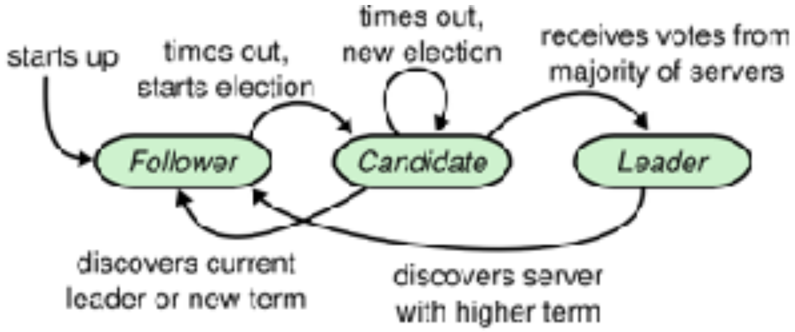
Database Instance I

```
SELECT x
  FROM R, S
 WHERE R.y = S.y
```

Query Q

x
10
700

Output Tuple t



State Machine M

a	b	c	d

Input Trace T

```
get(x); 1
```

Input i, Output o

Example 1: Key-Value Store

Example 1: Key-Value Store

`Trace T`

`set(x,1)`	`set(y,2)`

Example 1: Key-Value Store

`Trace T` `set(x,1)` `set(y,2)`

`Input i` `get(x)`

Example 1: Key-Value Store

`Trace T`	`set(x,1)`	`set(y,2)`
`Input i`	`get(x)`	
`Output o`	`1`	

Example 1: Key-Value Store

Trace T

`set(x,1)` `set(y,2)`

Input i `get(x)`

Output o `1`

Lessons

1. **The “cause” of an output *o* should be a subtrace of the input that suffices to generate *o*. We call such a subtrace a witness of *o*.**

Example 1: Key-Value Store

Trace T

`set(x,1)` `set(y,2)`

Input i

`get(x)`

Output o

`1`

Example 1: Key-Value Store

`Trace T`	`set(x,1)` `set(y,2)`
`Input i`	`get(x)`
`Output o`	`1`

Lessons

1. The “cause” of an output *o* is a subtrace of the input that suffices to generate *o*. We call such a subtrace a witness of *o*.

2. **The cause of an output *o* should be a “minimal” witness of *o*.**

Example 1: Key-Value Store

Trace T	`set(x,1)`	`set(y,2)`
Input i	`get(x)`	
Output o	1	

Example 1: Key-Value Store

`Trace T` `set(x,1)` `set(y,2)`

`Input i` `get(x)`

`Output o` `1`

Example 2: Boolean Formulas

Example 2: Boolean Formulas

`Trace T`	`set(a)`	`set(b)`	`set(c)`	`set(d)`

Example 2: Boolean Formulas

`Trace T`	`set(a)`	`set(b)`	`set(c)`	`set(d)`

`Input i` `eval((a and d) or (b and c))`

Example 2: Boolean Formulas

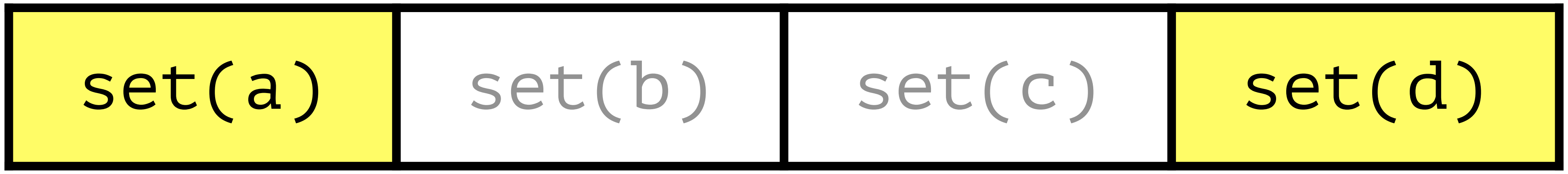
`Trace T`	`set(a)`	`set(b)`	`set(c)`	`set(d)`

`Input i` `eval((a and d) or (b and c))`

`Output o` `true`

Example 2: Boolean Formulas

Trace T



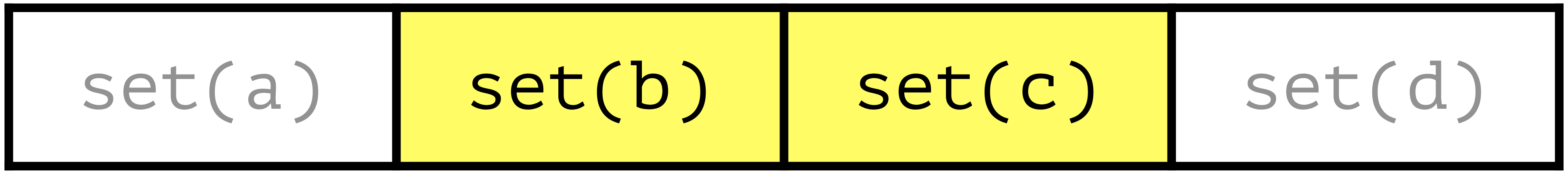
set(a) set(b) set(c) set(d)

Input i eval((a and d) or (b and c))

Output o true

Example 2: Boolean Formulas

`Trace T`



`set(a)` `set(b)` `set(c)` `set(d)`

`Input i` `eval((a and d) or (b and c))`

`Output o` `true`

Lessons

1. The “cause” of an output *o* is a subtrace of the input that suffices to generate *o*. We call such a subtrace a witness of *o*.

2. The cause of an output *o* should be a “minimal” witness of *o*.

3. **An output *o* could have multiple “minimal” witnesses.**

Example 3: Negation

`Trace T`	`set(a)`	`set(b)`	`set(c)`

`Input i` `eval((a and not b) or c)`

Example 3: Negation

`Trace T`

`set(a)`	`set(b)`	`set(c)`

`Input i` `eval((a and not b) or c)`

Example 3: Negation

Trace T			
	`set(a)`	`set(b)`	`set(c)`

Input i `eval((a and not b) or c)`

Example 3: Negation

`Trace T` `set(a)` `set(b)` `set(c)`

`Input i` `eval((a and not b) or c)`

Example 3: Negation

Trace T	`set(a)`	`set(b)`	`set(c)`

Input i `eval((a and not b) or c)`

Example 3: Negation

Trace T	set(a)	set(b)	set(c)

Input i `eval((a and not b) or c)`

Output o `true`

Example 3: Negation

Trace T	set(a)	set(b)	set(c)

Input i `eval((a and not b) or c)`

Output o `true`

Example 3: Negation

Trace T	`set(a)`	`set(b)`	`set(c)`

Input i `eval((a and not b) or c)`

Output o `true`

Example 3: Negation

Trace T

set(a)	set(b)	set(c)

Input i `eval((a and not b) or c)`

Output o `true`

Example 3: Negation

Trace T

`set(a)` `set(b)` `set(c)`

Input i `eval((a and not b) or c)`

Output o `true`

Example 3: Negation

Trace T

Input i `eval((a and not b) or c)`

Output o `true`

Example 3: Negation

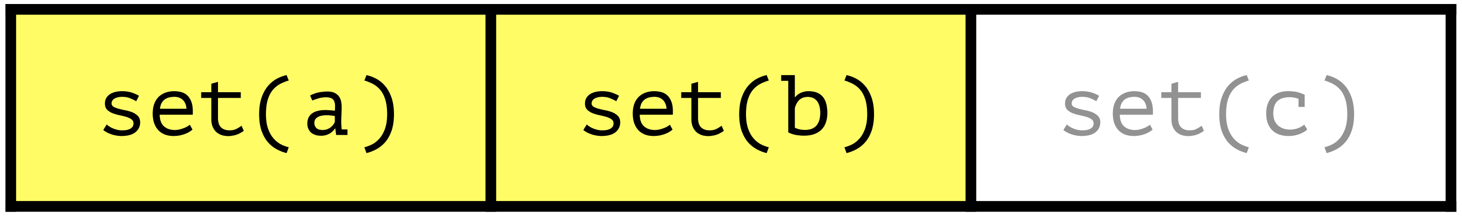
Trace T	set(a)	set(b)	set(c)

Input i `eval((a and not b) or c)`

Output o `true`

Example 3: Negation

Trace T



Input i `eval((a and not b) or c)`

Output o `true`

Lessons

1. The “cause” of an output *o* is a subtrace of the input that suffices to generate *o*. We call such a subtrace a witness of *o*.

2. The cause of an output *o* should be a “minimal” witness of *o*.

3. An output *o* could have multiple “minimal” witnesses.

4. If a witness is a “cause” of an output *o*, then all supertraces of the witness should be too.

Example 3: Negation

Trace T

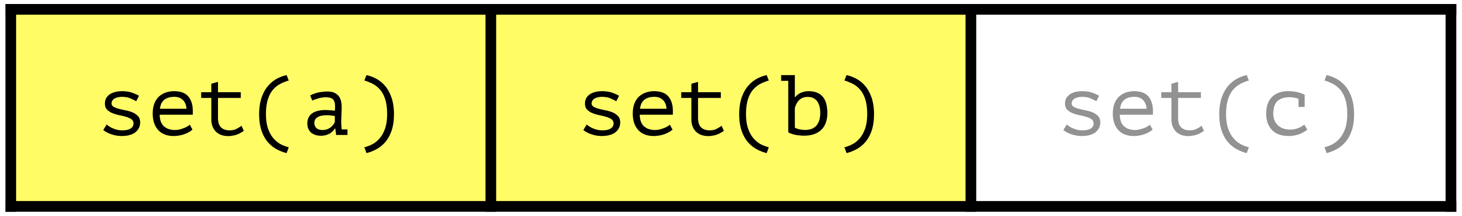
`set(a)` `set(b)` `set(c)`

Input i `eval((a and not b) or c)`

Output o `true`

Example 3: Negation

Trace T



Input i `eval((a and not b) or c)`

Output o `true`

Wat-Provenance

Wat-Provenance

- **Given state machine *M*, trace *T*, input *i*, and output *o*.**

Wat-Provenance

- **Given state machine *M*, trace *T*, input *i*, and output *o*.**
- **A witness of *o* is a subtrace of *T* that suffices to produce *o*.**

Wat-Provenance

- **Given state machine *M*, trace *T*, input *i*, and output *o*.**
- **A witness of *o* is a subtrace of *T* that suffices to produce *o*.**
- **A witness *T'* of *o* is closed under supertrace in *T* if every supertrace of *T'* in *T* is also a witness of *o*.**

Wat-Provenance

- **Given state machine *M*, trace *T*, input *i*, and output *o*.**
- **A witness of *o* is a subtrace of *T* that suffices to produce *o*.**
- **A witness *T'* of *o* is closed under supertrace in *T* if every supertrace of *T'* in *T* is also a witness of *o*.**
- **The wat-provenance of *o* is the set of minimal witnesses of *o* that are closed under supertrace in T.**

Causality is general-purpose but too coarse-grained

Why-provenance is fine-grained but not generally applicable

Wat-provenance is general-purpose *and* fine-grained

Computing wat-provenance

Causality is general-purpose but too coarse-grained

Why-provenance is fine-grained but not generally applicable

Wat-provenance is general-purpose *and* fine-grained

Causality is general-purpose but too coarse-grained and easy to compute

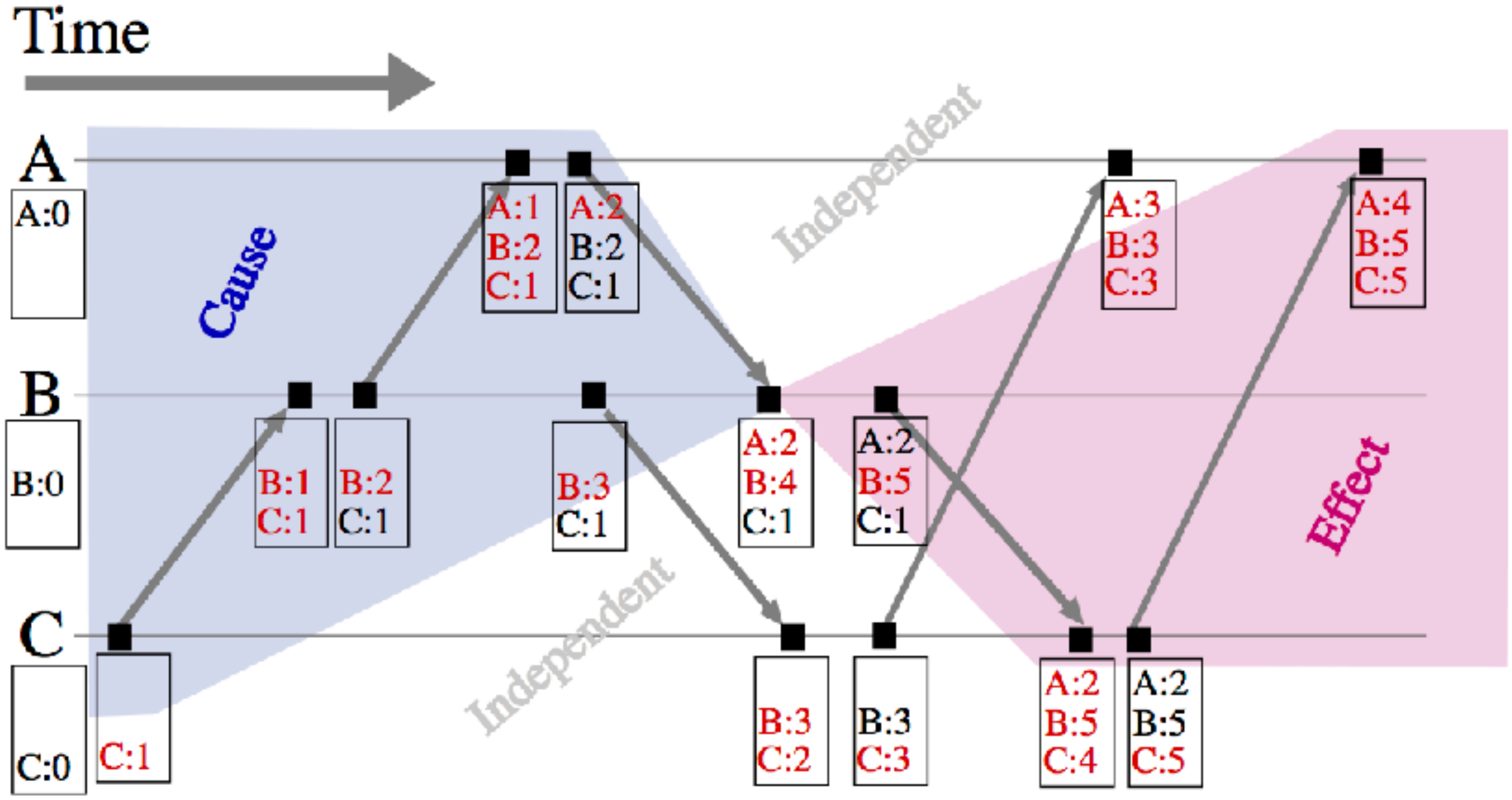
Why-provenance is fine-grained but not generally applicable and easy to compute

Wat-provenance is general-purpose *and* fine-grained but hard to compute

Computing Causal History

Computing Causal History

Use vector clocks



Computing Why-Provenance

Computing Why-Provenance

Compute it explicitly

$$\mathsf{Why}(\{t\}, I, \{u\}) = \begin{cases} \{\emptyset\}, & \text{if } (t = u), \\ \emptyset, & \text{otherwise.} \end{cases}$$

$$\mathsf{Why}(R, I, t) = \begin{cases} \{\{(R, t)\}\}, & \text{if } (t \in R(I)), \\ \emptyset, & \text{otherwise.} \end{cases}$$

$$\mathsf{Why}(\sigma_\theta(Q), I, t) = \begin{cases} \mathsf{Why}(Q, I, t), & \text{if } \theta(t), \\ \emptyset, & \text{otherwise.} \end{cases}$$

$$\mathsf{Why}(\pi_U(Q), I, t) = \bigcup\{\mathsf{Why}(Q, I, u) \mid u \in Q(I), t = u[U]\}$$

$$\mathsf{Why}(\rho_{A \mapsto B}(Q), I, t) = \mathsf{Why}(Q, I, t[B \mapsto A])$$

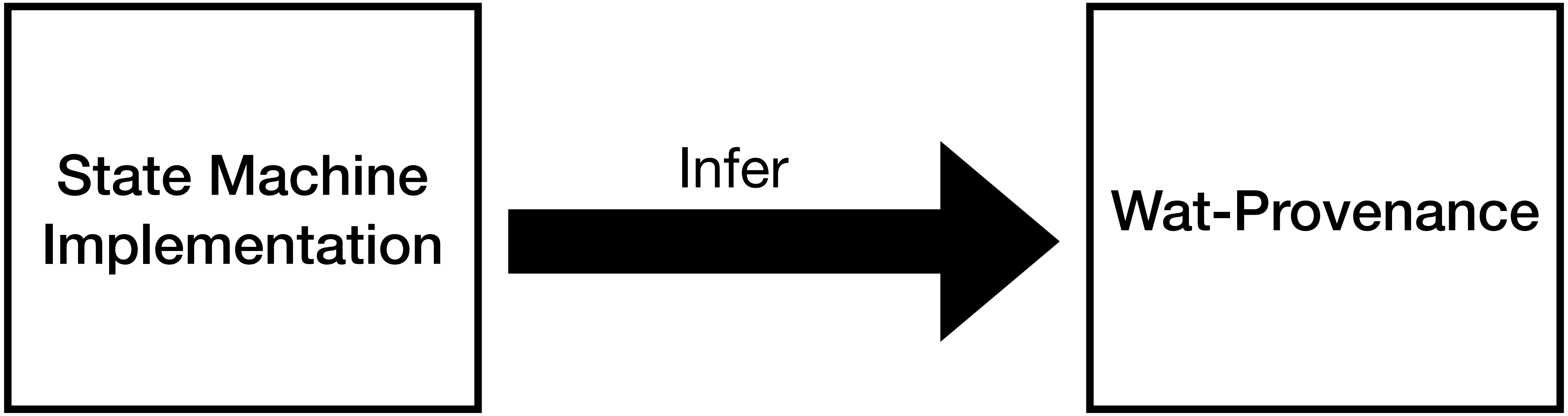
$$\mathsf{Why}(Q_1 \bowtie Q_2, I, t) = \mathsf{Why}(Q_1, I, t[U_1]) \uplus \mathsf{Why}(Q_2, I, t[U_2])$$

$$\mathsf{Why}(Q_1 \cup Q_2, I, t) = \mathsf{Why}(Q_1, I, t) \cup \mathsf{Why}(Q_2, I, t))$$

Computing Wat-Provenance

State Machine
Implementation

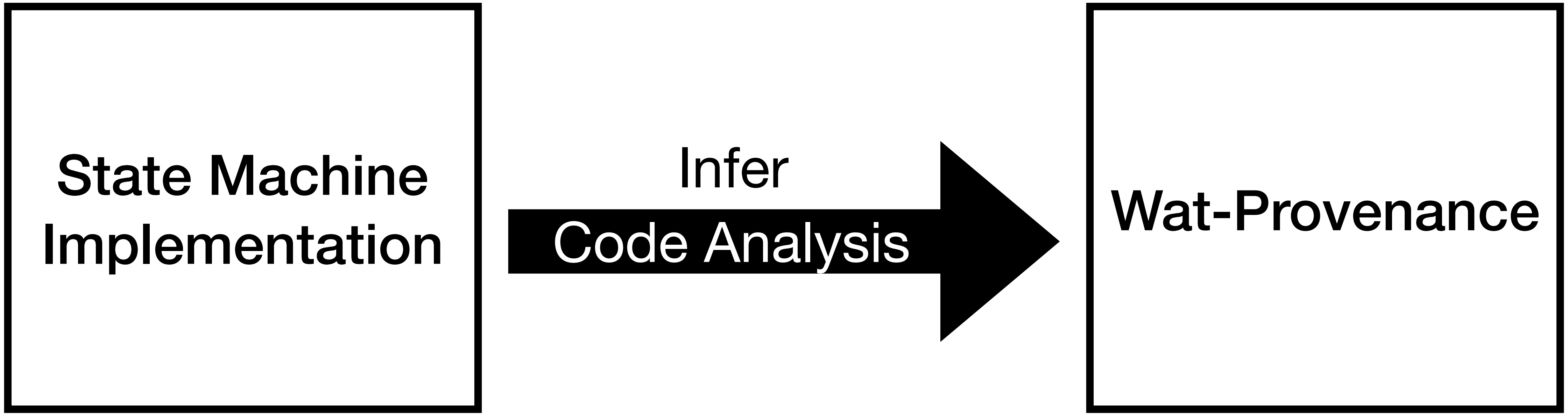
Computing Wat-Provenance


State Machine Implementation

Infer

Wat-Provenance

Computing Wat-Provenance



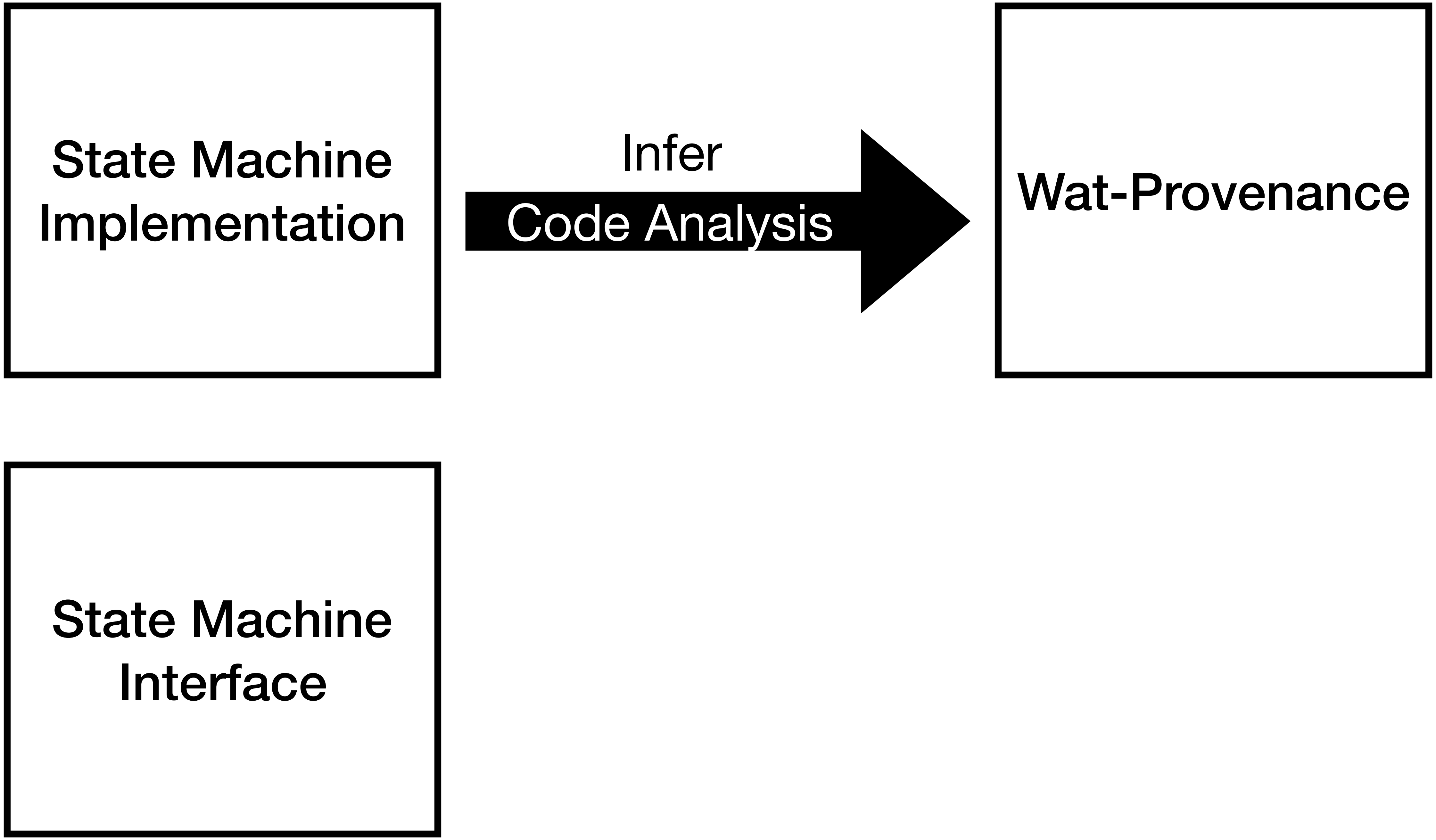
State Machine Implementation

Infer

Code Analysis

Wat-Provenance

Computing Wat-Provenance



State Machine Implementation

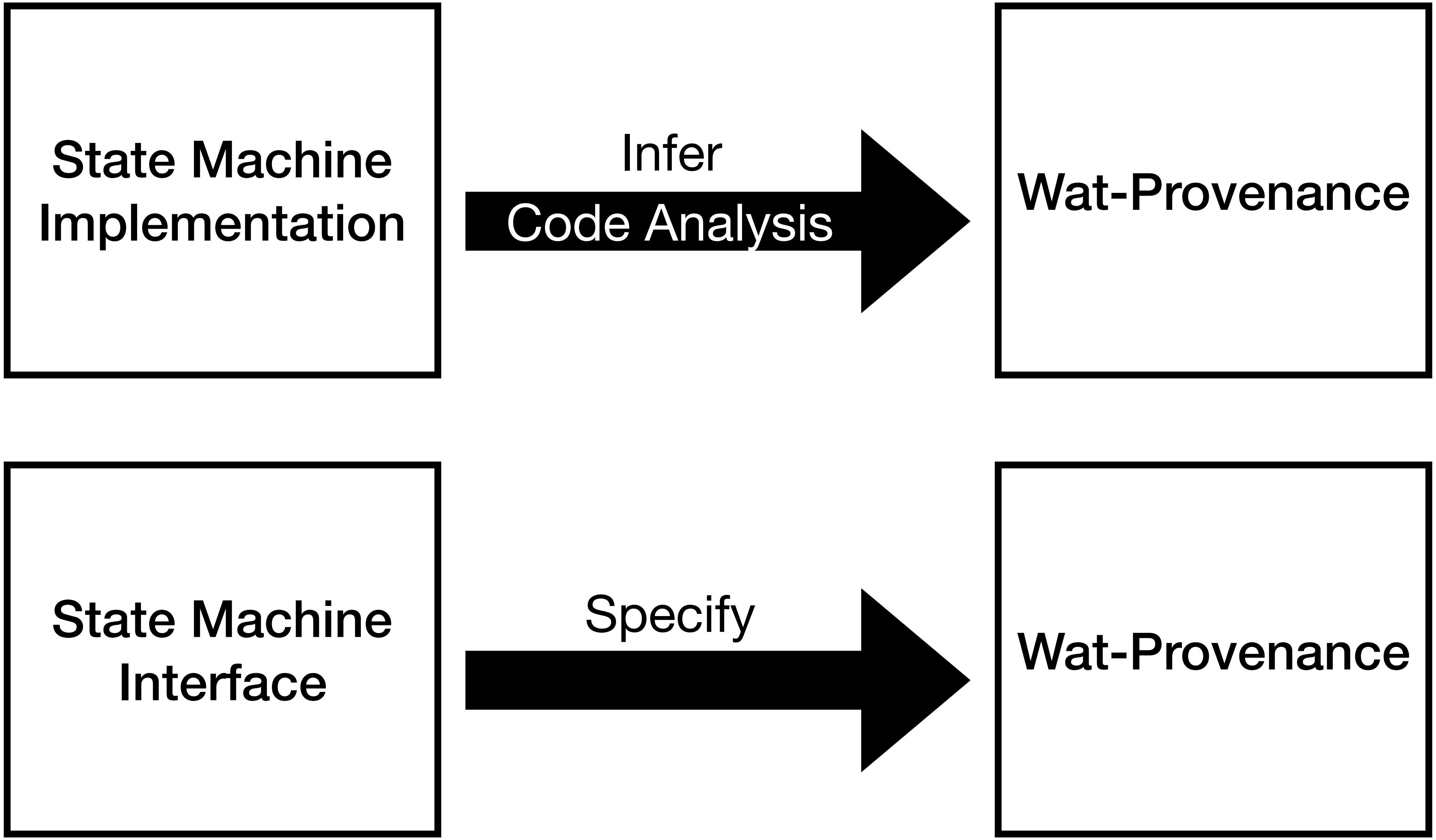
Infer

Code Analysis

Wat-Provenance

State Machine Interface

Computing Wat-Provenance



State Machine Implementation

Infer

Code Analysis

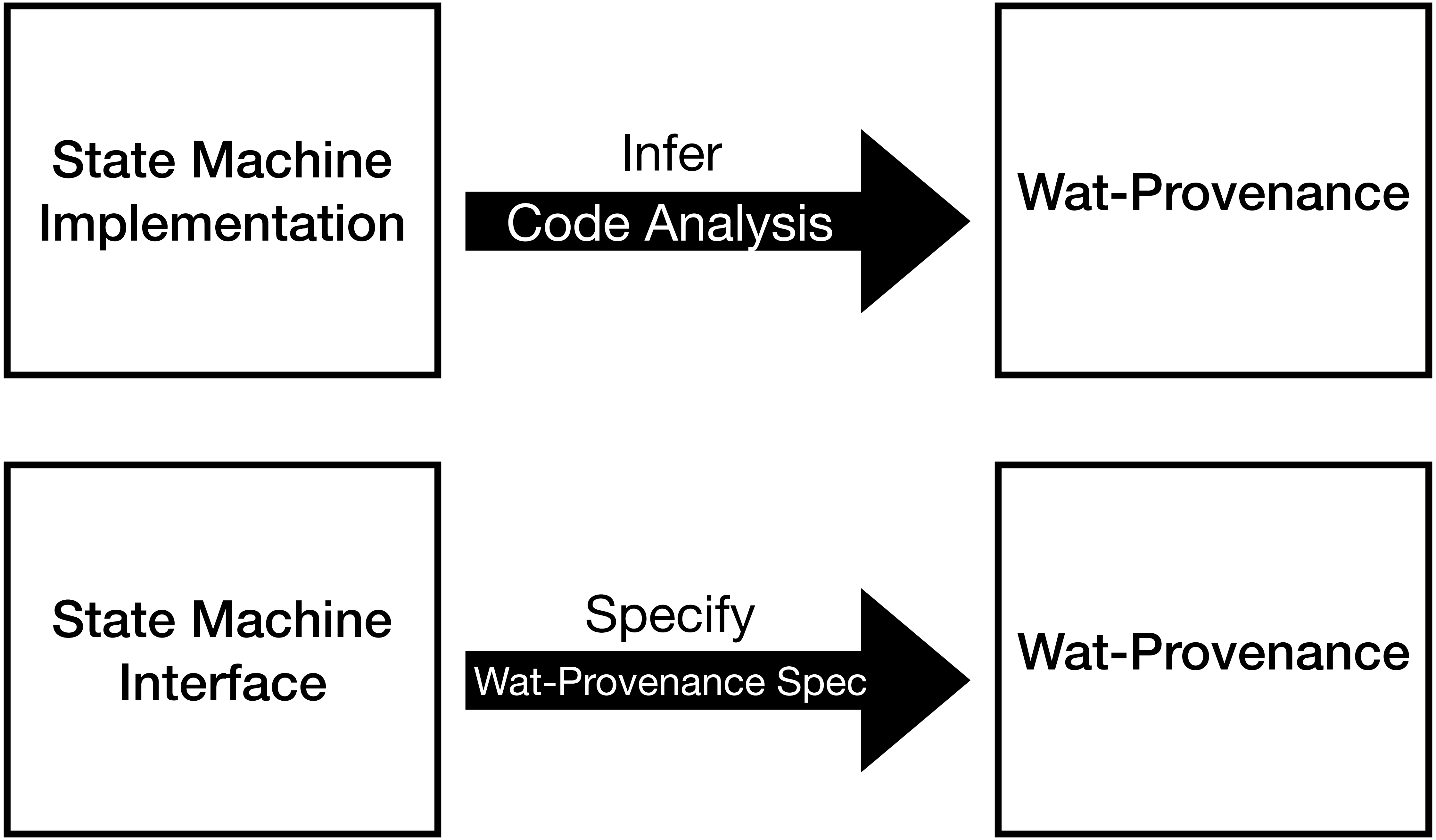
Wat-Provenance

State Machine Interface

Specify

Wat-Provenance

Computing Wat-Provenance



State Machine Implementation

Infer

Code Analysis

Wat-Provenance

State Machine Interface

Specify

Wat-Provenance Spec

Wat-Provenance

Example: Key-Value Store

Trace T

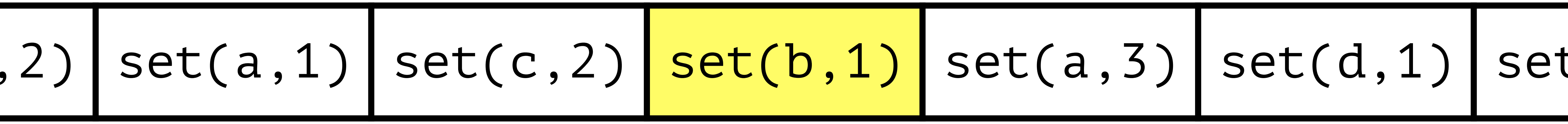
,1)	set(a,3)	set(e,1)	set(f,4)

Input i = get(b)

Output o = 1

Example: Key-Value Store

Trace T



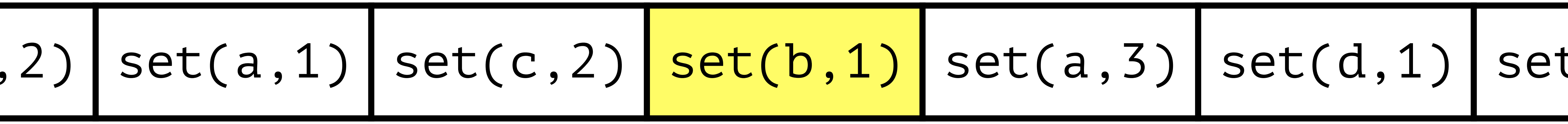
,2)	set(a,1)	set(c,2)	set(b,1)	set(a,3)	set(d,1)	set

Input i = get(b)

Output o = 1

Example: Key-Value Store

`Trace T`



`,2)` | `set(a,1)` | `set(c,2)` | `set(b,1)` | `set(a,3)` | `set(d,1)` | `set`

`Input i = get(b)`

`Output o = 1`

English wat-provenance specification:
The wat-provenance of a get of key *k* is
the most recent set to *k*.

Example: Key-Value Store

English wat-provenance specification:
The wat-provenance of a get of key *k* is
the most recent set to *k*.

Python wat-provenance specification:

```python
def get_prov(T: List[Request], i: GetRequest):
  for a in reversed(T):
    if (isinstance(a, SetRequest) and a.key == i.key):
      return {[a]}
  return {[]}
```

Wat-Provenance Specifications

Simple wat-provenance specifications are not uncommon:

- Key-Value Stores
- Object Stores
- Distributed File Systems
- Coordination Services
- Load Balancers
- Stateless Services

Come to my poster!

- ✓ **What is wat-provenance?**
- ✓ **How do you compute wat-provenance?**
- ✗ **How do you use wat-provenance?**
- ✗ **What are the limitations of wat-provenace?**

Thank you!