Engineering of Highly Concurrent Systems

Sung-Shik Jongmans[1,2]
ssj@ou.nl

[1]Open University of the Netherlands, the Netherlands
[2]Radboud University Nijmegen, the Netherlands

15 January 2016

Lots of Iranians have published papers on stuff related to what I will talk about:

Farhad Arbab, Nesa Asoudeh, Behnaz Changizi, Mehdi Dastani, Fatemeh Ghassemi, Mahmoud Reza Hashemi, Abbas Heydarnoori, Hossein Hojjat, Hamed Iravanchi, Mohammad Izadi, Mohammad Mahdi Jaghoori, Sarmen Keshishzadeh, Ramtin Khosravi, Farzad Mahdikhani, Farhad Mavaddat, Roshanak Zilouchian Moghaddam, MohammadReza Mousavi, Ali Movaghar, Sara NavidPour, Bahman Pourvatan, Niloofar Razavi, Nima Rouhy, Hamideh Sabouri, Shaghayegh Sahebi, Mahdi Sargolzaei, Marjan Sirjani, Samira Tasharofi, Mohsen Vakilian

Observations: [Fok]

- Multicore processors have become *ubiquitous*
- Parallel programming has become *essential*

Conceptually, a parallel programs consist of:

- Processes—Units of **computation** (*sequential*)
  - Known for decades
  - No new fundamental challenges

- Protocols—Rules of **communication** (*concurrent*)
  - Niche until recently
  - Not as well-understood

How to program protocols?

(Main topic of these lectures)

Running example: $\underbrace{\textbf{Producers}}_{\text{Alice, Bob}}$ / $\underbrace{\textbf{consumer}}_{\text{Carol}}$ protocol

Running example: $\underbrace{\textbf{Producers}}_{\text{Alice, Bob}}$ / $\underbrace{\textbf{consumer}}_{\text{Carol}}$ protocol

Properties:

- Asynchronous: Alice/Bob proceed after sending a message, possibly before Carol has received that message

Running example: $\underbrace{\textbf{Producers}}_{\text{Alice, Bob}}$ / $\underbrace{\textbf{consumer}}_{\text{Carol}}$ protocol

Properties:

- Asynchronous: Alice/Bob proceed after sending a message, possibly before Carol has received that message
- Reliable: No messages are lost or altered

Running example: $\underbrace{\textbf{Producers}}_{\text{Alice, Bob}}$ / $\underbrace{\textbf{consumer}}_{\text{Carol}}$ protocol

Properties:

- Asynchronous: Alice/Bob proceed after sending a message, possibly before Carol has received that message
- Reliable: No messages are lost or altered
- Unordered: Alice/Bob send messages in no order

Running example: $\underbrace{\textbf{Producers}}_{\text{Alice, Bob}}$ / $\underbrace{\textbf{consumer}}_{\text{Carol}}$ protocol

Properties:

- Asynchronous: Alice/Bob proceed after sending a message, possibly before Carol has received that message
- Reliable: No messages are lost or altered
- Unordered: Alice/Bob send messages in no order
- Transactional: After Alice/Bob has sent a message, Carol must receive that message before the next message is sent

```java
public class Buffer {
  public Object content;
  public Semaphore empty = new Semaphore(1);
  public Semaphore  full = new Semaphore(0);
}
```

Typical implementation

```
public class Producer extends Thread {
  private Buffer buffer;
  private Random rng;

  public Producer(Buffer buffer, long seed) {
    this.buffer = buffer;
    this.rng = new Random(seed);
  }

  @Override
  public void run() {
    while (true) {
      Object message = rng.nextInt(100);
      buffer.empty.acquire();
      buffer.content = message;
      buffer.full.release();
} } }
```

Typical implementation

```
public class Consumer extends Thread {
  private final Buffer buffer;

  public Consumer(Buffer buffer) {
    this.buffer = buffer;
  }

  @Override
  public void run() {
    while (true) {
      buffer.full.acquire();
      Object message = buffer.content;
      buffer.empty.release();
      System.out.println(message);
} } }
```

Typical implementation

```
public class Program {
  public static void main(String[] args) {
    Buffer buffer = new Buffer();

    new Producer(buffer, 0).start(); // Alice
    new Producer(buffer, 1).start(); // Bob
    new Consumer(buffer   ).start(); // Carol
} }
```

Typical implementation

Quiz :)

- **Rule 1:** Five questions
- **Rule 2:** I ask a question, then I count to five
- **Rule 3:** You raise your hand (high!) once you know the answer

- **Rule 1:** Five questions
- **Rule 2:** I ask a question, then I count to five
- **Rule 3:** You raise your hand (high!) once you know the answer

- **Rule 4a:** If *few* people raise their hand, *nobody* answers
- **Rule 4b:** If *many* people raise their hand, *somebody* answers

- **Question -2:** Do you understand the rules?

- **Question -2:** Do you understand the rules?

- **Question -1:** Are you going to participate?

```java
public class Program {
  public static void main(String[] args) {
    Buffer buffer = new Buffer();

    new Producer(buffer, 0).start();
    new Producer(buffer, 1).start();
    new Consumer(buffer   ).start();
} }
```

```java
public class Producer extends Thread {
  private Buffer buffer;
  private Random rng;

  public Producer(Buffer buffer, long seed) {
    this.buffer = buffer;
    this.rng = new Random(seed);
  }

  @Override
  public void run() {
    while (true) {
      Object message = rng.nextInt(100);
      buffer.empty.acquire();
      buffer.content = message;
      buffer.full.release();
} } }
```

```java
public class Buffer {
  public Object content;
  public Semaphore empty = new Semaphore(1);
  public Semaphore  full = new Semaphore(0);
}
```

```java
public class Consumer extends Thread {
  private Buffer buffer;

  public Consumer(Buffer buffer) {
    this.buffer = buffer;
  }

  @Override
  public void run() {
    while (true) {
      buffer.full.acquire();
      Object message = buffer.content;
      buffer.empty.release();
      System.out.println(message);
} } }
```

```java
public class Program {
  public static void main(String[] args) {
    Buffer buffer = new Buffer();

    new Producer(buffer, 0).start();
    new Producer(buffer, 1).start();
    new Consumer(buffer   ).start();
} }

public class Producer extends Thread {
  private Buffer buffer;
  private Random rng;

  public Producer(Buffer buffer, long seed) {
    this.buffer = buffer;
    this.rng = new Random(seed);
  }

  @Override
  public void run() {
    while (true) {
      Object message = rng.nextInt(100);
      buffer.empty.acquire();
      buffer.content = message;
      buffer.full.release();
} } }
```

```java
public class Buffer {
  public Object content;
  public Semaphore empty = new Semaphore(1);
  public Semaphore  full = new Semaphore(0);
}

public class Consumer extends Thread {
  private Buffer buffer;

  public Consumer(Buffer buffer) {
    this.buffer = buffer;
  }

  @Override
  public void run() {
    while (true) {
      buffer.full.acquire();
      Object message = buffer.content;
      buffer.empty.release();
      System.out.println(message);
} } }
```

Question 1: Where is the message produced?

```java
 1 public class Program {
 2   public static void main(String[] args) {
 3     Buffer buffer = new Buffer();
 4
 5     new Producer(buffer, 0).start();
 6     new Producer(buffer, 1).start();
 7     new Consumer(buffer   ).start();
 8 } }

 9 public class Producer extends Thread {
10   private Buffer buffer;
11   private Random rng;
12
13   public Producer(Buffer buffer, long seed) {
14     this.buffer = buffer;
15     this.rng = new Random(seed);
16   }
17
18   @Override
19   public void run() {
20     while (true) {
21       Object message = rng.nextInt(100);
22       buffer.empty.acquire();
23       buffer.content = message;
24       buffer.full.release();
25 } } }
```

```java
26 public class Buffer {
27   public Object content;
28   public Semaphore empty = new Semaphore(1);
29   public Semaphore  full = new Semaphore(0);
30 }

31 public class Consumer extends Thread {
32   private Buffer buffer;
33
34   public Consumer(Buffer buffer) {
35     this.buffer = buffer;
36   }
37
38   @Override
39   public void run() {
40     while (true) {
41       buffer.full.acquire();
42       Object message = buffer.content;
43       buffer.empty.release();
44       System.out.println(message);
45 } } }
```

Question 1: Where is the message produced? **Line 21**

```java
public class Program {
  public static void main(String[] args) {
    Buffer buffer = new Buffer();

    new Producer(buffer, 0).start();
    new Producer(buffer, 1).start();
    new Consumer(buffer   ).start();
} }
```

```java
public class Producer extends Thread {
  private Buffer buffer;
  private Random rng;

  public Producer(Buffer buffer, long seed) {
    this.buffer = buffer;
    this.rng = new Random(seed);
  }

  @Override
  public void run() {
    while (true) {
      Object message = rng.nextInt(100);
      buffer.empty.acquire();
      buffer.content = message;
      buffer.full.release();
} } }
```

```java
public class Buffer {
  public Object content;
  public Semaphore empty = new Semaphore(1);
  public Semaphore  full = new Semaphore(0);
}
```

```java
public class Consumer extends Thread {
  private Buffer buffer;

  public Consumer(Buffer buffer) {
    this.buffer = buffer;
  }

  @Override
  public void run() {
    while (true) {
      buffer.full.acquire();
      Object message = buffer.content;
      buffer.empty.release();
      System.out.println(message);
} } }
```

Question 2: Where is the message consumed?

```
 1 public class Program {
 2   public static void main(String[] args) {
 3     Buffer buffer = new Buffer();
 4
 5     new Producer(buffer, 0).start();
 6     new Producer(buffer, 1).start();
 7     new Consumer(buffer   ).start();
 8 } }

 9 public class Producer extends Thread {
10   private Buffer buffer;
11   private Random rng;
12
13   public Producer(Buffer buffer, long seed) {
14     this.buffer = buffer;
15     this.rng = new Random(seed);
16   }
17
18   @Override
19   public void run() {
20     while (true) {
21       Object message = rng.nextInt(100);
22       buffer.empty.acquire();
23       buffer.content = message;
24       buffer.full.release();
25 } } }
```

```
26 public class Buffer {
27   public Object content;
28   public Semaphore empty = new Semaphore(1);
29   public Semaphore  full = new Semaphore(0);
30 }

31 public class Consumer extends Thread {
32   private Buffer buffer;
33
34   public Consumer(Buffer buffer) {
35     this.buffer = buffer;
36   }
37
38   @Override
39   public void run() {
40     while (true) {
41       buffer.full.acquire();
42       Object message = buffer.content;
43       buffer.empty.release();
44       System.out.println(message);
45 } } }
```

Question 2: Where is the message consumed? **Line 44**

```
public class Program {
  public static void main(String[] args) {
    Buffer buffer = new Buffer();

    new Producer(buffer, 0).start();
    new Producer(buffer, 1).start();
    new Consumer(buffer   ).start();
} }
```

```
public class Producer extends Thread {
  private Buffer buffer;
  private Random rng;

  public Producer(Buffer buffer, long seed) {
    this.buffer = buffer;
    this.rng = new Random(seed);
  }

  @Override
  public void run() {
    while (true) {
      Object message = rng.nextInt(100);
      buffer.empty.acquire();
      buffer.content = message;
      buffer.full.release();
} } }
```

```
public class Buffer {
  public Object content;
  public Semaphore empty = new Semaphore(1);
  public Semaphore  full = new Semaphore(0);
}
```

```
public class Consumer extends Thread {
  private Buffer buffer;

  public Consumer(Buffer buffer) {
    this.buffer = buffer;
  }

  @Override
  public void run() {
    while (true) {
      buffer.full.acquire();
      Object message = buffer.content;
      buffer.empty.release();
      System.out.println(message);
} } }
```

Question 3: Where is a producer’s (non)termination?

```java
 1 public class Program {
 2   public static void main(String[] args) {
 3     Buffer buffer = new Buffer();
 4
 5     new Producer(buffer, 0).start();
 6     new Producer(buffer, 1).start();
 7     new Consumer(buffer   ).start();
 8 } }

 9 public class Producer extends Thread {
10   private Buffer buffer;
11   private Random rng;
12
13   public Producer(Buffer buffer, long seed) {
14     this.buffer = buffer;
15     this.rng = new Random(seed);
16   }
17
18   @Override
19   public void run() {
20     while (true) {
21       Object message = rng.nextInt(100);
22       buffer.empty.acquire();
23       buffer.content = message;
24       buffer.full.release();
25 } } }
```

```java
26 public class Buffer {
27   public Object content;
28   public Semaphore empty = new Semaphore(1);
29   public Semaphore  full = new Semaphore(0);
30 }

31 public class Consumer extends Thread {
32   private Buffer buffer;
33
34   public Consumer(Buffer buffer) {
35     this.buffer = buffer;
36   }
37
38   @Override
39   public void run() {
40     while (true) {
41       buffer.full.acquire();
42       Object message = buffer.content;
43       buffer.empty.release();
44       System.out.println(message);
45 } } }
```

Question 3: Where is a producer's (non)termination? **Line 20**

```java
public class Program {
  public static void main(String[] args) {
    Buffer buffer = new Buffer();

    new Producer(buffer, 0).start();
    new Producer(buffer, 1).start();
    new Consumer(buffer   ).start();
} }
```

```java
public class Producer extends Thread {
  private Buffer buffer;
  private Random rng;

  public Producer(Buffer buffer, long seed) {
    this.buffer = buffer;
    this.rng = new Random(seed);
  }

  @Override
  public void run() {
    while (true) {
      Object message = rng.nextInt(100);
      buffer.empty.acquire();
      buffer.content = message;
      buffer.full.release();
} } }
```

```java
public class Buffer {
  public Object content;
  public Semaphore empty = new Semaphore(1);
  public Semaphore  full = new Semaphore(0);
}
```

```java
public class Consumer extends Thread {
  private Buffer buffer;

  public Consumer(Buffer buffer) {
    this.buffer = buffer;
  }

  @Override
  public void run() {
    while (true) {
      buffer.full.acquire();
      Object message = buffer.content;
      buffer.empty.release();
      System.out.println(message);
} } }
```

Question 4: Where is the consumer?

```
 1 public class Program {
 2   public static void main(String[] args) {
 3     Buffer buffer = new Buffer();
 4
 5     new Producer(buffer, 0).start();
 6     new Producer(buffer, 1).start();
 7     new Consumer(buffer   ).start();
 8 } }

 9 public class Producer extends Thread {
10   private Buffer buffer;
11   private Random rng;
12
13   public Producer(Buffer buffer, long seed) {
14     this.buffer = buffer;
15     this.rng = new Random(seed);
16   }
17
18   @Override
19   public void run() {
20     while (true) {
21       Object message = rng.nextInt(100);
22       buffer.empty.acquire();
23       buffer.content = message;
24       buffer.full.release();
25 } } }
```

```
26 public class Buffer {
27   public Object content;
28   public Semaphore empty = new Semaphore(1);
29   public Semaphore  full = new Semaphore(0);
30 }

31 public class Consumer extends Thread {
32   private Buffer buffer;
33
34   public Consumer(Buffer buffer) {
35     this.buffer = buffer;
36   }
37
38   @Override
39   public void run() {
40     while (true) {
41       buffer.full.acquire();
42       Object message = buffer.content;
43       buffer.empty.release();
44       System.out.println(message);
45 } } }
```

Question 4: Where is the consumer? **Lines 31–45**

```java
public class Program {
  public static void main(String[] args) {
    Buffer buffer = new Buffer();

    new Producer(buffer, 0).start();
    new Producer(buffer, 1).start();
    new Consumer(buffer   ).start();
} }
```

```java
public class Producer extends Thread {
  private Buffer buffer;
  private Random rng;

  public Producer(Buffer buffer, long seed) {
    this.buffer = buffer;
    this.rng = new Random(seed);
  }

  @Override
  public void run() {
    while (true) {
      Object message = rng.nextInt(100);
      buffer.empty.acquire();
      buffer.content = message;
      buffer.full.release();
} } }
```

```java
public class Buffer {
  public Object content;
  public Semaphore empty = new Semaphore(1);
  public Semaphore  full = new Semaphore(0);
}
```

```java
public class Consumer extends Thread {
  private Buffer buffer;

  public Consumer(Buffer buffer) {
    this.buffer = buffer;
  }

  @Override
  public void run() {
    while (true) {
      buffer.full.acquire();
      Object message = buffer.content;
      buffer.empty.release();
      System.out.println(message);
} } }
```

Question 5: Where is the protocol?

```java
public class Program {
  public static void main(String[] args) {
    Buffer buffer = new Buffer();

    new Producer(buffer, 0).start();
    new Producer(buffer, 1).start();
    new Consumer(buffer   ).start();
} }

public class Producer extends Thread {
  private Buffer buffer;
  private Random rng;

  public Producer(Buffer buffer, long seed) {
    this.buffer = buffer;
    this.rng = new Random(seed);
  }

  @Override
  public void run() {
    while (true) {
      Object message = rng.nextInt(100);
      buffer.empty.acquire();
      buffer.content = message;
      buffer.full.release();
} } }
```

```java
public class Buffer {
  public Object content;
  public Semaphore empty = new Semaphore(1);
  public Semaphore  full = new Semaphore(0);
}

public class Consumer extends Thread {
  private Buffer buffer;

  public Consumer(Buffer buffer) {
    this.buffer = buffer;
  }

  @Override
  public void run() {
    while (true) {
      buffer.full.acquire();
      Object message = buffer.content;
      buffer.empty.release();
      System.out.println(message);
} } }
```

Question 5: Where is the protocol? **Ehrm...**

```java
public class Program {
  public static void main(String[] args) {
    Buffer buffer = new Buffer();

    new Producer(buffer, 0).start();
    new Producer(buffer, 1).start();
    new Consumer(buffer   ).start();
} }
```

```java
public class Producer extends Thread {
  private Buffer buffer;
  private Random rng;

  public Producer(Buffer buffer, long seed) {
    this.buffer = buffer;
    this.rng = new Random(seed);
  }

  @Override
  public void run() {
    while (true) {
      Object message = rng.nextInt(100);
      buffer.empty.acquire();
      buffer.content = message;
      buffer.full.release();
} } }
```

```java
public class Buffer {
  public Object content;
  public Semaphore empty = new Semaphore(1);
  public Semaphore  full = new Semaphore(0);
}
```

```java
public class Consumer extends Thread {
  private Buffer buffer;

  public Consumer(Buffer buffer) {
    this.buffer = buffer;
  }

  @Override
  public void run() {
    while (true) {
      buffer.full.acquire();
      Object message = buffer.content;
      buffer.empty.release();
      System.out.println(message);
} } }
```

Question 5a: Where is asynchrony?

```java
public class Program {
  public static void main(String[] args) {
    Buffer buffer = new Buffer();

    new Producer(buffer, 0).start();
    new Producer(buffer, 1).start();
    new Consumer(buffer   ).start();
} }
```

```java
public class Producer extends Thread {
  private Buffer buffer;
  private Random rng;

  public Producer(Buffer buffer, long seed) {
    this.buffer = buffer;
    this.rng = new Random(seed);
  }

  @Override
  public void run() {
    while (true) {
      Object message = rng.nextInt(100);
      buffer.empty.acquire();
      buffer.content = message;
      buffer.full.release();
} } }
```

```java
public class Buffer {
  public Object content;
  public Semaphore empty = new Semaphore(1);
  public Semaphore  full = new Semaphore(0);
}
```

```java
public class Consumer extends Thread {
  private Buffer buffer;

  public Consumer(Buffer buffer) {
    this.buffer = buffer;
  }

  @Override
  public void run() {
    while (true) {
      buffer.full.acquire();
      Object message = buffer.content;
      buffer.empty.release();
      System.out.println(message);
} } }
```

Question 5b: Where is reliability?

```java
public class Program {
  public static void main(String[] args) {
    Buffer buffer = new Buffer();

    new Producer(buffer, 0).start();
    new Producer(buffer, 1).start();
    new Consumer(buffer   ).start();
} }

public class Producer extends Thread {
  private Buffer buffer;
  private Random rng;

  public Producer(Buffer buffer, long seed) {
    this.buffer = buffer;
    this.rng = new Random(seed);
  }

  @Override
  public void run() {
    while (true) {
      Object message = rng.nextInt(100);
      buffer.empty.acquire();
      buffer.content = message;
      buffer.full.release();
} } }
```

```java
public class Buffer {
  public Object content;
  public Semaphore empty = new Semaphore(1);
  public Semaphore  full = new Semaphore(0);
}

public class Consumer extends Thread {
  private Buffer buffer;

  public Consumer(Buffer buffer) {
    this.buffer = buffer;
  }

  @Override
  public void run() {
    while (true) {
      buffer.full.acquire();
      Object message = buffer.content;
      buffer.empty.release();
      System.out.println(message);
} } }
```

Question 5c: Where is unorderedness?

```java
public class Program {
  public static void main(String[] args) {
    Buffer buffer = new Buffer();

    new Producer(buffer, 0).start();
    new Producer(buffer, 1).start();
    new Consumer(buffer   ).start();
} }
```

```java
public class Producer extends Thread {
  private Buffer buffer;
  private Random rng;

  public Producer(Buffer buffer, long seed) {
    this.buffer = buffer;
    this.rng = new Random(seed);
  }

  @Override
  public void run() {
    while (true) {
      Object message = rng.nextInt(100);
      buffer.empty.acquire();
      buffer.content = message;
      buffer.full.release();
} } }
```

```java
public class Buffer {
  public Object content;
  public Semaphore empty = new Semaphore(1);
  public Semaphore  full = new Semaphore(0);
}
```

```java
public class Consumer extends Thread {
  private Buffer buffer;

  public Consumer(Buffer buffer) {
    this.buffer = buffer;
  }

  @Override
  public void run() {
    while (true) {
      buffer.full.acquire();
      Object message = buffer.content;
      buffer.empty.release();
      System.out.println(message);
} } }
```

Question 5d: Where is transactionality?

```java
public class Program {
  public static void main(String[] args) {
    Buffer buffer = new Buffer();

    new Producer(buffer, 0).start();
    new Producer(buffer, 1).start();
    new Consumer(buffer   ).start();
} }
```

```java
public class Producer extends Thread {
  private Buffer buffer;
  private Random rng;

  public Producer(Buffer buffer, long seed) {
    this.buffer = buffer;
    this.rng = new Random(seed);
  }

  @Override
  public void run() {
    while (true) {
      Object message = rng.nextInt(100);
      buffer.empty.acquire();
      buffer.content = message;
      buffer.full.release();
} } }
```

```java
public class Buffer {
  public Object content;
  public Semaphore empty = new Semaphore(1);
  public Semaphore  full = new Semaphore(0);
}
```

```java
public class Consumer extends Thread {
  private Buffer buffer;

  public Consumer(Buffer buffer) {
    this.buffer = buffer;
  }

  @Override
  public void run() {
    while (true) {
      buffer.full.acquire();
      Object message = buffer.content;
      buffer.empty.release();
      System.out.println(message);
} } }
```

Question 5: Where is the protocol?

```
 1 public class Program {
 2   public static void main(String[] args) {
 3     Buffer buffer = new Buffer();
 4
 5     new Producer(buffer, 0).start();
 6     new Producer(buffer, 1).start();
 7     new Consumer(buffer   ).start();
 8 } }

 9 public class Producer extends Thread {
10   private Buffer buffer;
11   private Random rng;
12
13   public Producer(Buffer buffer, long seed) {
14     this.buffer = buffer;
15     this.rng = new Random(seed);
16   }
17
18   @Override
19   public void run() {
20     while (true) {
21       Object message = rng.nextInt(100);
22       buffer.empty.acquire();
23       buffer.content = message;
24       buffer.full.release();
25 } } }
```

```
26 public class Buffer {
27   public Object content;
28   public Semaphore empty = new Semaphore(1);
29   public Semaphore  full = new Semaphore(0);
30 }

31 public class Consumer extends Thread {
32   private Buffer buffer;
33
34   public Consumer(Buffer buffer) {
35     this.buffer = buffer;
36   }
37
38   @Override
39   public void run() {
40     while (true) {
41       buffer.full.acquire();
42       Object message = buffer.content;
43       buffer.empty.release();
44       System.out.println(message);
45 } } }
```

Lines [10, 13-14, 22–24], [26–30], [32, 34–35, 41–43] (?)

- **Observation:** The protocol is *not* a separate module
- Parnas' advantages of modularization:

> *"(1) managerial—development time should be shortened because separate groups would work on each module with little need for communication; (2) product flexibility—it should be possible to make drastic changes to one module without a need to change others; (3) comprehensibility—it should be possible to study the system one module at a time."*

- **Observation:** The protocol is *not* a separate module
- Parnas' advantages of modularization:

> *"(1) managerial—development time should be shortened because separate groups would work on each module with little need for communication; (2) product flexibility—it should be possible to make drastic changes to one module without a need to change others; (3) comprehensibility—it should be possible to study the system one module at a time."*

- Protocol modularization seems a **good idea**

```java
public class Program {
  public static void main(String[] args) {
    Protocol protocol = new Protocol();

    new Producer(protocol, 0).start();
    new Producer(protocol, 1).start();
    new Consumer(protocol   ).start();
} }

public class Protocol {
  private Buffer buffer = new Buffer();

  public void send(Object message) {
    buffer.empty.acquire();
    buffer.content = message;
    buffer.full.release();
  }

  public Object receive() {
    buffer.full.acquire();
    Object message = buffer.content;
    buffer.empty.release();
    return message;
} }

public class Buffer {
  public Object content;
  public Semaphore empty = new Semaphore(1);
  public Semaphore  full = new Semaphore(0);
}
```

```java
public class Producer extends Thread {
  private Protocol protocol;
  private Random rng;

  public Producer(Protocol protocol, long seed) {
    this.protocol = protocol;
    this.rng = new Random(seed);
  }

  @Override
  public void run() {
    while (true) {
      Object message = rng.nextInt(100);
      protocol.send(message);
} } }

public class Consumer extends Thread {
  private Protocol protocol;

  public Consumer(Protocol protocol) {
    this.protocol = protocol;
  }

  @Override
  public void run() {
    while (true) {
      Object message = protocol.receive();
      System.out.println(message);
} } }
```

Modularize the protocol in its own class

```java
1  public class Program {
2    public static void main(String[] args) {
3      Protocol protocol = new Protocol();
4
5      new Producer(protocol, 0).start();
6      new Producer(protocol, 1).start();
7      new Consumer(protocol   ).start();
8  } }

9  public class Protocol {
10   private Buffer buffer = new Buffer();
11
12   public void send(Object message) {
13     buffer.empty.acquire();
14     buffer.content = message;
15     buffer.full.release();
16   }
17
18   public Object receive() {
19     buffer.full.acquire();
20     Object message = buffer.content;
21     buffer.empty.release();
22     return message;
23 } }

24 public class Buffer {
25   public Object content;
26   public Semaphore empty = new Semaphore(1);
27   public Semaphore  full = new Semaphore(0);
28 }
```

```java
29 public class Producer extends Thread {
30   private Protocol protocol;
31   private Random rng;
32
33   public Producer(Protocol protocol, long seed) {
34     this.protocol = protocol;
35     this.rng = new Random(seed);
36   }
37
38   @Override
39   public void run() {
40     while (true) {
41       Object message = rng.nextInt(100);
42       protocol.send(message);
43 } } }

44 public class Consumer extends Thread {
45   private Protocol protocol;
46
47   public Consumer(Protocol protocol) {
48     this.protocol = protocol;
49   }
50
51   @Override
52   public void run() {
53     while (true) {
54       Object message = protocol.receive();
55       System.out.println(message);
56 } } }
```

Question 5: Where is the protocol? **Lines 9–28**

```java
public class Program {
  public static void main(String[] args) {
    Protocol protocol = new Protocol();

    new Producer(protocol, 0).start();
    new Producer(protocol, 1).start();
    new Consumer(protocol   ).start();
} }

public class Protocol {
  private Buffer buffer = new Buffer();

  public void send(Object message) {
    buffer.empty.acquire();
    buffer.content = message;
    buffer.full.release();
  }

  public Object receive() {
    buffer.full.acquire();
    Object message = buffer.content;
    buffer.empty.release();
    return message;
} }

public class Buffer {
  public Object content;
  public Semaphore empty = new Semaphore(1);
  public Semaphore  full = new Semaphore(0);
}
```

```java
public class Producer extends Thread {
  private Protocol protocol;
  private Random rng;

  public Producer(Protocol protocol, long seed) {
    this.protocol = protocol;
    this.rng = new Random(seed);
  }

  @Override
  public void run() {
    while (true) {
      Object message = rng.nextInt(100);
      protocol.send(message);
} } }

public class Consumer extends Thread {
  private Protocol protocol;

  public Consumer(Protocol protocol) {
    this.protocol = protocol;
  }

  @Override
  public void run() {
    while (true) {
      Object message = protocol.receive();
      System.out.println(message);
} } }
```

Claim: Parnas’ advantages of modularization apply

Managerial advantages

("Obviously"...)

Product flexibility advantages

Running example′: $\underbrace{\textbf{Producers}}_{\text{Alice, Bob}}$ / $\underbrace{\textbf{consumer}}_{\text{Carol}}$ protocol

Properties:

- Asynchronous: Alice/Bob proceed after sending a message, possibly before Carol has received that message
- Reliable: No messages are lost or altered
- Unordered: Alice/Bob send messages in no order
- ~~Transactional: After Alice/Bob has sent a message, Carol must receive that message before the next message is sent~~

```java
public class Program {
  public static void main(String[] args) {
    Protocol protocol = new Protocol();

    new Producer(protocol, 0).start();
    new Producer(protocol, 1).start();
    new Consumer(protocol   ).start();
} }

public class Protocol {
  private Buffer buffer = new Buffer();

  public void send(Object message) {
    buffer.empty.acquire();
    buffer.content.enq(message);
    buffer.full.release();
  }

  public Object receive() {
    buffer.full.acquire();
    Object message = buffer.content.deq();
    buffer.empty.release();
    return message;
} }

public class Buffer {
  public Queue<?> content = new LockFreeQueue();
  public Semaphore empty = new Semaphore(2);
  public Semaphore  full = new Semaphore(0);
}
```

```java
public class Producer extends Thread {
  private Protocol protocol;
  private Random rng;

  public Producer(Protocol protocol, long seed) {
    this.protocol = protocol;
    this.rng = new Random(seed);
  }

  @Override
  public void run() {
    while (true) {
      Object message = rng.nextInt(100);
      protocol.send(message);
} } }

public class Consumer extends Thread {
  private Protocol protocol;

  public Consumer(Protocol protocol) {
    this.protocol = protocol;
  }

  @Override
  public void run() {
    while (true) {
      Object message = protocol.receive();
      System.out.println(message);
} } }
```

Use a concurrent queue, such as `LockFreeQueue` [Fok]

```java
public class Program {
  public static void main(String[] args) {
    Protocol protocol = new Protocol();

    new Producer(protocol, 0).start();
    new Producer(protocol, 1).start();
    new Consumer(protocol   ).start();
} }

public class Protocol {
  private Buffer buffer = new Buffer();

  public void send(Object message) {
    buffer.empty.acquire();
    buffer.content.enq(message);
    buffer.full.release();
  }

  public Object receive() {
    buffer.full.acquire();
    Object message = buffer.content.deq();
    buffer.empty.release();
    return message;
} }

public class Buffer {
  public Queue<?> content = new LockFreeQueue();
  public Semaphore empty = new Semaphore(2);
  public Semaphore  full = new Semaphore(0);
}
```

```java
public class Producer extends Thread {
  private Protocol protocol;
  private Random rng;

  public Producer(Protocol protocol, long seed) {
    this.protocol = protocol;
    this.rng = new Random(seed);
  }

  @Override
  public void run() {
    while (true) {
      Object message = rng.nextInt(100);
      protocol.send(message);
} } }

public class Consumer extends Thread {
  private Protocol protocol;

  public Consumer(Protocol protocol) {
    this.protocol = protocol;
  }

  @Override
  public void run() {
    while (true) {
      Object message = protocol.receive();
      System.out.println(message);
} } }
```

Classes `Program`, `Producer`, and `Consumer` remain unaffected

Running example$''$: $\underbrace{\textbf{Producers}}_{\text{Alice, Bob}}$ / $\underbrace{\textbf{consumer}}_{\text{Carol}}$ protocol

Properties:

- Asynchronous: Alice/Bob proceed after sending a message, possibly before Carol has received that message
- Reliable: No messages are lost or altered
- Unordered: Alice/Bob send messages in no order
- ~~Transactional: After Alice/Bob has sent a message, Carol must receive that message before the next message is sent~~
- Considerate: Alice/Bob cannot send her/his next message before Carol has received her/his current message

```java
public class Protocol {
  private Map<Long,Buffer> map = new HashMap<>();

  public void send(Object message) {
    long id = Thread.currentThread.getId();
    if (!map.containsKey(id))
      map.put(id, new Buffer());

    Buffer buffer = map.get(id);
    buffer.empty.acquire();
    buffer.content = message;
    buffer.full.release();
  }

  public Object receive() {
    while (true) {
      for (Buffer buffer : map.values()) {
        if (buffer.full.tryAcquire()) {
          Object message = buffer.content;
          buffer.empty.release();
          return message;
} } } } }
```

```java
public class Buffer {
  public Object content;
  public Semaphore empty = new Semaphore(1);
  public Semaphore  full = new Semaphore(0);
}
```

Use multiple buffers and busy-waiting

```
public class Protocol {
  private Map<Long,Buffer> map = new HashMap<>();

  public void send(Object message) {
    long id = Thread.currentThread.getId();
    if (!map.containsKey(id))
      map.put(id, new Buffer());

    Buffer buffer = map.get(id);
    buffer.empty.acquire();
    buffer.content = message;
    buffer.full.release();
  }

  public Object receive() {
    while (true) {
      for (Buffer buffer : map.values()) {
        if (buffer.full.tryAcquire()) {
          Object message = buffer.content;
          buffer.empty.release();
          return message;
} } } } }
```

```
public class Buffer {
  public Object content;
  public Semaphore empty = new Semaphore(1);
  public Semaphore  full = new Semaphore(0);
}
```

Use multiple buffers ~~and busy-waiting~~?

```
public class Protocol {
  private Map<Long,Buffer> map = new HashMap<>();
  private Semaphore available = new Semaphore(0);

  public void send(Object message) {
    long id = Thread.currentThread.getId();
    if (!map.containsKey(id))
      map.put(id, new Buffer());

    Buffer buffer = map.get(id);
    buffer.empty.acquire();
    buffer.content = message;
    buffer.isFull = true;
    available.release();
  }

  public Object receive() {
    ~~while (true) {~~
      available.acquire();
      for (Buffer buffer : map.values()) {
        ~~if (buffer.full.tryAcquire()) {~~
        if (buffer.isFull) {
          Object message = buffer.content;
          buffer.isFull = false;
          buffer.empty.release();
          return message;
} } ~~}~~ } }
```

```
public class Buffer {
  public Object content;
  public Semaphore empty = new Semaphore(1);
  ~~public Semaphore  full = new Semaphore(0);~~
  public boolean isFull = false;
}
```

Use multiple buffers ~~and busy-waiting~~

Comprehensibility advantages

```
public class Protocol {
  private Buffer buffer = new Buffer();

  public void send(Object message) {
    buffer.empty.acquire();
    buffer.content = message;
    buffer.full.release();
  }

  public Object receive() {
    buffer.full.acquire();
    Object message = buffer.content;
    buffer.empty.release();
    return message;
} }
```

```
public class Buffer {
  public Object content;
  public Semaphore empty = new Semaphore(1);
  public Semaphore  full = new Semaphore(0);
}
```

Running example: Is the protocol correct?

```
public class Protocol {
  private Buffer buffer = new Buffer();

  public void send(Object message) {
    buffer.empty.acquire();
    buffer.content = message;
    buffer.full.release();
  }

  public Object receive() {
    buffer.full.acquire();
    Object message = buffer.content;
    buffer.empty.release();
    return message;
} }
```

```
public class Buffer {
  public Object content;
  public Semaphore empty = new Semaphore(1);
  public Semaphore  full = new Semaphore(0);
}
```

Running example: Is communication *really* reliable?

```java
public class Protocol {
  private Buffer buffer = new Buffer();

  public void send(Object message) {
    buffer.empty.acquire();
    buffer.content = message;
    buffer.full.release();
  }

  public Object receive() {
    buffer.full.acquire();
    Object message = buffer.content;
    buffer.empty.release();
    return message;
} }
```

```java
public class Buffer {
  public Object content;
  public Semaphore empty = new Semaphore(1);
  public Semaphore  full = new Semaphore(0);
}
```

Running example: Should `Buffer.content` be **volatile**? [Fok]

- Should `Buffer.content` be **volatile**? No
- From the Java API:

> *“Actions in a thread prior to calling a "release" method such as release() happen-before actions following a successful "acquire" method such as acquire() in another thread.”*

```java
public class Protocol {
  private Buffer buffer = new Buffer();

  public void send(Object message) {
    buffer.empty.acquire();
    buffer.content.enq(message);
    buffer.full.release();
  }

  public Object receive() {
    buffer.full.acquire();
    Object message = buffer.content.deq();
    buffer.empty.release();
    return message;
} }
```

```java
public class Buffer {
  public Queue<?> content = new LockFreeQueue();
  public Semaphore empty = new Semaphore(2);
  public Semaphore  full = new Semaphore(0);
}
```

Running example′: Is the protocol correct?

```
public class Protocol {
  private Buffer buffer = new Buffer();

  public void send(Object message) {
    buffer.empty.acquire();
    buffer.content.enq(message);
    buffer.full.release();
  }

  public Object receive() {
    buffer.full.acquire();
    Object message = buffer.content.deq();
    buffer.empty.release();
    return message;
} }
```

```
public class Buffer {
  public Queue<?> content = new LockFreeQueue();
  public Semaphore empty = new Semaphore(2);
  public Semaphore  full = new Semaphore(0);
}
```

Yes (because `LockFreeQueue` is correct [Fok])

```java
public class Protocol {
  private Map<Long,Buffer> map = new HashMap<>();

  public void send(Object message) {
    long id = Thread.currentThread.getId();
    if (!map.containsKey(id))
      map.put(id, new Buffer());

    Buffer buffer = map.get(id);
    buffer.empty.acquire();
    buffer.content = message;
    buffer.full.release();
  }

  public Object receive() {
    while (true) {
      for (Buffer buffer : map.values()) {
        if (buffer.full.tryAcquire()) {
          Object message = buffer.content;
          buffer.empty.release();
          return message;
} } } } }
```

```java
public class Buffer {
  public Object content;
  public Semaphore empty = new Semaphore(1);
  public Semaphore  full = new Semaphore(0);
}
```

Running example$''$ (busy-waiting): Is the protocol correct?

```java
public class Protocol {
  private Map<Long,Buffer> map = new HashMap<>();

  public void send(Object message) {
    long id = Thread.currentThread.getId();
    if (!map.containsKey(id))
      map.put(id, new Buffer());

    Buffer buffer = map.get(id);
    buffer.empty.acquire();
    buffer.content = message;
    buffer.full.release();
  }

  public Object receive() {
    while (true) {
      for (Buffer buffer : map.values()) {
        if (buffer.full.tryAcquire()) {
          Object message = buffer.content;
          buffer.empty.release();
          return message;
} } } } }
```

```java
public class Buffer {
  public Object content;
  public Semaphore empty = new Semaphore(1);
  public Semaphore  full = new Semaphore(0);
}
```

No (because `HashMap` is not thread-safe)

```java
public class Protocol {
  private Map<Long,Buffer> map = new HashMap<>();
  private Semaphore available = new Semaphore(0);

  public void send(Object message) {
    long id = Thread.currentThread.getId();
    if (!map.containsKey(id))
      map.put(id, new Buffer());

    Buffer buffer = map.get(id);
    buffer.empty.acquire();
    buffer.content = message;
    buffer.isFull = true;
    available.release();
  }

  public Object receive() {
    ~~while (true) {~~
      available.acquire();
      for (Buffer buffer : map.values()) {
        ~~if (buffer.full.tryAcquire()) {~~
        if (buffer.isFull) {
          Object message = buffer.content;
          buffer.isFull = false;
          buffer.empty.release();
          return message;
} } ~~}~~ } }
```

```java
public class Buffer {
  public Object content;
  public Semaphore empty = new Semaphore(1);
  ~~public Semaphore  full = new Semaphore(0);~~
  public boolean isFull = false;
}
```

Running example″ (~~busy-waiting~~): Is the protocol correct?

```
public class Protocol {
  private Map<Long,Buffer> map = new HashMap<>();
  private Semaphore available = new Semaphore(0);

  public void send(Object message) {
    long id = Thread.currentThread.getId();
    if (!map.containsKey(id))
      map.put(id, new Buffer());

    Buffer buffer = map.get(id);
    buffer.empty.acquire();
    buffer.content = message;
    buffer.isFull = true;
    available.release();
  }

  public Object receive() {
    ~~while (true) {~~
      available.acquire();
      for (Buffer buffer : map.values()) {
        ~~if (buffer.full.tryAcquire()) {~~
        if (buffer.isFull) {
          Object message = buffer.content;
          buffer.isFull = false;
          buffer.empty.release();
          return message;
} } ~~}~~ } }
```

```
public class Buffer {
  public Object content;
  public Semaphore empty = new Semaphore(1);
  ~~public Semaphore  full = new Semaphore(0);~~
  public boolean isFull = false;
}
```

No (because `Buffer.content` is not **volatile**)

- **Observation:** Despite modularization, reasoning about protocols is still difficult [Fok, Sif]
- Accounting for all, seemingly *nondeterministic*, thread schedulings is too difficult

- **Observation:** Despite modularization, reasoning about protocols is still difficult [Fok, Sif]
- Accounting for all, seemingly *nondeterministic*, thread schedulings is too difficult

- Concurrency primitives (semaphores, monitors, etc.) are not the right **level of abstraction** for “average programmers” to effectively write correct *and* efficient protocol code
  - Programmers need to concern themselves with too many protocol-irrelevant, low-level details
  - (Assembly language vs. C, C++, Java, etc.)

Alternatives:

- Software transactional memory [Fok]
- Algorithmic skeletons / parallellism patterns
- *Domain-specific languages* (DSL) for protocols
- ...

Alternatives:

- Software transactional memory [Fok]
- Algorithmic skeletons / parallellism patterns
- *Domain-specific languages* (DSL) for protocols
- ...

Programming model

For now, forget everything you know about sempahores, data races, shared memory, mutual exclusion, ... [Fok]

For now, forget everything you know about sempahores, data races, shared memory, mutual exclusion, … [Fok]

Let there be only *(sequential) computation* and *ports* [Sif]

- Every process **owns** a set of ports

- Every process **owns** a set of ports

- Ports mark the **interface** between processes
- **All** inter-process communication occurs through ports
- (Conceptually, there is no shared memory!)

Running examples:

Alice
A
Bob
B
Carol
C

- Processes perform **blocking** operations on ports

```
public interface OutputPort {
  public void put(Object datum);
}

public interface InputPort {
  public Object get();
}
```

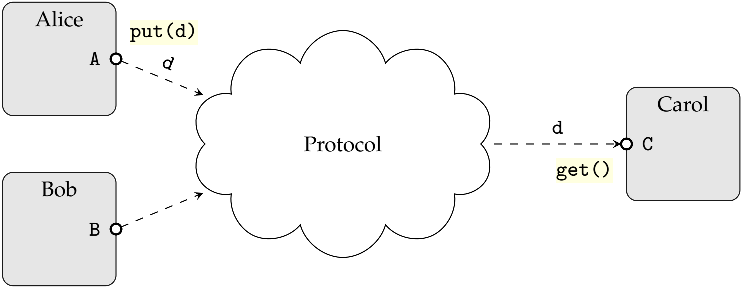
- Processes perform **blocking** operations on ports

```
public interface OutputPort {
  public void put(Object datum);
}

public interface InputPort {
  public Object get();
}
```

- Processes are **oblivious** to data-flows between ports
  - When `put(d)` returns, they know **not** whereto `d` goes
  - When `d=get()` returns, they know **not** wherefrom `d` comes
- *Only* protocols state how data flow

Running examples:



```
public class Processes {

  public static void Producer(OutputPort port, long seed) {
    Random rng = new Random(seed);
    while (true) {
      Object message = rng.nextInt(100);
      port.put(message);
  } }

  public static void Consumer(InputPort port) {
    while (true) {
      Object message = port.get();
      System.out.println(message);
} } }
```

Running examples: Port-based implementation

```
public class Program {
  public static void main(String[] args) {
    final OutputPort A = Port.newOutputPort();
    final OutputPort B = Port.newOutputPort();
    final InputPort  C = Port.newInputPort();

    (new Protocol(A,B,C)).start();

    Thread alice = new Thread() {
      public void run() { Processes.Producer(A, 0) } }
    Thread   bob = new Thread() {
      public void run() { Processes.Producer(B, 1) } }
    Thread carol = new Thread() {
      public void run() { Processes.Consumer(C) } }

    alice.start();
      bob.start();
    carol.start();
} }
```

Running examples: Port-based implementation

```
public class Program {
  public static void main(String[] args) {
    final OutputPort A = Port.newOutputPort();
    final OutputPort B = Port.newOutputPort();
    final InputPort  C = Port.newInputPort();

    (new Protocol(A,B,C)).start();

    Thread alice = new Thread() {
      public void run() { Processes.Producer(A, 0) } }
    Thread   bob = new Thread() {
      public void run() { Processes.Producer(B, 1) } }
    Thread carol = new Thread() {
      public void run() { Processes.Consumer(C) } }

    alice.start();
      bob.start();
    carol.start();
} }
```

`Protocol` is **specified** in, and **generated** from, a protocol DSL

```
public class Program {
  public static void main(String[] args) {
    final OutputPort A = Port.newOutputPort();
    final OutputPort B = Port.newOutputPort();
    final InputPort  C = Port.newInputPort();

    (new Protocol(A,B,C)).start();

    Thread alice = new Thread() {
      public void run() { Processes.Producer(A, 0) } }
    Thread   bob = new Thread() {
      public void run() { Processes.Producer(B, 1) } }
    Thread carol = new Thread() {
      public void run() { Processes.Consumer(C) } }

    alice.start();
      bob.start();
    carol.start();
} }
```

(And so is Program)

Burning questions:

- What is the specification like?
  - What is the *syntax* of a protocol DSL?
  - What is the *semantics* of a protocol DSL?
  - What is the *expressiveness* of a protocol DSL?

- What is the generation like?
  - How to generate *lower-level* protocol code (e.g., Java) from *higher-level* protocol specs?
  - How to **efficiently** generate code from specs?
  - How to generate **efficient** code from specs?

Summary:

1. Program processes in a *general-purpose language* (GPL)
2. Program protocols in a *domain-specific language* (DSL)
3. Have a DSL compiler:
    1. Generate GPL code for DSL code
    2. Merge all GPL code into an integrated program
4. Have a GPL compiler generate an executable for the integrated program

What is the specification like?

Approach: First $\underbrace{\text{semantics}}_{\substack{\text{What are suitable}\\ \text{models of protocols?}}}$, then $\overbrace{\text{syntax}}^{\substack{\text{What are suitable}\\ \text{programming constructs}\\ \text{to denote such models?}}}$

- **Observation:** During a run, `put`/`get` actions complete

t	Alice	Bob	Carol	
1	`A.put(60)`			`A.put(60)`
2			`60=C.get()`	`60=C.get()`
3		`B.put(85)`		`B.put(85)`
4			`85=C.get()`	`85=C.get()`
5		`B.put(88)`		`B.put(88)`
6			`88=C.get()`	`88=C.get()`
7	`A.put(48)`			`A.put(48)`
8			`48=C.get()`	`48=C.get()`
9		`B.put(47)`		`B.put(47)`
10			`47=C.get()`	`47=C.get()`
...	...	...	...	...

- **Observation:** During a run, `put`/`get` actions complete

t	Alice	Bob	Carol	**interaction**
1	`A.put(60)`			`A.put(60)`
2			`60=C.get()`	`60=C.get()`
3		`B.put(85)`		`B.put(85)`
4			`85=C.get()`	`85=C.get()`
5		`B.put(88)`		`B.put(88)`
6			`88=C.get()`	`88=C.get()`
7	`A.put(48)`			`A.put(48)`
8			`48=C.get()`	`48=C.get()`
9		`B.put(47)`		`B.put(47)`
10			`47=C.get()`	`47=C.get()`
...	...	...	...	...

- **Terminology:**
  - A sequence of completions is an **interaction**
  - A set of *admissible* interactions is a **protocol**

Henceforth:

- $\mathbb{N}$ denotes the set of natural numbers
- $\mathbb{P}$ denotes the set of ports
- $\mathbb{D}$ denotes the set of data

Attempt 1: An interaction is a function $u : \mathbb{N} \to \mathbb{P}$ (or, equivalently, a stream $u \in \mathbb{P}^\omega$ [Rut])

Running example:

- $\mathtt{A}, \mathtt{C}, \mathtt{B}, \mathtt{C}, \mathtt{B}, \mathtt{C}, \mathtt{A}, \mathtt{C}, \mathtt{B}, \mathtt{C}, \ldots$
- $\mathtt{A}, \mathtt{C}, \mathtt{A}, \mathtt{C}, \mathtt{A}, \mathtt{C}, \mathtt{A}, \mathtt{C}, \mathtt{A}, \mathtt{C}, \ldots$
- $\mathtt{B}, \mathtt{C}, \mathtt{A}, \mathtt{C}, \mathtt{B}, \mathtt{C}, \mathtt{A}, \mathtt{C}, \mathtt{B}, \mathtt{C}, \ldots$
- $\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{C}, \mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{C}, \mathtt{A}, \mathtt{B}, \ldots$

Attempt 1: An interaction is a function $u : \mathbb{N} \to \mathbb{P}$
(or, equivalently, a stream $u \in \mathbb{P}^\omega$ [Rut])

Running example:

- `A`, `C`, `B`, `C`, `B`, `C`, `A`, `C`, `B`, `C`, . . .
- `A`, `C`, `A`, `C`, `A`, `C`, `A`, `C`, `A`, `C`, . . .
- `B`, `C`, `A`, `C`, `B`, `C`, `A`, `C`, `B`, `C`, . . .
- ~~A, B, C, C, A, B, C, C, A, B, . . .~~ [nontransactional]

- **Problem:** Cannot express synchronization
- **Solution:** Sets instead of elements

Attempt 2: An interaction is a function $u : \mathbb{N} \to 2^{\mathbb{P}}$
(or, equivalently, a stream $u \in (2^{\mathbb{P}})^{\omega}$ [Rut])

Running example:

- $\{\mathtt{A}\}, \{\mathtt{C}\}, \{\mathtt{B}\}, \{\mathtt{C}\}, \{\mathtt{B}\}, \{\mathtt{C}\}, \{\mathtt{A}\}, \{\mathtt{C}\}, \{\mathtt{B}\}, \{\mathtt{C}\}, \ldots$
- $\{\mathtt{A}\}, \{\mathtt{C}\}, \{\mathtt{A}\}, \{\mathtt{C}\}, \{\mathtt{A}\}, \{\mathtt{C}\}, \{\mathtt{A}\}, \{\mathtt{C}\}, \{\mathtt{A}\}, \{\mathtt{C}\}, \ldots$
- $\{\mathtt{B}\}, \{\mathtt{C}\}, \{\mathtt{A}\}, \{\mathtt{C}\}, \{\mathtt{B}\}, \{\mathtt{C}\}, \{\mathtt{A}\}, \{\mathtt{C}\}, \{\mathtt{B}\}, \{\mathtt{C}\}, \ldots$
- $\{\mathtt{A}, \mathtt{C}\}, \{\mathtt{B}, \mathtt{C}\}, \{\mathtt{A}, \mathtt{C}\}, \{\mathtt{B}, \mathtt{C}\}, \{\mathtt{A}, \mathtt{C}\}, \{\mathtt{B}, \mathtt{C}\}, \ldots$

Attempt 2: An interaction is a function $u : \mathbb{N} \to 2^{\mathbb{P}}$
(or, equivalently, a stream $u \in (2^{\mathbb{P}})^{\omega}$ [Rut])

Running example:

- $\{\mathtt{A}\}, \{\mathtt{C}\}, \{\mathtt{B}\}, \{\mathtt{C}\}, \{\mathtt{B}\}, \{\mathtt{C}\}, \{\mathtt{A}\}, \{\mathtt{C}\}, \{\mathtt{B}\}, \{\mathtt{C}\}, \ldots$
- $\{\mathtt{A}\}, \{\mathtt{C}\}, \{\mathtt{A}\}, \{\mathtt{C}\}, \{\mathtt{A}\}, \{\mathtt{C}\}, \{\mathtt{A}\}, \{\mathtt{C}\}, \{\mathtt{A}\}, \{\mathtt{C}\}, \ldots$
- $\{\mathtt{B}\}, \{\mathtt{C}\}, \{\mathtt{A}\}, \{\mathtt{C}\}, \{\mathtt{B}\}, \{\mathtt{C}\}, \{\mathtt{A}\}, \{\mathtt{C}\}, \{\mathtt{B}\}, \{\mathtt{C}\}, \ldots$
- ~~$\{\mathtt{A}, \mathtt{C}\}, \{\mathtt{B}, \mathtt{C}\}, \{\mathtt{A}, \mathtt{C}\}, \{\mathtt{B}, \mathtt{C}\}, \{\mathtt{A}, \mathtt{C}\}, \{\mathtt{B}, \mathtt{C}\}, \ldots$~~ [synchronous]

- **Problem:** Cannot express data-sensitivity
- **Solution:** Functions instead of sets

Attempt 3: An interaction is a function $u : \mathbb{N} \to \mathbb{P} \rightharpoonup \mathbb{D}$ (or, equivalently, a stream $u \in (\mathbb{P} \rightharpoonup \mathbb{D})^\omega$ [Rut])

Running example:

- $\{\mathtt{A} \mapsto \mathtt{60}\}, \{\mathtt{C} \mapsto \mathtt{60}\}, \{\mathtt{B} \mapsto \mathtt{85}\}, \{\mathtt{C} \mapsto \mathtt{85}\}, \{\mathtt{B} \mapsto \mathtt{88}\}, \ldots$
- $\{\mathtt{A} \mapsto \mathtt{60}\}, \{\mathtt{C} \mapsto \mathtt{60}\}, \{\mathtt{A} \mapsto \mathtt{48}\}, \{\mathtt{C} \mapsto \mathtt{48}\}, \{\mathtt{A} \mapsto \mathtt{29}\}, \ldots$
- $\{\mathtt{B} \mapsto \mathtt{85}\}, \{\mathtt{C} \mapsto \mathtt{85}\}, \{\mathtt{A} \mapsto \mathtt{60}\}, \{\mathtt{C} \mapsto \mathtt{60}\}, \{\mathtt{B} \mapsto \mathtt{88}\}, \ldots$
- $\{\mathtt{A} \mapsto \mathtt{60}, \mathtt{C} \mapsto \mathtt{60}\}, \{\mathtt{B} \mapsto \mathtt{85}, \mathtt{C} \mapsto \mathtt{85}\}, \ldots$
- $\{\mathtt{A} \mapsto \mathtt{60}\}, \{\mathtt{C} \mapsto \mathtt{nil}\}, \{\mathtt{B} \mapsto \mathtt{85}\}, \{\mathtt{C} \mapsto \mathtt{85}\}, \ldots$
- $\{\mathtt{A} \mapsto \mathtt{60}\}, \{\mathtt{B} \mapsto \mathtt{85}\}, \{\mathtt{C} \mapsto \mathtt{60}\}, \ldots$

Attempt 3: An interaction is a function $u : \mathbb{N} \to \mathbb{P} \rightharpoonup \mathbb{D}$
(or, equivalently, a stream $u \in (\mathbb{P} \rightharpoonup \mathbb{D})^{\omega}$ [Rut])

Running example:

- $\{\texttt{A} \mapsto \texttt{60}\}, \{\texttt{C} \mapsto \texttt{60}\}, \{\texttt{B} \mapsto \texttt{85}\}, \{\texttt{C} \mapsto \texttt{85}\}, \{\texttt{B} \mapsto \texttt{88}\}, \ldots$
- $\{\texttt{A} \mapsto \texttt{60}\}, \{\texttt{C} \mapsto \texttt{60}\}, \{\texttt{A} \mapsto \texttt{48}\}, \{\texttt{C} \mapsto \texttt{48}\}, \{\texttt{A} \mapsto \texttt{29}\}, \ldots$
- $\{\texttt{B} \mapsto \texttt{85}\}, \{\texttt{C} \mapsto \texttt{85}\}, \{\texttt{A} \mapsto \texttt{60}\}, \{\texttt{C} \mapsto \texttt{60}\}, \{\texttt{B} \mapsto \texttt{88}\}, \ldots$
- ~~$\{\texttt{A} \mapsto \texttt{60}, \texttt{C} \mapsto \texttt{60}\}, \{\texttt{B} \mapsto \texttt{85}, \texttt{C} \mapsto \texttt{85}\}, \ldots$~~ [synchronous]
- $\{\texttt{A} \mapsto \texttt{60}\}, \{\texttt{C} \mapsto \texttt{nil}\}, \{\texttt{B} \mapsto \texttt{85}\}, \{\texttt{C} \mapsto \texttt{85}\}, \ldots$
- $\{\texttt{A} \mapsto \texttt{60}\}, \{\texttt{B} \mapsto \texttt{85}\}, \{\texttt{C} \mapsto \texttt{60}\}, \ldots$

Attempt 3: An interaction is a function $u : \mathbb{N} \to \mathbb{P} \rightharpoonup \mathbb{D}$
(or, equivalently, a stream $u \in (\mathbb{P} \rightharpoonup \mathbb{D})^\omega$ [Rut])

Running example:

- $\{\mathtt{A} \mapsto \mathtt{60}\}, \{\mathtt{C} \mapsto \mathtt{60}\}, \{\mathtt{B} \mapsto \mathtt{85}\}, \{\mathtt{C} \mapsto \mathtt{85}\}, \{\mathtt{B} \mapsto \mathtt{88}\}, \ldots$
- $\{\mathtt{A} \mapsto \mathtt{60}\}, \{\mathtt{C} \mapsto \mathtt{60}\}, \{\mathtt{A} \mapsto \mathtt{48}\}, \{\mathtt{C} \mapsto \mathtt{48}\}, \{\mathtt{A} \mapsto \mathtt{29}\}, \ldots$
- $\{\mathtt{B} \mapsto \mathtt{85}\}, \{\mathtt{C} \mapsto \mathtt{85}\}, \{\mathtt{A} \mapsto \mathtt{60}\}, \{\mathtt{C} \mapsto \mathtt{60}\}, \{\mathtt{B} \mapsto \mathtt{88}\}, \ldots$
- ~~$\{\mathtt{A} \mapsto \mathtt{60}, \mathtt{C} \mapsto \mathtt{60}\}, \{\mathtt{B} \mapsto \mathtt{85}, \mathtt{C} \mapsto \mathtt{85}\}, \ldots$~~ [synchronous]
- ~~$\{\mathtt{A} \mapsto \mathtt{60}\}, \{\mathtt{C} \mapsto \mathtt{nil}\}, \{\mathtt{B} \mapsto \mathtt{85}\}, \{\mathtt{C} \mapsto \mathtt{85}\}, \ldots$~~ [unreliable]
- $\{\mathtt{A} \mapsto \mathtt{60}\}, \{\mathtt{B} \mapsto \mathtt{85}\}, \{\mathtt{C} \mapsto \mathtt{60}\}, \ldots$

Attempt 3: An interaction is a function $u : \mathbb{N} \to \mathbb{P} \rightharpoonup \mathbb{D}$
(or, equivalently, a stream $u \in (\mathbb{P} \rightharpoonup \mathbb{D})^{\omega}$ [Rut])

Running example:

- $\{\mathtt{A} \mapsto \mathtt{60}\}, \{\mathtt{C} \mapsto \mathtt{60}\}, \{\mathtt{B} \mapsto \mathtt{85}\}, \{\mathtt{C} \mapsto \mathtt{85}\}, \{\mathtt{B} \mapsto \mathtt{88}\}, \ldots$
- $\{\mathtt{A} \mapsto \mathtt{60}\}, \{\mathtt{C} \mapsto \mathtt{60}\}, \{\mathtt{A} \mapsto \mathtt{48}\}, \{\mathtt{C} \mapsto \mathtt{48}\}, \{\mathtt{A} \mapsto \mathtt{29}\}, \ldots$
- $\{\mathtt{B} \mapsto \mathtt{85}\}, \{\mathtt{C} \mapsto \mathtt{85}\}, \{\mathtt{A} \mapsto \mathtt{60}\}, \{\mathtt{C} \mapsto \mathtt{60}\}, \{\mathtt{B} \mapsto \mathtt{88}\}, \ldots$
- ~~$\{\mathtt{A} \mapsto \mathtt{60}, \mathtt{C} \mapsto \mathtt{60}\}, \{\mathtt{B} \mapsto \mathtt{85}, \mathtt{C} \mapsto \mathtt{85}\}, \ldots$~~ [synchronous]
- ~~$\{\mathtt{A} \mapsto \mathtt{60}\}, \{\mathtt{C} \mapsto \mathtt{nil}\}, \{\mathtt{B} \mapsto \mathtt{85}\}, \{\mathtt{C} \mapsto \mathtt{85}\}, \ldots$~~ [unreliable]
- ~~$\{\mathtt{A} \mapsto \mathtt{60}\}, \{\mathtt{B} \mapsto \mathtt{85}\}, \{\mathtt{C} \mapsto \mathtt{60}\}, \ldots$~~ [nontransactional]

Finally: A protocol is a set $L \subseteq \mathbb{N} \to (\mathbb{P} \rightharpoonup \mathbb{D})$
(or, equivalently, a predicate $L \subseteq (\mathbb{P} \rightharpoonup \mathbb{D})^\omega$)

Running example:

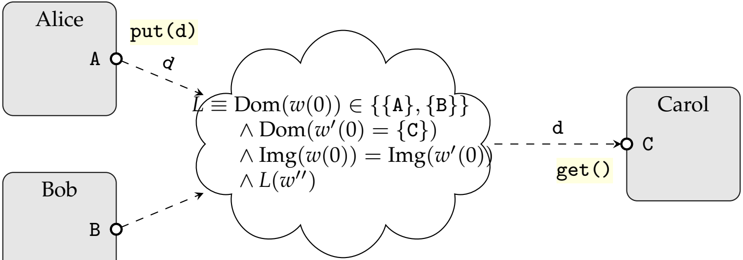
$$L = \left\{ w \,\middle|\, \begin{array}{l} w : \mathbb{N} \to (\mathbb{P} \rightharpoonup \mathbb{D}) \\ \textbf{and } \left[\mathrm{Dom}(w(i)) \in \{\{\mathtt{A}\}, \{\mathtt{B}\}\} \textbf{ for all } i \in \mathbb{N}_{\text{even}}\right] \\ \textbf{and } \left[\mathrm{Dom}(w(i)) = \{\mathtt{C}\} \textbf{ for all } i \in \mathbb{N}_{\text{odd}}\right] \\ \textbf{and } \left[\mathrm{Img}(w(i)) = \mathrm{Img}(w(i+1)) \textbf{ for all } i \in \mathbb{N}_{\text{even}}\right] \end{array} \right\}$$

(Or, equivalently:

$$\begin{aligned} L \equiv\ & \mathrm{Dom}(w(0)) \in \{\{\mathtt{A}\}, \{\mathtt{B}\}\} \\ & \wedge \mathrm{Dom}(w'(0) = \{\mathtt{C}\}) \\ & \wedge \mathrm{Img}(w(0)) = \mathrm{Img}(w'(0)) \\ & \wedge L(w'') \end{aligned}$$

[Rut])

Running example:



$$
\begin{aligned}
L \equiv\ & \mathrm{Dom}(w(0)) \in \{\{\mathtt{A}\}, \{\mathtt{B}\}\} \\
& \wedge \mathrm{Dom}(w'(0) = \{\mathtt{C}\}) \\
& \wedge \mathrm{Img}(w(0)) = \mathrm{Img}(w'(0)) \\
& \wedge L(w'')
\end{aligned}
$$

But...

Observations:

- *Directly* using set-based (or, equivalently, predicate-based) protocol specs is inconvenient for:
  - Automated composition
  - Automated code generation
  - Automated reasoning

Observations:

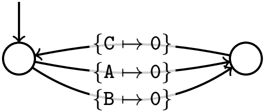
- *Directly* using set-based (or, equivalently, predicate-based) protocol specs is inconvenient for:
  - Automated composition
  - Automated code generation
  - Automated reasoning

- Interactions are **words**
- Protocols are **languages**
- Represent languages as automata

Attempt 1: A protocol is a tuple $(Q, P, \longrightarrow, q_0)$, where $\longrightarrow \subseteq Q \times (P \rightharpoonup \mathbb{D}) \times Q$

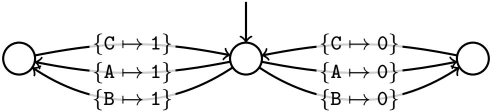
Attempt 1: A protocol is a tuple $(Q, P, \longrightarrow, q_0)$, where $\longrightarrow \subseteq Q \times (P \rightharpoonup \mathbb{D}) \times Q$

Running example, $\mathbb{D} = \{\mathtt{0}\}$**:**



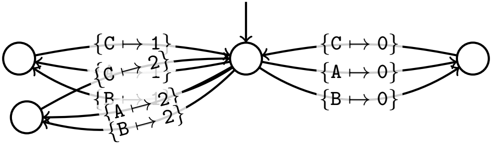
Attempt 1: A protocol is a tuple $(Q, P, \longrightarrow, q_0)$, where $\longrightarrow \subseteq Q \times (P \rightharpoonup \mathbb{D}) \times Q$

Running example, $\mathbb{D} = \{\texttt{0}, \texttt{1}\}$**:**



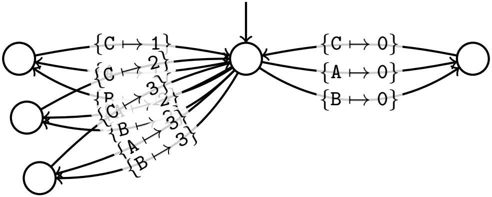
Attempt 1: A protocol is a tuple $(Q, P, \longrightarrow, q_0)$, where $\longrightarrow \subseteq Q \times (P \rightharpoonup \mathbb{D}) \times Q$

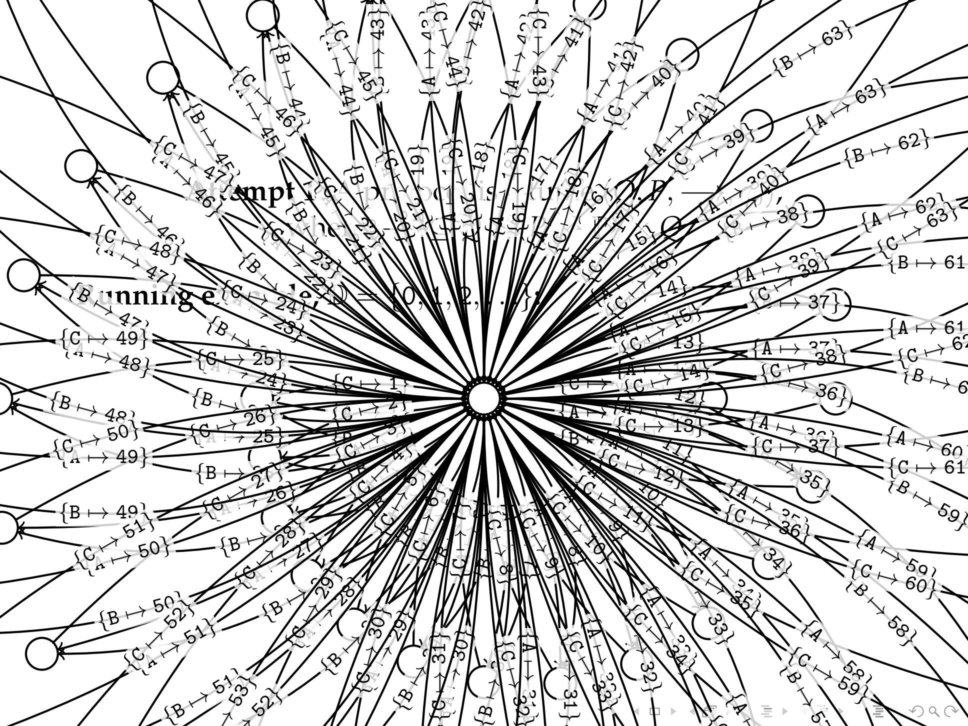
Running example, $\mathbb{D} = \{\mathtt{0}, \mathtt{1}, \mathtt{2}, \ldots\}$**:**



Attempt 1: A protocol is a tuple $(Q, P, \longrightarrow, q_0)$,
where $\longrightarrow \subseteq Q \times (P \rightharpoonup \mathbb{D}) \times Q$

Running example, $\mathbb{D} = \{\mathtt{0}, \mathtt{1}, \mathtt{2}, \ldots\}$**:**





Attempt 1 (A protocol is a tuple (Q, P, ⟶ …),
where …

Running example, D = {0, 1, 2, . . .};

- **Problem:** Infinitely many states
  (Because every state *encodes* a particular buffer content)
- **Solution:** Model buffers explicitly

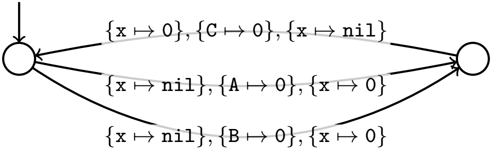
Henceforth:

- $\mathbb{M}$ denotes the set of memory cells
- $\mathbb{M} \rightharpoonup \mathbb{D}$ denotes the set of memory snapshots (ranged over by μ)

Attempt 2: A protocol is a tuple $(Q, P, M, \longrightarrow, q_0, \mu_0)$, where $\longrightarrow \subseteq Q \times (M \to \mathbb{D}) \times (P \rightharpoonup \mathbb{D}) \times (M \to \mathbb{D}) \times Q$

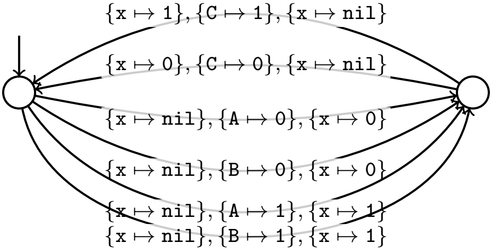
Attempt 2: A protocol is a tuple $(Q, P, M, \longrightarrow, q_0, \mu_0)$, where $\longrightarrow \subseteq Q \times (M \to \mathbb{D}) \times (P \rightharpoonup \mathbb{D}) \times (M \to \mathbb{D}) \times Q$

Running example, $\mathbb{D} = \{\texttt{0}\}$**:**


{x ↦ 0}, {C ↦ 0}, {x ↦ nil}
{x ↦ nil}, {A ↦ 0}, {x ↦ 0}
{x ↦ nil}, {B ↦ 0}, {x ↦ 0}

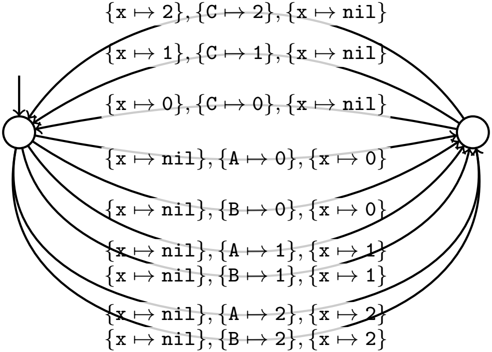
Attempt 2: A protocol is a tuple $(Q, P, M, \longrightarrow, q_0, \mu_0)$, where $\longrightarrow \subseteq Q \times (M \to \mathbb{D}) \times (P \rightharpoonup \mathbb{D}) \times (M \to \mathbb{D}) \times Q$

Running example, $\mathbb{D} = \{\mathtt{0}, \mathtt{1}\}$**:**


$\{\mathtt{x} \mapsto \mathtt{1}\}, \{\mathtt{C} \mapsto \mathtt{1}\}, \{\mathtt{x} \mapsto \mathtt{nil}\}$
$\{\mathtt{x} \mapsto \mathtt{0}\}, \{\mathtt{C} \mapsto \mathtt{0}\}, \{\mathtt{x} \mapsto \mathtt{nil}\}$
$\{\mathtt{x} \mapsto \mathtt{nil}\}, \{\mathtt{A} \mapsto \mathtt{0}\}, \{\mathtt{x} \mapsto \mathtt{0}\}$
$\{\mathtt{x} \mapsto \mathtt{nil}\}, \{\mathtt{B} \mapsto \mathtt{0}\}, \{\mathtt{x} \mapsto \mathtt{0}\}$
$\{\mathtt{x} \mapsto \mathtt{nil}\}, \{\mathtt{A} \mapsto \mathtt{1}\}, \{\mathtt{x} \mapsto \mathtt{1}\}$
$\{\mathtt{x} \mapsto \mathtt{nil}\}, \{\mathtt{B} \mapsto \mathtt{1}\}, \{\mathtt{x} \mapsto \mathtt{1}\}$

Attempt 2: A protocol is a tuple $(Q, P, M, \longrightarrow, q_0, \mu_0)$, where $\longrightarrow \subseteq Q \times (M \to \mathbb{D}) \times (P \rightharpoonup \mathbb{D}) \times (M \to \mathbb{D}) \times Q$

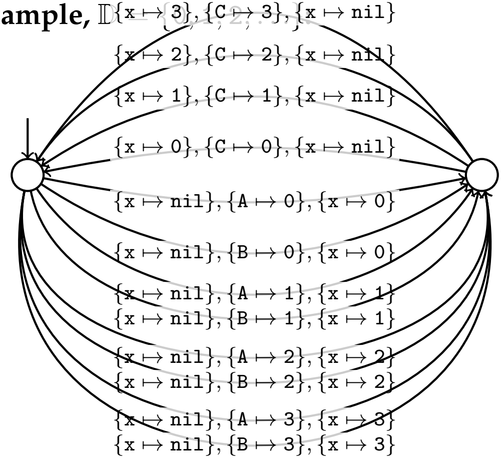
Running example, $\mathbb{D} = \{0, 1, 2, \ldots\}$**:**



{x ↦ 2}, {C ↦ 2}, {x ↦ nil}
{x ↦ 1}, {C ↦ 1}, {x ↦ nil}
{x ↦ 0}, {C ↦ 0}, {x ↦ nil}
{x ↦ nil}, {A ↦ 0}, {x ↦ 0}
{x ↦ nil}, {B ↦ 0}, {x ↦ 0}
{x ↦ nil}, {A ↦ 1}, {x ↦ 1}
{x ↦ nil}, {B ↦ 1}, {x ↦ 1}
{x ↦ nil}, {A ↦ 2}, {x ↦ 2}
{x ↦ nil}, {B ↦ 2}, {x ↦ 2}

Attempt 2: A protocol is a tuple $(Q, P, M, \longrightarrow, q_0, \mu_0)$, where $\longrightarrow \subseteq Q \times (M \to \mathbb{D}) \times (P \rightharpoonup \mathbb{D}) \times (M \to \mathbb{D}) \times Q$

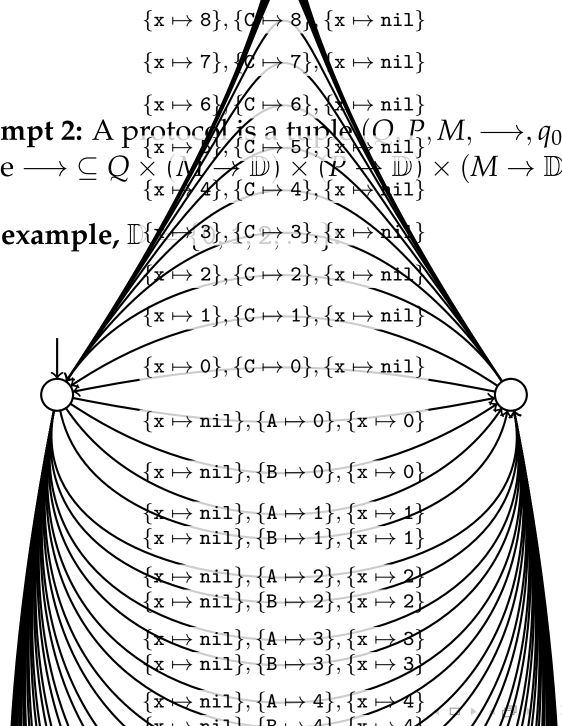
Running example, $\mathbb{D} = \{0, 1, 2, \ldots\}$:



{x ↦ 3}, {C ↦ 3}, {x ↦ nil}
{x ↦ 2}, {C ↦ 2}, {x ↦ nil}
{x ↦ 1}, {C ↦ 1}, {x ↦ nil}
{x ↦ 0}, {C ↦ 0}, {x ↦ nil}
{x ↦ nil}, {A ↦ 0}, {x ↦ 0}
{x ↦ nil}, {B ↦ 0}, {x ↦ 0}
{x ↦ nil}, {A ↦ 1}, {x ↦ 1}
{x ↦ nil}, {B ↦ 1}, {x ↦ 1}
{x ↦ nil}, {A ↦ 2}, {x ↦ 2}
{x ↦ nil}, {B ↦ 2}, {x ↦ 2}
{x ↦ nil}, {A ↦ 3}, {x ↦ 3}
{x ↦ nil}, {B ↦ 3}, {x ↦ 3}

Attempt 2: A protocol is a tuple $(Q, P, M, \longrightarrow, q_0, \mu_0)$, where $\longrightarrow \subseteq Q \times (M \to \mathbb{D}) \times (P \to \mathbb{D}) \times (M \to \mathbb{D}) \times Q$

Running example, $\mathbb{D} = \{0, 1, 2, \ldots\}$:



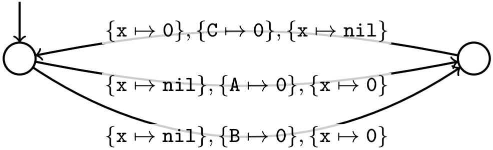
{x ↦ 8}, {C ↦ 8}, {x ↦ nil}
{x ↦ 7}, {C ↦ 7}, {x ↦ nil}
{x ↦ 6}, {C ↦ 6}, {x ↦ nil}
{x ↦ 5}, {C ↦ 5}, {x ↦ nil}
{x ↦ 4}, {C ↦ 4}, {x ↦ nil}
{x ↦ 3}, {C ↦ 3}, {x ↦ nil}
{x ↦ 2}, {C ↦ 2}, {x ↦ nil}
{x ↦ 1}, {C ↦ 1}, {x ↦ nil}
{x ↦ 0}, {C ↦ 0}, {x ↦ nil}
{x ↦ nil}, {A ↦ 0}, {x ↦ 0}
{x ↦ nil}, {B ↦ 0}, {x ↦ 0}
{x ↦ nil}, {A ↦ 1}, {x ↦ 1}
{x ↦ nil}, {B ↦ 1}, {x ↦ 1}
{x ↦ nil}, {A ↦ 2}, {x ↦ 2}
{x ↦ nil}, {B ↦ 2}, {x ↦ 2}
{x ↦ nil}, {A ↦ 3}, {x ↦ 3}
{x ↦ nil}, {B ↦ 3}, {x ↦ 3}
{x ↦ nil}, {A ↦ 4}, {x ↦ 4}
{x ↦ nil}, {B ↦ 4}, {x ↦ 4}

- **Problem:** Infinitely many transitions
  (Because every transition is a “concrete” data-flow)
- **Solution:** Model *sets* of concrete data-flows symbolically

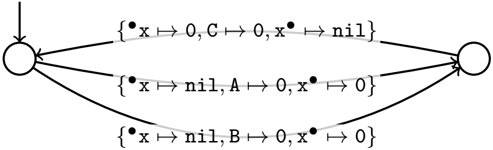
$$\left[\xrightarrow{\mu,\lambda,\mu'}\right] \sim \left[\xrightarrow{\{{}^{\bullet}m\mapsto\mu(m) \mid m\in\mathrm{Dom}(\mu)\}\cup\lambda\cup\{m^{\bullet}\mapsto\mu'(m) \mid m\in\mathrm{Dom}(\mu')\}}\right]$$

$$\left[\xrightarrow{\mu,\lambda,\mu'}\right] \sim \left[\xrightarrow{\{{}^\bullet m\mapsto\mu(m) \mid m\in\mathrm{Dom}(\mu)\}\cup\lambda\cup\{m^\bullet\mapsto\mu'(m) \mid m\in\mathrm{Dom}(\mu')\}}\right]$$

Running example, $\mathbb{D} = \{\mathtt{0}\}$**:**


$\{\mathtt{x} \mapsto \mathtt{0}\}, \{\mathtt{C} \mapsto \mathtt{0}\}, \{\mathtt{x} \mapsto \mathtt{nil}\}$
$\{\mathtt{x} \mapsto \mathtt{nil}\}, \{\mathtt{A} \mapsto \mathtt{0}\}, \{\mathtt{x} \mapsto \mathtt{0}\}$
$\{\mathtt{x} \mapsto \mathtt{nil}\}, \{\mathtt{B} \mapsto \mathtt{0}\}, \{\mathtt{x} \mapsto \mathtt{0}\}$

$\sim$


$\{{}^\bullet\mathtt{x} \mapsto \mathtt{0}, \mathtt{C} \mapsto \mathtt{0}, \mathtt{x}^\bullet \mapsto \mathtt{nil}\}$
$\{{}^\bullet\mathtt{x} \mapsto \mathtt{nil}, \mathtt{A} \mapsto \mathtt{0}, \mathtt{x}^\bullet \mapsto \mathtt{0}\}$
$\{{}^\bullet\mathtt{x} \mapsto \mathtt{nil}, \mathtt{B} \mapsto \mathtt{0}, \mathtt{x}^\bullet \mapsto \mathtt{0}\}$

Henceforth:

- $\mathbb{X} = \mathbb{P} \cup \{{}^{\bullet}m \mid m \in M\} \cup \{m^{\bullet} \mid m \in M\}$ denotes the set of data variables
- $\mathbb{X} \rightharpoonup \mathbb{D}$ denotes the set of data assignments (ranged over by σ)

- **Observation:** Every data assignment models a data-flow (“Model *sets* of concrete data-flows symbolically”)

- **Observation:** Every data assignment models a data-flow ("Model *sets* of concrete data-flows symbolically")

- **To do:** Symbolically represent sets of data assignments
- **Approach:** Define a logic whose semantics is defined in terms of data assignments
  - Formulas: $\mathcal{L}$
  - Entailment: $\models \subseteq (\mathbb{X} \rightharpoonup \mathbb{D}) \times \mathcal{L}$

- **Observation:** Every data assignment models a data-flow ("Model *sets* of concrete data-flows symbolically")

- **To do:** Symbolically represent sets of data assignments
- **Approach:** Define a logic whose semantics is defined in terms of data assignments
  - Formulas: $\mathcal{L}$
  - Entailment: $\models \subseteq (\mathbb{X} \rightharpoonup \mathbb{D}) \times \mathcal{L}$

- $\xrightarrow{P,\varphi} \sim \{ \xrightarrow{\sigma} \mid P = \mathrm{Dom}(\sigma) \cap \mathbb{P} \ \textbf{and} \ \sigma \models \varphi \}$
  - P is called a synchronization constraint
  - φ is called a data constraint

Syntax:

$$\begin{array}{lcll}
a & ::= & \top \mid \bot \mid x = x \mid \texttt{Keep}(M) & \text{(data atoms)} \\
\ell & ::= & a \mid \neg a & \text{(data literals)} \\
\varphi & ::= & \ell \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \exists x . \varphi & \text{(data constraints)}
\end{array}$$

Semantics:

$$\begin{array}{lcl}
\sigma \models x_1 = x_2 & \textbf{iff} & \sigma(x_1) = \sigma(x_2) \\
\sigma \models \texttt{Keep}(M) & \textbf{iff} & \sigma \models {}^{\bullet}m = m^{\bullet} \;\; \textbf{for all} \;\; m \in M
\end{array}$$

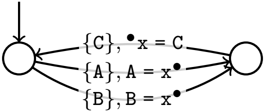
Henceforth:

- $\mathbb{DC}$ denotes the set of data constraints
- $\mathcal{L} := \mathbb{DC}$

Attempt 3: A protocol is a tuple $(Q, P, M, \longrightarrow, q_0, \mu_0)$, where $\longrightarrow \subseteq Q \times 2^P \times \mathbb{DC} \times Q$

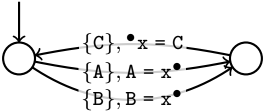
Attempt 3: A protocol is a tuple $(Q, P, M, \longrightarrow, q_0, \mu_0)$,
where $\longrightarrow \subseteq Q \times 2^P \times \mathbb{DC} \times Q$

Running example, $\mathbb{D} = \{\texttt{0}\}$**:**



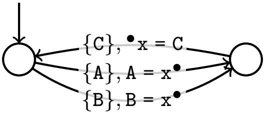
Attempt 3: A protocol is a tuple $(Q, P, M, \longrightarrow, q_0, \mu_0)$, where $\longrightarrow \subseteq Q \times 2^P \times \mathbb{DC} \times Q$

Running example, $\mathbb{D} = \{\mathtt{0}, \mathtt{1}\}$**:**



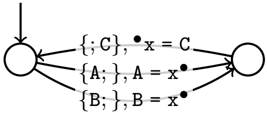
Attempt 3: A protocol is a tuple $(Q, P, M, \longrightarrow, q_0, \mu_0)$,
where $\longrightarrow \subseteq Q \times 2^P \times \mathbb{D}\mathbb{C} \times Q$

Running example, $\mathbb{D} = \{\mathtt{0}, \mathtt{1}, \mathtt{2}, \ldots\}$**:**



Finally: A protocol is a tuple $(Q, (P^{\text{in}}, P^{\text{out}}), M, \longrightarrow, q_0, \mu_0)$, where $\longrightarrow \subseteq Q \times 2^P \times \mathbb{D}\mathbb{C} \times Q$

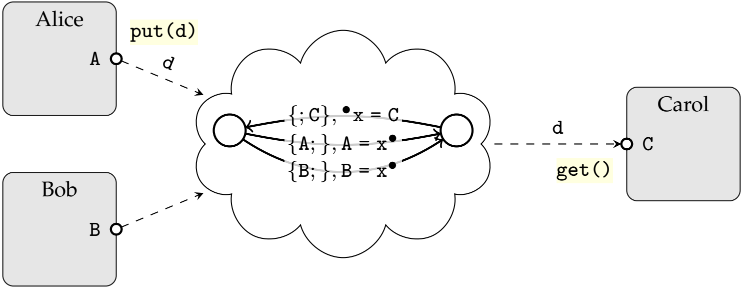
Running example, $\mathbb{D} = \{0, 1, 2, \ldots\}$**:**



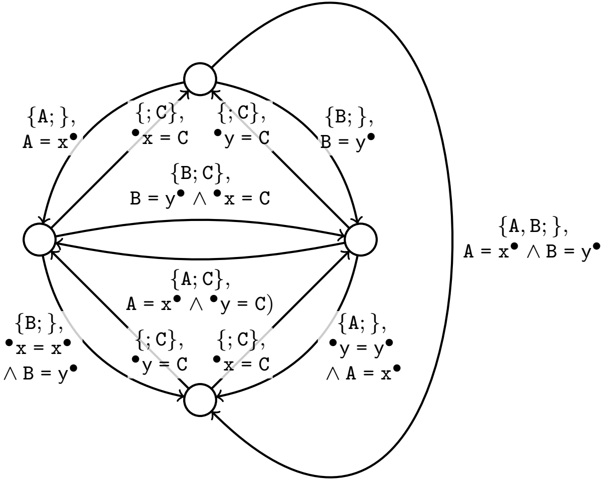
About directions:

- An *output port* to a process is an *input port* to the protocol
- An *input port* to a process is an *output port* to the protocol

Running example:

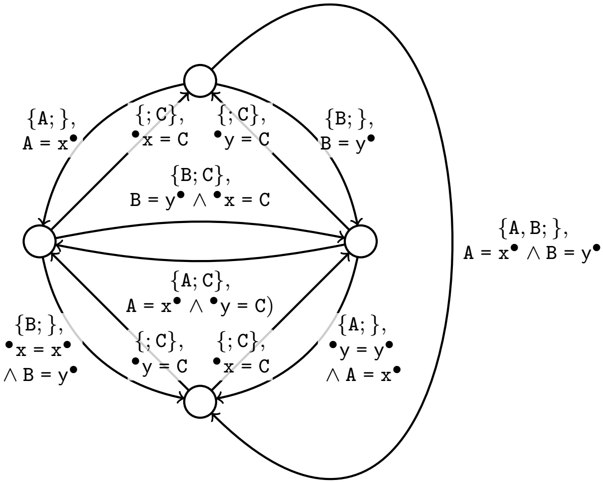


But...



$\{A;\},$ $A = x^\bullet$
$\{;C\},$ ${}^\bullet x = C$
$\{;C\},$ ${}^\bullet y = C$
$\{B;\},$ $B = y^\bullet$
$\{B;C\},$ $B = y^\bullet \wedge {}^\bullet x = C$
$\{A,B;\},$ $A = x^\bullet \wedge B = y^\bullet$
$\{A;C\},$ $A = x^\bullet \wedge {}^\bullet y = C)$
$\{B;\},$ ${}^\bullet x = x^\bullet$ $\wedge B = y^\bullet$
$\{;C\},$ ${}^\bullet y = C$
$\{;C\},$ ${}^\bullet x = C$
$\{A;\},$ ${}^\bullet y = y^\bullet$ $\wedge A = x^\bullet$

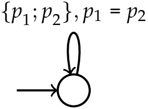
Running example″: Async., reliab., unord., ~~transact.~~, consid.



For k producers, 2^k states and $\mathcal{O}(k \cdot 2^k)$ transitions *per state*

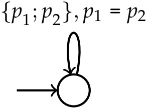
Compositional construction, through *multiplication*

Compositional construction, through *multiplication*

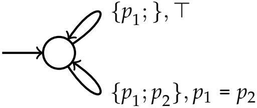


$\mathsf{Sync}(p_1; p_2)$

Compositional construction, through *multiplication*

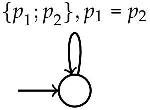


$\mathsf{Sync}(p_1; p_2)$

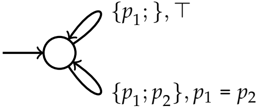


$\mathsf{LossySync}(p_1; p_2)$

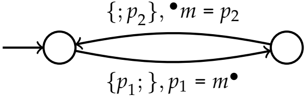
Compositional construction, through *multiplication*



Sync$(p_1;p_2)$

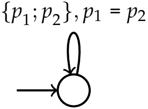


LossySync$(p_1;p_2)$

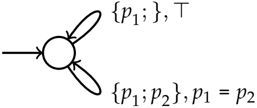


Fifo$(m)(p_1;p_2)$

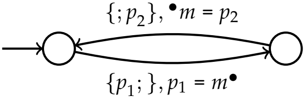
Compositional construction, through *multiplication*



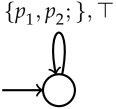
$\mathsf{Sync}(p_1; p_2)$



$\mathsf{LossySync}(p_1; p_2)$

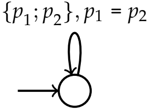


$\mathsf{Fifo}(m)(p_1; p_2)$

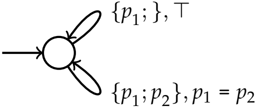


$\mathsf{Drain}(p_1, p_2;)$

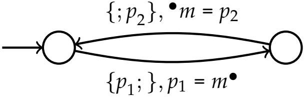
Compositional construction, through *multiplication*



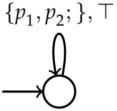
Sync$(p_1; p_2)$



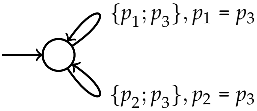
LossySync$(p_1; p_2)$



Fifo$(m)(p_1; p_2)$

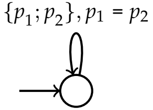


Drain$(p_1, p_2;)$

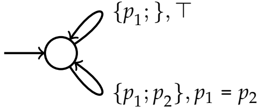


Merger$(p_1, p_2; p_3)$

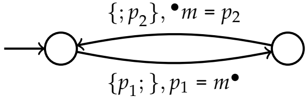
Compositional construction, through *multiplication*



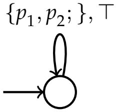
$\mathsf{Sync}(p_1;p_2)$



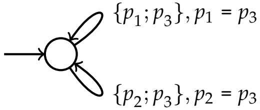
$\mathsf{LossySync}(p_1;p_2)$



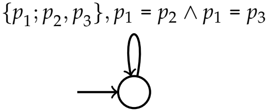
$\mathsf{Fifo}(m)(p_1;p_2)$



$\mathsf{Drain}(p_1,p_2;)$

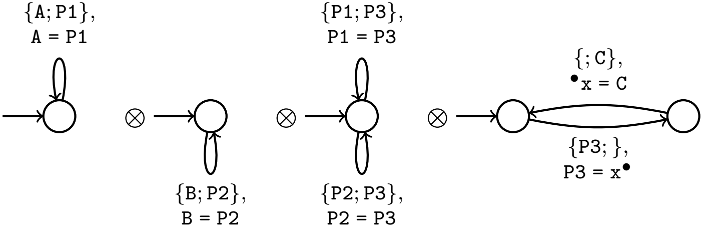


$\mathsf{Merger}(p_1,p_2;p_3)$

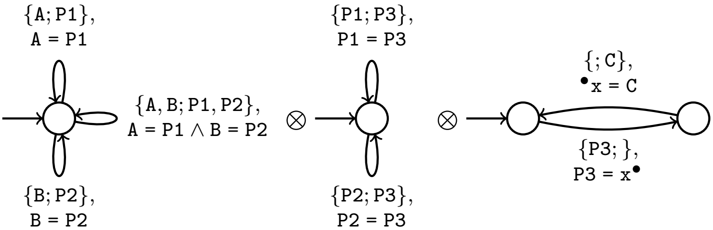


$\mathsf{Replicator}(p_1;p_2,p_3)$

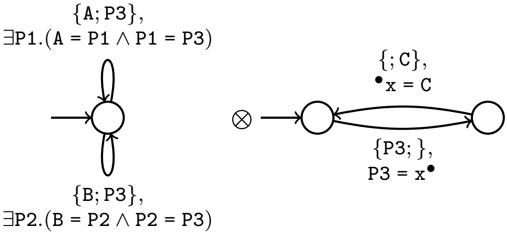
Running example:



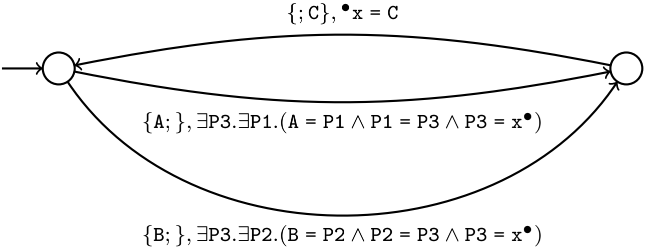
Running example:



Running example:



Running example:

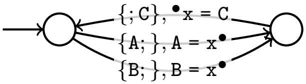


$\{;\mathtt{C}\}, {}^{\bullet}\mathtt{x} = \mathtt{C}$

$\{\mathtt{A};\}, \exists \mathtt{P3}.\exists \mathtt{P1}.(\mathtt{A} = \mathtt{P1} \wedge \mathtt{P1} = \mathtt{P3} \wedge \mathtt{P3} = \mathtt{x}^{\bullet})$

$\{\mathtt{B};\}, \exists \mathtt{P3}.\exists \mathtt{P2}.(\mathtt{B} = \mathtt{P2} \wedge \mathtt{P2} = \mathtt{P3} \wedge \mathtt{P3} = \mathtt{x}^{\bullet})$

Running example:



Definition:

$$\begin{pmatrix} Q_1, \\ P_1^{\text{in}}, \\ P_1^{\text{out}}, \\ M_1, \\ \longrightarrow_1, \\ q_1^0 \end{pmatrix} \otimes \begin{pmatrix} Q_2, \\ P_2^{\text{in}}, \\ P_2^{\text{out}}, \\ M_2, \\ \longrightarrow_2, \\ q_2^0 \end{pmatrix} = \begin{pmatrix} Q_1 \times Q_2, \\ (P_1^{\text{in}} \cup P_2^{\text{in}}) \setminus (P_1^{\text{out}} \cup P_2^{\text{out}}), \\ (P_1^{\text{out}} \cup P_2^{\text{out}}) \setminus (P_1^{\text{in}} \cup P_2^{\text{in}}), \\ M_1 \cup M_2, \\ \longrightarrow, \\ (q_1^0, q_2^0) \end{pmatrix}$$

$$\textbf{if} \;\; \begin{bmatrix} P_1^{\text{in}} \cap P_2^{\text{in}} = P_1^{\text{out}} \cap P_2^{\text{out}} = \emptyset \\ \textbf{and} \;\; M_1 \cap M_2 = \emptyset \end{bmatrix}$$

where $\longrightarrow$ is the smallest relation induced by the rules on the next slide

$$\frac{\begin{array}{l} q_1 \xrightarrow{P_1,\varphi_1}_1 q_1' \ \ \textbf{and} \ \ q_2 \xrightarrow{P_2,\varphi_2}_2 q_2' \\ \textbf{and} \ \ (P_1^{\mathrm{in}} \cup P_1^{\mathrm{out}}) \cap P_2 = (P_2^{\mathrm{in}} \cup P_2^{\mathrm{out}}) \cap P_1 \end{array}}{(q_1, q_2) \xrightarrow{(P_1 \cup P_2) \setminus (P_1 \cap P_2), \exists (P_1 \cap P_2). \varphi_1 \wedge \varphi_2} (q_1', q_2')}$$

$$\frac{\begin{array}{c} q_1 \xrightarrow{P_1,\varphi_1}_1 q_1' \;\; \textbf{and} \;\; q_2 \xrightarrow{P_2,\varphi_2}_2 q_2' \\ \textbf{and} \;\; (P_1^{\mathrm{in}} \cup P_1^{\mathrm{out}}) \cap P_2 = (P_2^{\mathrm{in}} \cup P_2^{\mathrm{out}}) \cap P_1 \end{array}}{(q_1, q_2) \xrightarrow{(P_1 \cup P_2) \setminus (P_1 \cap P_2), \exists (P_1 \cap P_2). \varphi_1 \wedge \varphi_2} (q_1', q_2')}$$

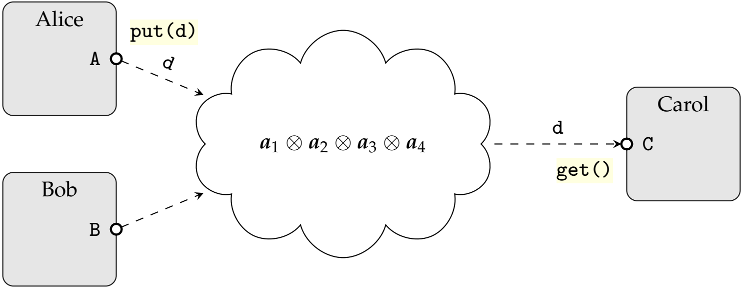
$$\frac{\begin{array}{c} q_1 \xrightarrow{P_1,\varphi_1}_1 q_1' \;\; \textbf{and} \;\; q_2 \in Q_2 \\ \textbf{and} \;\; (P_2^{\mathrm{in}} \cup P_2^{\mathrm{out}}) \cap P_1 = \emptyset \end{array}}{(q_1, q_2) \xrightarrow{P_1, \varphi_1 \wedge \texttt{Keep}(M_2)} (q_1', q_2)}$$

$$\frac{\begin{array}{c} q_1 \xrightarrow{P_1,\varphi_1}_1 q_1' \ \ \textbf{and} \ \ q_2 \xrightarrow{P_2,\varphi_2}_2 q_2' \\ \textbf{and} \ \ (P_1^{\mathrm{in}} \cup P_1^{\mathrm{out}}) \cap P_2 = (P_2^{\mathrm{in}} \cup P_2^{\mathrm{out}}) \cap P_1 \end{array}}{(q_1, q_2) \xrightarrow{(P_1 \cup P_2) \setminus (P_1 \cap P_2), \exists (P_1 \cap P_2) . \varphi_1 \wedge \varphi_2} (q_1', q_2')}$$

$$\frac{\begin{array}{c} q_1 \xrightarrow{P_1,\varphi_1}_1 q_1' \ \ \textbf{and} \ \ q_2 \in Q_2 \\ \textbf{and} \ \ (P_2^{\mathrm{in}} \cup P_2^{\mathrm{out}}) \cap P_1 = \emptyset \end{array}}{(q_1, q_2) \xrightarrow{P_1, \varphi_1 \wedge \mathtt{Keep}(M_2)} (q_1', q_2)}$$

$$\frac{\begin{array}{c} q_2 \xrightarrow{P_2,\varphi_2}_2 q_2' \ \ \textbf{and} \ \ q_1 \in Q_1 \\ \textbf{and} \ \ (P_1^{\mathrm{in}} \cup P_1^{\mathrm{out}}) \cap P_2 = \emptyset \end{array}}{(q_1, q_2) \xrightarrow{P_2, \varphi_2 \wedge \mathtt{Keep}(M_1)} (q_1, q_2')}$$

Running example:



But...

- **Observation:** Directly/explicitly multiplying automata requires a significant intellectual effort
- Can we think of more convenient ways of writing multiplication expressions?

Approach: First $\underbrace{\text{semantics}}_{\substack{\text{What are suitable}\\ \text{models of protocols?}}}$, then $\overbrace{\text{syntax}}^{\substack{\text{What are suitable}\\ \text{programming constructs}\\ \text{to denote such models?}}}$

Summary:

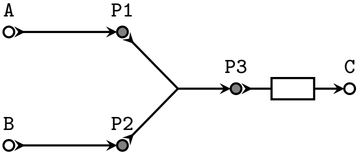
- *Interactions* are streams
- *Protocols* are:
  - languages (of streams)
  - automata
  - multiplication expressions (over automata)

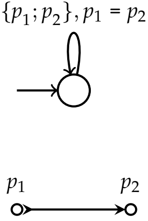
- (How are automata related to languages!?)

One syntax

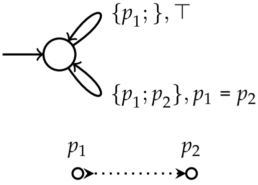
Approach: *Denote* multiplication expressions by (hyper)digraphs

- $[\![$vertex$]\!]$ = port
- $[\![$(hyper)arc$]\!]$ = automaton over connected $[\![$vertices$]\!]$

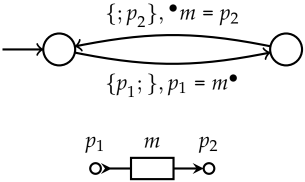




$\mathsf{Sync}(p_1;p_2)$

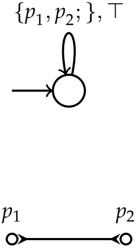


$\mathsf{LossySync}(p_1;p_2)$

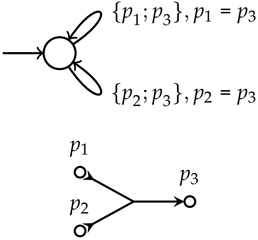

$\{;p_2\}, {}^{\bullet}m = p_2$
$\{p_1;\}, p_1 = m^{\bullet}$
p_1
m
p_2

$\mathsf{Fifo}(m)(p_1;p_2)$

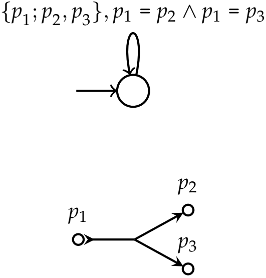
(See demo)



$\mathsf{Drain}(p_1, p_2;)$



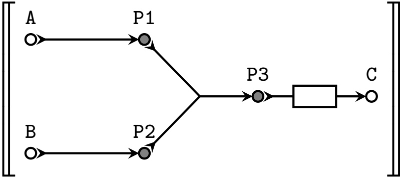
$\mathsf{Merger}(p_1, p_2; p_3)$



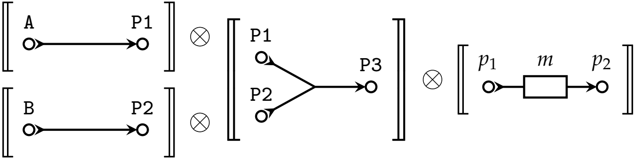
$\mathsf{Replicator}(p_1; p_2, p_3)$

(See demo)

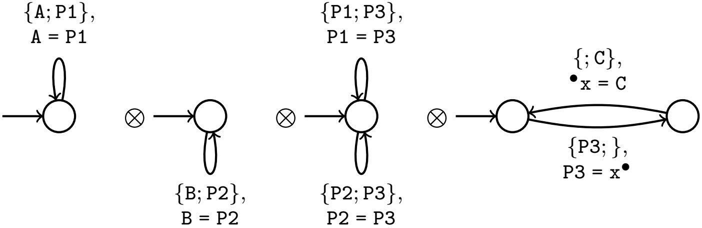
Running example:



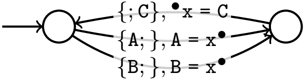
Running example:



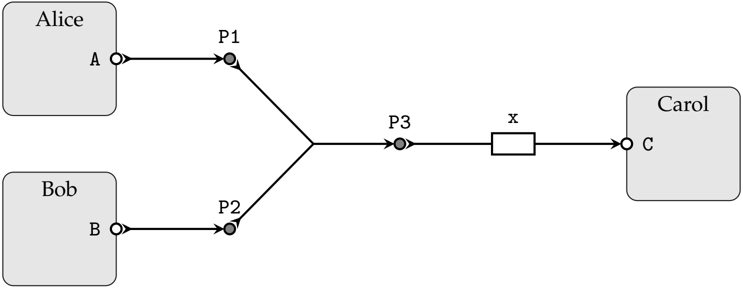
Running example:



Running example:

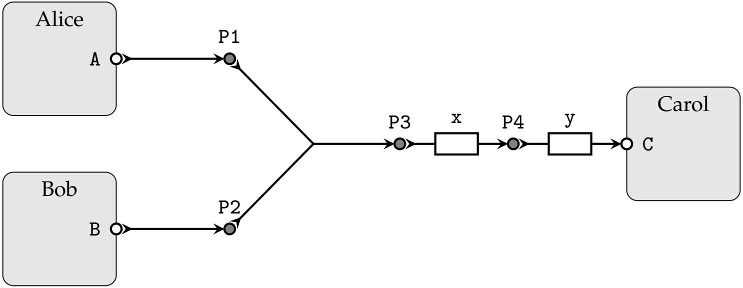


Running example:



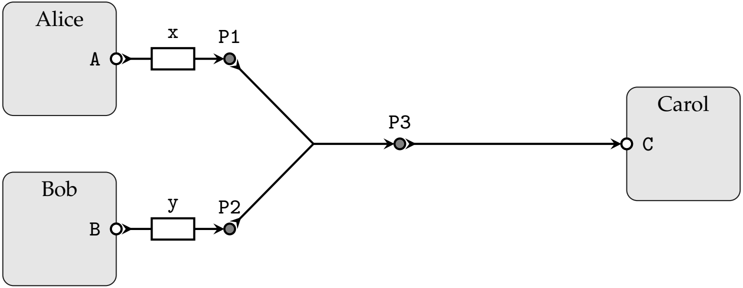
(See demo)

Running example$'$:



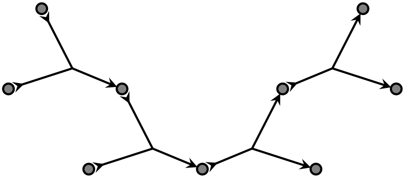
(See demo)

Running example″:

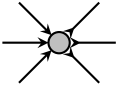


(See demo)

- **Suppose:** Merge from *three inputs*; Replicate to *three outputs*
- Inconvenient:

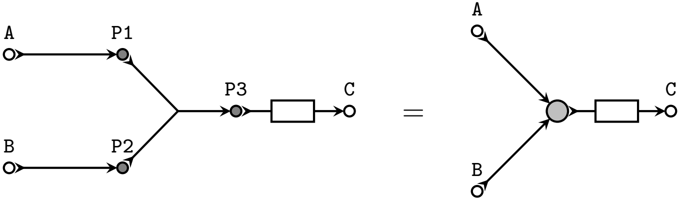


- Convenient:



- (See demo)

Running example:

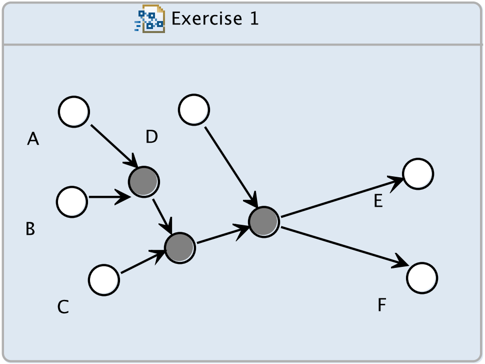


- "Fat" vertices (i.e., abbreviations of sequences of Mergers and Replicators) are called nodes
- Binary arcs are called channels
- (Hyper)digraphs are called circuits
- This graphical language is called Reo [Arb]

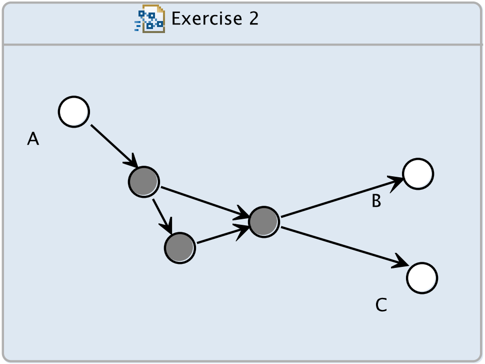
- "Fat" vertices (i.e., abbreviations of sequences of Mergers and Replicators) are called nodes
- Binary arcs are called channels
- (Hyper)digraphs are called circuits
- This graphical language is called Reo [Arb]

- Eclipse plugins (editor, animator, compiler) for Reo development
- `http://www.open.ou.nl/ssj/prdk`
- (See demo)

Exercises

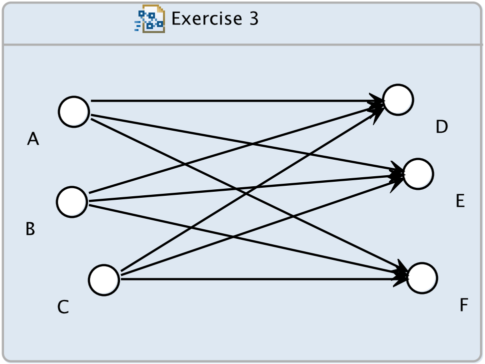


Exercise 1: Describe the protocol specified by this circuit (in natural language or as an automaton)

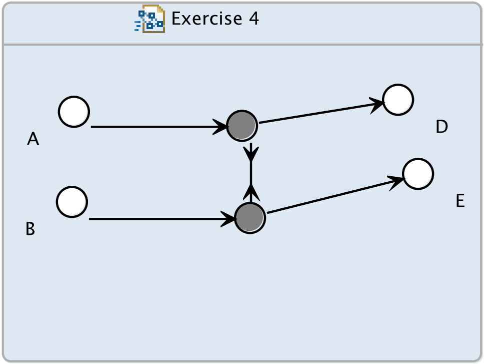


Exercise 2: Describe the protocol specified by this circuit (in natural language or as an automaton)

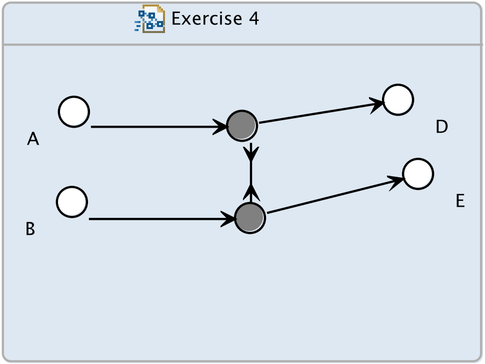
Exercise 3: Design a circuit for a protocol among 3 producers and 3 consumers, where a message sent by a producer is synchronously received by all consumers



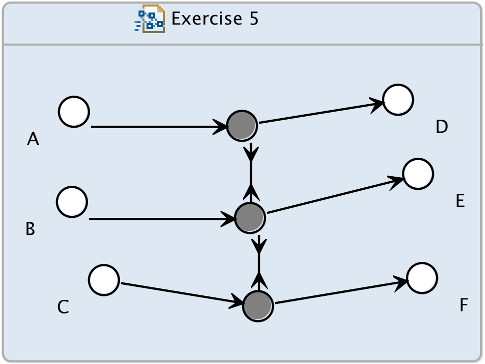
Exercise 3: Design a circuit for a protocol among 3 producers and 3 consumers, where a message sent by a producer is synchronously received by all consumers



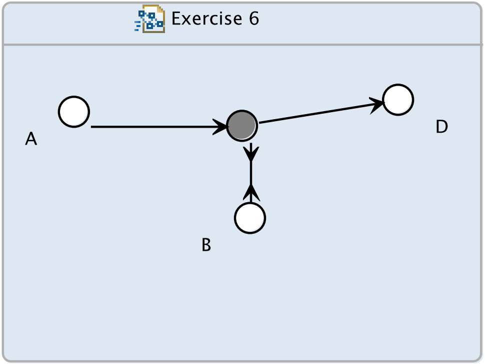
Exercise 4: Describe the protocol specified by this circuit (in natural language or as an automaton)



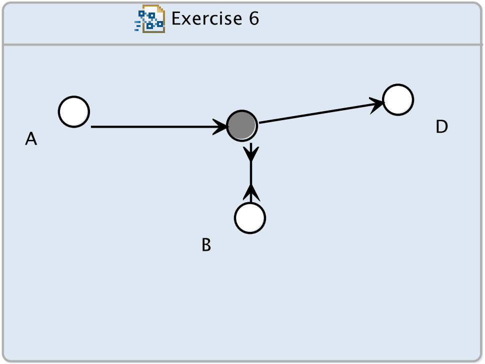
Exercise 5: Extend the circuit from Exercise 4 for a protocol among 3 producers and 3 consumers



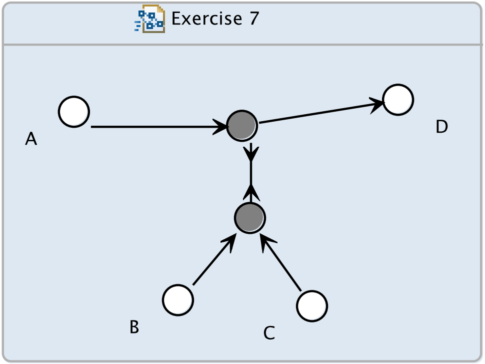
Exercise 5: Extend the circuit from Exercise 4 for a protocol among 3 producers and 3 consumers



Exercise 6: Describe the protocol specified by this circuit (in natural language or as an automaton)

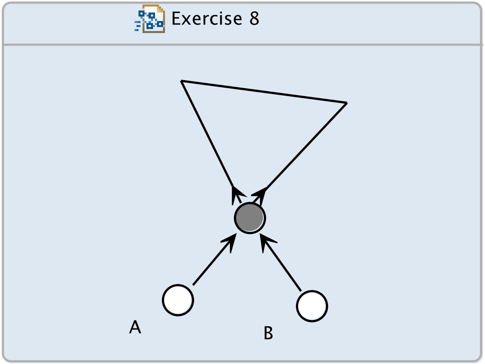


Exercise 7: Extend the circuit from Exercise 6 for a protocol with 2 regulators

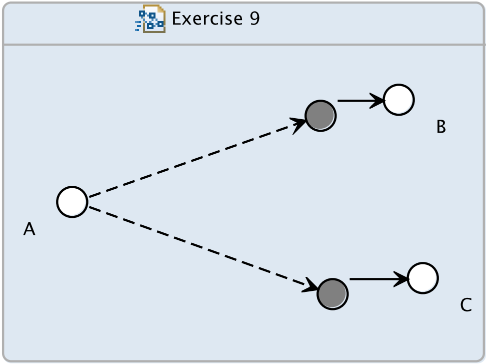


Exercise 7: Extend the circuit from Exercise 6 for a protocol with 2 regulators

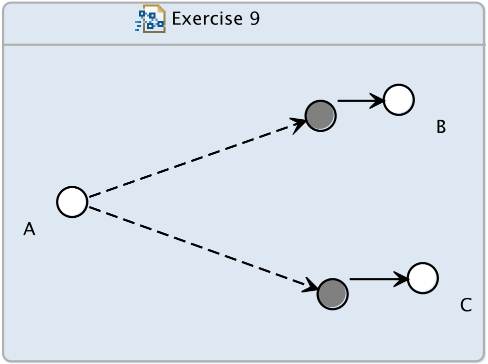
Exercise 8: Design a circuit for a protocol among two producers, where producers send messages only asynchronously (cf. Drain).



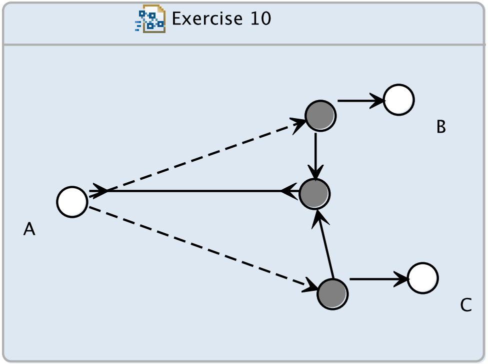
Exercise 8: Design a circuit for a protocol among two producers, where producers send messages only asynchronously (cf. Drain).



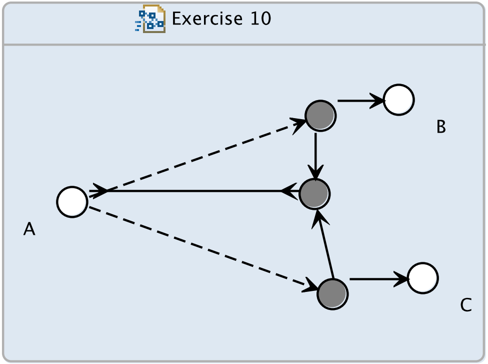
Exercise 9: Describe the protocol specified by this circuit (in natural language or as an automaton)



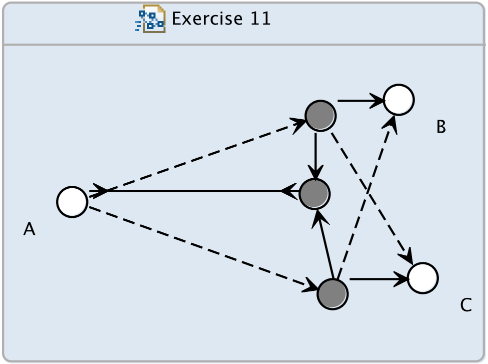
Exercise 10: Extend the circuit from Exercise 9 for a protocol where *exactly one* consumer synchronously receives the message sent by the producer (cf. XOR)



Exercise 10: Extend the circuit from Exercise 9 for a protocol where *exactly one* consumer synchronously receives the message sent by the producer (cf. XOR)



Exercise 11: Extend the circuit from Exercise 10 for a protocol where *at least one* consumer synchronously receives the message sent by the producer (cf. OR)



Exercise 11: Extend the circuit from Exercise 10 for a protocol where *at least one* consumer synchronously receives the message sent by the producer (cf. OR)

Summary:

- (Hyper)digraphs denote multiplication expressions
- Reo circuits are (hyper)digraphs
- **Comparison:**

GPL (e.g., Java)	DSL (e.g., Reo+automata)
action-based	interaction-based
process as primitive	protocol as primitive
imperative/functional	declarative/relational

Summary:

- (Hyper)digraphs denote multiplication expressions
- Reo circuits are (hyper)digraphs
- **Comparison:**

GPL (e.g., Java)	DSL (e.g., Reo+automata)
action-based	interaction-based
process as primitive	protocol as primitive
imperative/functional	declarative/relational

- Other syntaxes:
  - Pr
  - Petri nets [Sif]
  - Connector algebras [Sif]
  - UML Sequence/Activity Diagrams, BPMN, BPEL

How does the compiler work?

Two basic approaches:

- Distributed approach
- Centralized approach

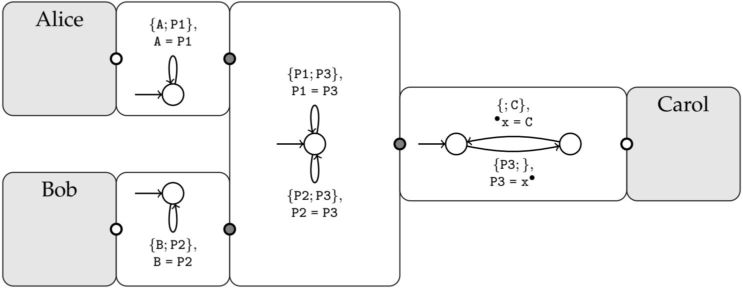
Distributed compilation:

1. Find a “small” automaton for the *local* behavior of every node/channel in the input circuit

Distributed compilation:

1. Find a “small” automaton for the *local* behavior of every node/channel in the input circuit
2. Translate the resulting small automata into Java code for their run-time execution, *using a consensus algorithm*

Running example:



Distributed execution:

1. Protocol thread α_i awakes to handle event from θ on port p
2. For all transitions $q^{\text{curr}} \xrightarrow{P,\varphi} q'$ such that $p \in P$:

Distributed execution:

1. Protocol thread α_i awakes to handle event from θ on port p
2. For all transitions $q^{\text{curr}} \xrightarrow{P,\varphi} q'$ such that $p \in P$:
    1. α_i checks synchronization constraint P:
        - Are “neighboring” process threads “behind” public ports in P ready for data-flow through those ports?
        - Are “neighboring” protocol threads “behind” private ports in P ready for data-flow through those ports? *(New events!)*

Distributed execution:

1. Protocol thread α_i awakes to handle event from θ on port p
2. For all transitions $q^{\text{curr}} \xrightarrow{P,\varphi} q'$ such that $p \in P$:
    1. α_i checks synchronization constraint P:
        - Are “neighboring” process threads “behind” public ports in P ready for data-flow through those ports?
        - Are “neighboring” protocol threads “behind” private ports in P ready for data-flow through those ports? *(New events!)*
    2. α_i checks data constraint φ:
        - Do data to be exchanged through ports in P satisfy φ?

Distributed execution:

1. Protocol thread α_i awakes to handle event from θ on port p
2. For all transitions $q^{\text{curr}} \xrightarrow{P,\varphi} q'$ such that $p \in P$:
    1. α_i checks synchronization constraint P:
        - Are "neighboring" process threads "behind" public ports in P ready for data-flow through those ports?
        - Are "neighboring" protocol threads "behind" private ports in P ready for data-flow through those ports? *(New events!)*
    2. α_i checks data constraint φ:
        - Do data to be exchanged through ports in P satisfy φ?
    3. α_i commits to make transition, and informs θ
    4. α_i awaits confirmation from θ, and makes transition
    5. α_i breaks the loop
3. α_i goes back to sleep

Centralized compilation:

1. Find a “small” automaton for the *local* behavior of every node/channel in the input circuit

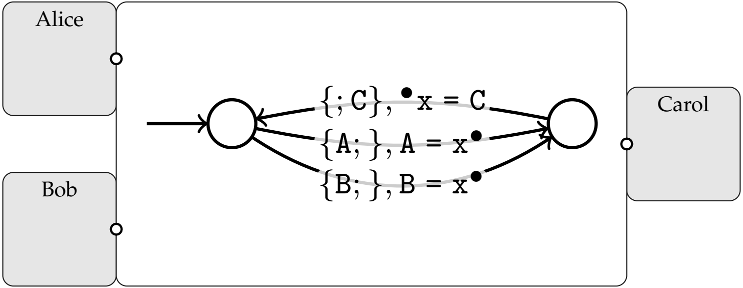
Centralized compilation:

1. Find a “small” automaton for the *local* behavior of every node/channel in the input circuit
2. Multiply the resulting small automata into a “big” automaton for the *global* behavior of the input circuit

Centralized compilation:

1. Find a “small” automaton for the *local* behavior of every node/channel in the input circuit
2. Multiply the resulting small automata into a “big” automaton for the *global* behavior of the input circuit
3. Translate the resulting big automaton into Java code for its run-time execution

Running example:



Centralized execution:

1. Protocol thread α awakes to handle event from process thread on port p
2. For all transitions $q^{\text{curr}} \xrightarrow{P,\varphi} q'$ such that $p \in P$:

Centralized execution:

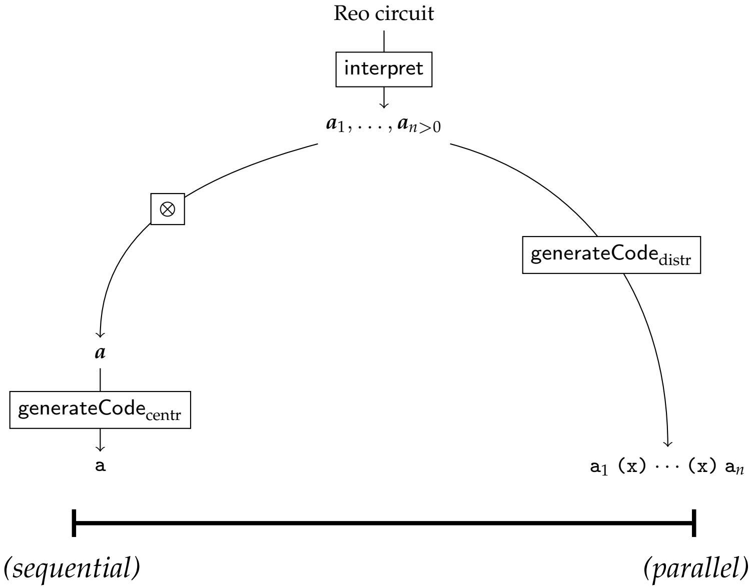
1. Protocol thread α awakes to handle event from process thread on port p
2. For all transitions $q^{\text{curr}} \xrightarrow{P,\varphi} q'$ such that $p \in P$:
    1. α checks synchronization constraint P:
        - Are "neighboring" process threads "behind" public ports in P ready for data-flow through those ports?
        - ~~Are "neighboring" protocol threads "behind" private ports in P ready for data-flow through those ports? *(New events!)*~~

Centralized execution:

1. Protocol thread α awakes to handle event from process thread on port p
2. For all transitions $q^{\text{curr}} \xrightarrow{P,\varphi} q'$ such that $p \in P$:
    1. α checks synchronization constraint P:
        - Are "neighboring" process threads "behind" public ports in P ready for data-flow through those ports?
        - ~~Are "neighboring" protocol threads "behind" private ports in P ready for data-flow through those ports? *(New events!)*~~
    2. α checks data constraint φ:
        - Do data to be exchanged through ports in P satisfy φ?

Centralized execution:

1. Protocol thread α awakes to handle event from process thread on port p
2. For all transitions $q^{\text{curr}} \xrightarrow{P,\varphi} q'$ such that $p \in P$:
    1. α checks synchronization constraint P:
        - Are “neighboring” process threads “behind” public ports in P ready for data-flow through those ports?
        - ~~Are “neighboring” protocol threads “behind” private ports in P ready for data-flow through those ports? *(New events!)*~~
    2. α checks data constraint φ:
        - Do data to be exchanged through ports in P satisfy φ?
    3. ~~α_i commits to make transition, and informs θ~~
    4. α ~~awaits confirmation from θ, and~~ makes transition
    5. α breaks the loop
3. α goes back to sleep



Reo circuit

interpret

$\boldsymbol{a}_1, \ldots, \boldsymbol{a}_{n>0}$

$\otimes$

$\boldsymbol{a}$

$\mathsf{generateCode}_{\mathrm{centr}}$

`a`

$\mathsf{generateCode}_{\mathrm{distr}}$

$\mathtt{a}_1 \; \mathtt{(x)} \cdots \mathtt{(x)} \; \mathtt{a}_n$

(sequential)

(parallel)

- The compiler uses (at least) the centralized compilation
- (See demo)

- The compiler uses (at least) the centralized compilation
- (See demo)

- **Observation:** State space explosion at compile-time
  - *Two* producers: 4 states, 11 transitions
  - *Four* producers: 16 states, 173 transitions
  - *Eight* producers: 256 states, 23801 transitions

- The compiler uses (at least) the centralized compilation
- (See demo)

- **Observation:** State space explosion at compile-time
  - *Two* producers: 4 states, 11 transitions
  - *Four* producers: 16 states, 173 transitions
  - *Eight* producers: 256 states, 23801 transitions
- **Observation:** Oversequentialization at run-time

Optimizations are necessary

- Transformations at the level of automata instead of at the level of generated code
- **Formally:** Behavior-preserving functions from automata (low performance) to automata (higher performance)

$$Reo \xrightarrow{f} Aut_0 \xrightarrow{g_1} Aut_1 \xrightarrow{g_k \circ \cdots \circ g_2} Aut_k \xrightarrow{h} Java$$

- I want to discuss two such optimizations

- Transformations at the level of automata instead of at the level of generated code
- **Formally:** Behavior-preserving functions from automata (low performance) to automata (higher performance)

$$Reo \xrightarrow{f} Aut_0 \xrightarrow{g_1} Aut_1 \xrightarrow{g_k \circ \cdots \circ g_2} Aut_k \xrightarrow{h} Java$$

- I want to discuss two such optimizations

But!

- What is the meaning of “behavior-preserving”?
- What is the meaning of “behavior”?

- Remember that, *initially*, protocols were languages
- Thus, the behavior of an automaton is its accepted language
- Draw inspiration from classical pushdown automata

- Instantaneous description: (q, w, μ)
    - $q \in Q$ is the current state
    - $w : \mathbb{N} \to (P \rightharpoonup \mathbb{D})$ is the remaining word (input tape)
    - $\mu : M \to \mathbb{D}$ is the current memory snapshot (stack)

- Instantaneous description: (q, w, μ)
    - $q \in Q$ is the current state
    - $w : \mathbb{N} \to (P \rightharpoonup \mathbb{D})$ is the remaining word (input tape)
    - $\mu : M \to \mathbb{D}$ is the current memory snapshot (stack)

- Move: $(q, w, \mu') \vdash (q', w', \mu')$

- Instantaneous description: (q, w, μ)
    - $q \in Q$ is the current state
    - $w : \mathbb{N} \to (P \rightharpoonup \mathbb{D})$ is the remaining word (input tape)
    - $\mu : M \to \mathbb{D}$ is the current memory snapshot (stack)

- Move: $(q, w, \mu') \vdash (q', w', \mu')$

- Language: $\{w \mid (q_0, w, \mu_0) \vdash (q_1, w', \mu_1) \vdash (q_2, w'', \mu_2) \vdash \cdots\}$
- Language equivalence: $\approx$

$$\frac{}{(q_i, w, \mu_i) \vdash (q_{i+1}, w', \mu_{i+1})}$$

$$\frac{q_i \xrightarrow{P,\varphi} q_{i+1}}{(q_i, w, \mu_i) \vdash (q_{i+1}, w', \mu_{i+1})}$$

$$q_i \xrightarrow{P,\varphi} q_{i+1}$$

and $\mathrm{Dom}(w(0)) = P$

$$(q_i, w, \mu_i) \vdash (q_{i+1}, w', \mu_{i+1})$$

$$\frac{\begin{array}{l} q_i \xrightarrow{P,\varphi} q_{i+1} \\ \\ \textbf{and}\ \ \mathrm{Dom}(w(0)) = P \\ \textbf{and}\ \ \{{}^{\bullet}m \mapsto \mu_i(m) \mid m \in M\} \\ \qquad\quad \cup\, w(0) \cup \{m^{\bullet} \mapsto \mu_{i+1}(m) \mid m \in M\} \models \varphi \end{array}}{(q_i, w, \mu_i) \vdash (q_{i+1}, w', \mu_{i+1})}$$

$$\frac{\begin{array}{ll} q_i \xrightarrow{P,\varphi} q_{i+1} & \\ \textbf{and} & \mathrm{Dom}(\mu_i) = \mathrm{Dom}(\mu_{i+1}) = M \\ \textbf{and} & \mathrm{Dom}(w(0)) = P \\ \textbf{and} & \{{}^{\bullet}m \mapsto \mu_i(m) \mid m \in M\} \\ & \cup\, w(0) \cup \{m^{\bullet} \mapsto \mu_{i+1}(m) \mid m \in M\} \models \varphi \end{array}}{(q_i, w, \mu_i) \vdash (q_{i+1}, w', \mu_{i+1})}$$

$$\frac{\begin{array}{l} q_i \xrightarrow{P,\varphi} q_{i+1} \\ \textbf{and}\ \ \mathrm{Dom}(\mu_i) = \mathrm{Dom}(\mu_{i+1}) = M \\ \textbf{and}\ \ \mathrm{Dom}(w(0)) = P \\ \textbf{and}\ \ \{{}^\bullet m \mapsto \mu_i(m) \mid m \in M\} \\ \qquad \cup\, w(0) \cup \{m^\bullet \mapsto \mu_{i+1}(m) \mid m \in M\} \models \varphi \end{array}}{(q_i, w, \mu_i) \vdash (q_{i+1}, w', \mu_{i+1})}$$

$$\frac{\begin{array}{l} q_i \xrightarrow{\emptyset,\varphi} q_{i+1} \\ \textbf{and}\ \ \mathrm{Dom}(\mu_i) = \mathrm{Dom}(\mu_{i+1}) = M \\ \textbf{and}\ \ \{{}^\bullet m \mapsto \mu_i(m) \mid m \in M\} \\ \qquad \cup\, \{m^\bullet \mapsto \mu_{i+1}(m) \mid m \in M\} \models \varphi \end{array}}{(q_i, w, \mu_i) \vdash (q_{i+1}, w, \mu_{i+1})}$$

- **Definition:** g is behavior-preserving if $g(Aut) \approx Aut$
- **Theorem:** $\boldsymbol{a}_1 \approx \boldsymbol{a}_2$ **not implies** $\boldsymbol{a}_1 \otimes \boldsymbol{a} \approx \boldsymbol{a}_2 \otimes \boldsymbol{a}$
    - $\approx$ is *not* a congruence for $\otimes$
    - Witness: Variant of $a(b + c)$ vs. $(ab) + (ac)$
    - Makes reasoning very difficult

- **Definition:** g is behavior-preserving if $g(Aut) \approx Aut$
- **Theorem:** $\boldsymbol{a}_1 \approx \boldsymbol{a}_2$ **not implies** $\boldsymbol{a}_1 \otimes \boldsymbol{a} \approx \boldsymbol{a}_2 \otimes \boldsymbol{a}$
  - $\approx$ is *not* a congruence for $\otimes$
  - Witness: Variant of $a(b + c)$ vs. $(ab) + (ac)$
  - Makes reasoning very difficult

- Define a congruence $\simeq$ that subsumes $\approx$, based on bisimulation [Rut]

$$\frac{}{(Q_1, \mathbf{p}, M, \longrightarrow_1, q_1^0, \mu^0) \simeq_R (Q_2, \mathbf{p}, M, \longrightarrow_2, q_2^0, \mu^0)}$$

$$\frac{R \subseteq Q_1 \times Q_2 \ \textbf{and} \ q_1^0 \, R \, q_2^0}{(Q_1, \mathbf{p}, M, \longrightarrow_1, q_1^0, \mu^0) \simeq_R (Q_2, \mathbf{p}, M, \longrightarrow_2, q_2^0, \mu^0)}$$

$$\frac{\begin{array}{l} R \subseteq Q_1 \times Q_2 \ \textbf{and} \ q_1^0 \, R \, q_2^0 \\ \textbf{and} \ \left[\begin{array}{c} \Big[\left[\begin{array}{l} q_1 \xrightarrow{P,\varphi_1}_1 q_1' \\ \textbf{and} \ q_1 \, R \, q_2 \end{array}\right] \ \textbf{implies} \ \varphi_1 \Rightarrow \bigvee \left\{ \varphi_2 \,\middle|\, \begin{array}{l} q_2 \xrightarrow{P,\varphi_2}_2 q_2' \\ \textbf{and} \ q_1' \, R \, q_2' \end{array} \right\} \Big] \\ \textbf{for all} \ q_1, q_1', q_2, P, \varphi_1 \end{array} \right] \end{array}}{(Q_1, \mathbf{p}, M, \longrightarrow_1, q_1^0, \mu^0) \simeq_R (Q_2, \mathbf{p}, M, \longrightarrow_2, q_2^0, \mu^0)}$$

$$
\begin{array}{c}
\begin{array}{ll}
& R \subseteq Q_1 \times Q_2 \ \textbf{ and } \ q_1^0 \, R \, q_2^0 \\
\textbf{and} & \left[\begin{array}{c} \big[\left[\begin{array}{l} q_1 \xrightarrow{P,\varphi_1}_1 q_1' \\ \textbf{and} \ \ q_1 \, R \, q_2 \end{array}\right] \ \textbf{implies} \ \varphi_1 \Rightarrow \bigvee \left\{ \varphi_2 \ \middle| \ \begin{array}{l} q_2 \xrightarrow{P,\varphi_2}_2 q_2' \\ \textbf{and} \ \ q_1' \, R \, q_2' \end{array} \right\} \big] \\ \textbf{for all} \ \ q_1, q_1', q_2, P, \varphi_1 \end{array}\right] \\
\textbf{and} & \left[\begin{array}{c} \big[\left[\begin{array}{l} q_2 \xrightarrow{P,\varphi_2}_2 q_2' \\ \textbf{and} \ \ q_1 \, R \, q_2 \end{array}\right] \ \textbf{implies} \ \varphi_2 \Rightarrow \bigvee \left\{ \varphi_1 \ \middle| \ \begin{array}{l} q_1 \xrightarrow{P,\varphi_1}_1 q_1' \\ \textbf{and} \ \ q_1' \, R \, q_2' \end{array} \right\} \big] \\ \textbf{for all} \ \ q_1, q_2, q_2', P, \varphi_2 \end{array}\right]
\end{array} \\
\hline
(Q_1, \mathbf{p}, M, \longrightarrow_1, q_1^0, \mu^0) \simeq_R (Q_2, \mathbf{p}, M, \longrightarrow_2, q_2^0, \mu^0)
\end{array}
$$

- **Theorem:** $\simeq \subseteq \approx$
- **Theorem:**

$$\left[\boldsymbol{a}_1 \simeq \boldsymbol{a}_2 \ \ \textbf{and} \ \ \boldsymbol{a}_3 \simeq \boldsymbol{a}_4\right] \ \ \textbf{implies} \ \ \boldsymbol{a}_1 \otimes \boldsymbol{a}_3 \simeq \boldsymbol{a}_2 \otimes \boldsymbol{a}_4$$

- **Theorem:** $\simeq \, \subseteq \, \approx$
- **Theorem:**

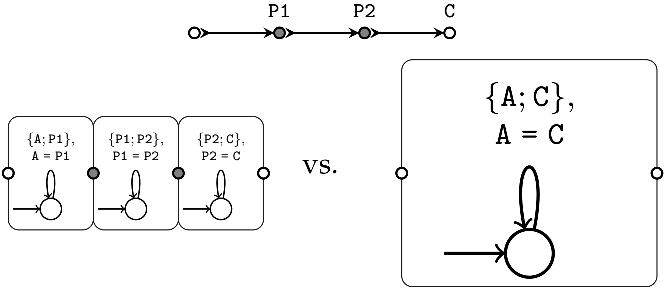
$$\left[\boldsymbol{a}_1 \simeq \boldsymbol{a}_2 \ \textbf{ and } \ \boldsymbol{a}_3 \simeq \boldsymbol{a}_4\right] \ \textbf{ implies } \ \boldsymbol{a}_1 \otimes \boldsymbol{a}_3 \simeq \boldsymbol{a}_2 \otimes \boldsymbol{a}_4$$

- **Corollary:** g is behavior-preserving if $g(Aut) \simeq Aut$

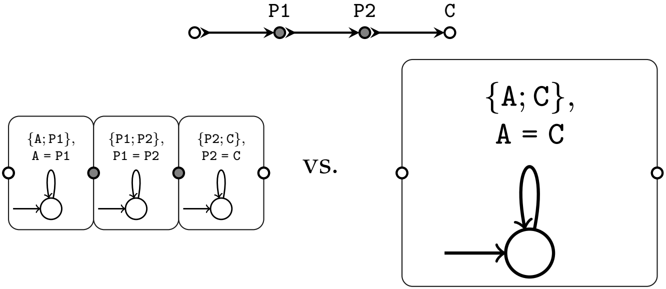
Optimization I

- **Observation:** Centralized approach suffers from compile-time state-space explosion and run-time oversequentialization
- **Claim:** Distributed approach suffers from run-time overparallelization

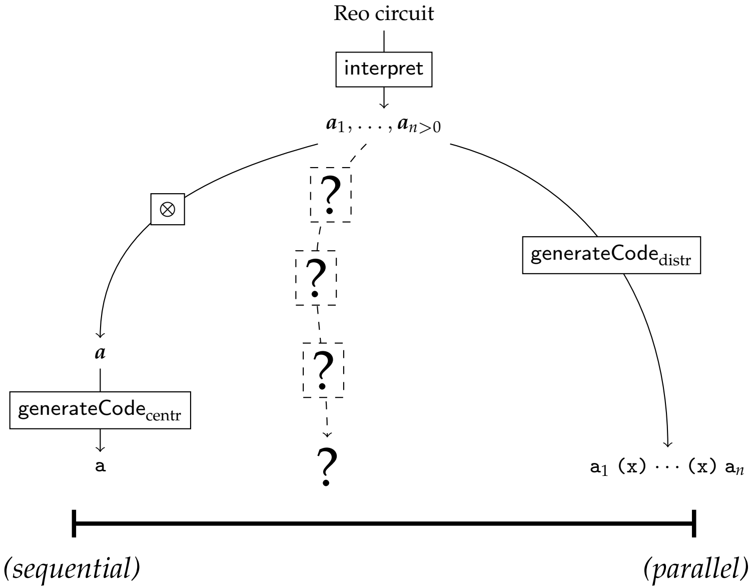
- **Observation:** Centralized approach suffers from compile-time state-space explosion and run-time oversequentialization
- **Claim:** Distributed approach suffers from run-time overparallelization



- **Observation:** Centralized approach suffers from compile-time state-space explosion and run-time oversequentialization
- **Claim:** Distributed approach suffers from run-time overparallelization



- **Goal:** Find a middle ground between these approaches



Reo circuit

interpret

$\boldsymbol{a}_1, \ldots, \boldsymbol{a}_{n>0}$

$\otimes$

?

?

?

$\boldsymbol{a}$

generateCode$_{\text{centr}}$

a

?

generateCode$_{\text{distr}}$

$\mathtt{a}_1 \ \mathtt{(x)} \cdots \mathtt{(x)} \ \mathtt{a}_n$

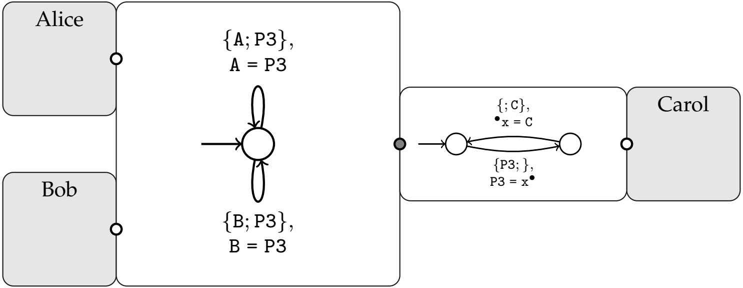
(sequential)

(parallel)

Hybrid compilation:

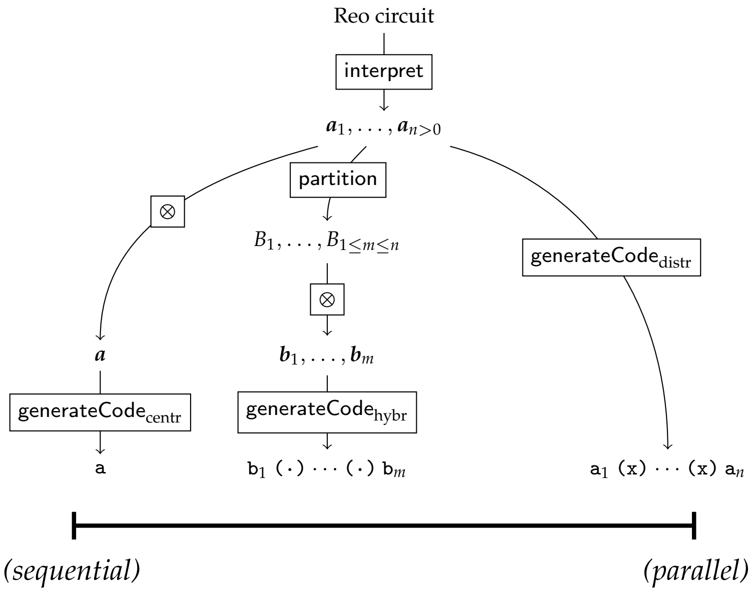
1. Find a "small" automaton for the *local* behavior of every node/channel in the input circuit
2. Partition the resulting set of small automata into subsets
3. For every resulting subset:
    - Multiply its small automata into a "medium" automaton for the *regional* behavior of a region of the input circuit
4. Translate the resulting medium automata into Java code for their run-time execution, *using a consensus algorithm*

Running example:



Hybrid execution:

- Similar to distributed execution, *except*:
    - Fewer protocol threads to reach consensus with
    - “Cheaper” consensus when partitioning *carefully*



Reo circuit

interpret

$\boldsymbol{a}_1, \ldots, \boldsymbol{a}_{n>0}$

$\otimes$

partition

$B_1, \ldots, B_{1 \leq m \leq n}$

$\otimes$

generateCode$_{\text{distr}}$

$\boldsymbol{a}$

$\boldsymbol{b}_1, \ldots, \boldsymbol{b}_m$

generateCode$_{\text{centr}}$

generateCode$_{\text{hybr}}$

`a`

$\texttt{b}_1$ `(.)` $\cdots$ `(.)` $\texttt{b}_m$

$\texttt{a}_1$ `(x)` $\cdots$ `(x)` $\texttt{a}_n$

(sequential)

(parallel)

- The compiler partitions such that consensus is “cheap”
- (See demo)

- The compiler partitions such that consensus is “cheap”
- (See demo)

- **Main question:** How?
- **Helper question:** What makes consensus in the distributed approach “expensive”? **Propagation of synchrony**
- (Anecdote)

- The compiler partitions such that consensus is “cheap”
- (See demo)

- **Main question:** How?
- **Helper question:** What makes consensus in the distributed approach “expensive”? **Propagation of synchrony**
- (Anecdote)

- **Definition:**
  - Expensive consensus supports propagation of synchrony
  - Cheap consensus does *not*
- (Continue anecdote)

- The compiler partitions such that ~~consensus is "cheap"~~ the resulting medium automata require no propag. of sync.
- (See demo)

- **Main question:** How?
- **Helper question:** What makes consensus in the distributed approach "expensive"? **Propagation of synchrony**
- (Anecdote)

- **Definition:**
  - Expensive consensus supports propagation of synchrony
  - Cheap consensus does *not*
- (Continue anecdote)

- An automaton with transition relation $\longrightarrow$ requires propagation of synchrony iff:

  $$\left[q \xrightarrow{P,\varphi} q' \;\; \textbf{and} \;\; |P| > 1\right] \;\; \textbf{for some} \;\; q, q', P, \varphi$$

  (I.e., it has a transition to synchronize two or more ports)

- An automaton with transition relation $\longrightarrow$ requires propagation of synchrony iff:

  $$\left[q \xrightarrow{P,\varphi} q' \textbf{ and } |P| > 1\right] \textbf{ for some } q, q', P, \varphi$$

  (I.e., it has a transition to synchronize two or more ports)

- Let R denote an auxiliary relation such that $\boldsymbol{a}\ R\ \boldsymbol{a}'$ iff:
  - $\boldsymbol{a}$ and $\boldsymbol{a}'$ require propagation of synchrony
  - $\boldsymbol{a}$ and $\boldsymbol{a}'$ share at least one port ("neighbors")
- (R is symmetric)

- An automaton with transition relation $\longrightarrow$ requires propagation of synchrony iff:

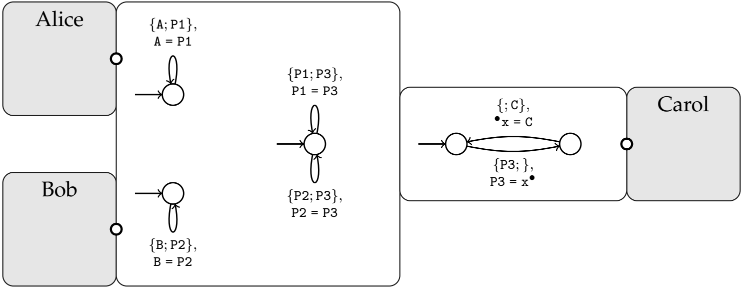
$$\left[q \xrightarrow{P,\varphi} q' \ \ \textbf{and} \ \ |P| > 1\right] \ \ \textbf{for some} \ \ q, q', P, \varphi$$

(I.e., it has a transition to synchronize two or more ports)

- Let R denote an auxiliary relation such that $\boldsymbol{a} \; R \; \boldsymbol{a}'$ iff:
  - $\boldsymbol{a}$ and $\boldsymbol{a}'$ require propagation of synchrony
  - $\boldsymbol{a}$ and $\boldsymbol{a}'$ share at least one port ("neighbors")
- (R is symmetric)

- Partition criterion: $\boldsymbol{a}, \boldsymbol{a}'$ are in the same part if $\boldsymbol{a} \; R \; \boldsymbol{a}'$
- Partition definition: $\{\{\boldsymbol{a}' \mid \boldsymbol{a} \; R^* \; \boldsymbol{a}'\} \mid \boldsymbol{a} \in \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n\}\}$

Running example:



- What if we use the distributed approach with cheap consensus? **Unsound circuit execution**
- How to prove the hybrid approach sound?

- What if we use the distributed approach with cheap consensus? **Unsound circuit execution**
- How to prove the hybrid approach sound?

- **Observation:**
  - Expensive consensus computes $\otimes$ *at run-time*
  - Equivalently: $\otimes$ models expensive consensus

- What if we use the distributed approach with cheap consensus? **Unsound circuit execution**
- How to prove the hybrid approach sound?

- **Observation:**
  - Expensive consensus computes $\otimes$ *at run-time*
  - Equivalently: $\otimes$ models expensive consensus

- Proof steps:
  1. Define *another* multiplication $\odot$ to model cheap consensus
  2. Establish that substituting $\odot$ for $\otimes$ is behavior-preserving

  (Key: model consensus algorithms with multiplications)

Transition rule for $\otimes$:

$$\frac{\begin{array}{l} q_1 \xrightarrow{P_1,\varphi_1}_1 q_1' \ \ \textbf{and} \ \ q_2 \xrightarrow{P_2,\varphi_2}_2 q_2' \\ \textbf{and} \ \ (P_1^{\text{in}} \cup P_1^{\text{out}}) \cap P_2 = (P_2^{\text{in}} \cup P_2^{\text{out}}) \cap P_1 \end{array}}{q \xrightarrow{P_1 \triangle P_2, \exists (P_1 \cap P_2). \varphi_1 \wedge \varphi_2} q'}$$

Transition rule for $\otimes$:

$$\frac{\begin{array}{l} q_1 \xrightarrow{P_1,\varphi_1}_1 q_1' \ \ \textbf{and} \ \ q_2 \xrightarrow{P_2,\varphi_2}_2 q_2' \\ \textbf{and} \ \ (P_1^{\text{in}} \cup P_1^{\text{out}}) \cap P_2 = (P_2^{\text{in}} \cup P_2^{\text{out}}) \cap P_1 \end{array}}{q \xrightarrow{P_1 \triangle P_2, \exists (P_1 \cap P_2). \varphi_1 \wedge \varphi_2} q'}$$

Transition rule for $\odot$:

$$\frac{\begin{array}{l} q_1 \xrightarrow{P_1,\varphi_1}_1 q_1' \ \ \textbf{and} \ \ q_2 \xrightarrow{P_2,\varphi_2}_2 q_2' \\ \textbf{and} \ \ \left[P_1 \cap P_2 = \emptyset \ \ \textbf{or} \ \ P_1 \subseteq P_2 \ \ \textbf{or} \ \ P_2 \subseteq P_1 \right] \end{array}}{q \xrightarrow{P_1 \triangle P_2, \exists (P_1 \cap P_2). \varphi_1 \wedge \varphi_2} q'}$$

- **Theorem:**

  $$\begin{gathered}\bigl[P_1 \cap P_2 = \emptyset \ \ \textbf{or} \ \ P_1 \subseteq P_2 \ \ \textbf{or} \ \ P_2 \subseteq P_1\bigr] \\ \textbf{implies} \ \ (P_1^{\text{in}} \cup P_1^{\text{out}}) \cap P_2 = (P_2^{\text{in}} \cup P_2^{\text{out}}) \cap P_1\end{gathered}$$

  - Expensive consensus can safely be used instead of cheap consensus
  - Supporting propagation of synchrony is more powerful than *not* supporting propagation of synchrony

- **Theorem:**

$$\left[\begin{array}{l} \boldsymbol{a}_1 \text{ or } \boldsymbol{a}_2 \text{ requires no propagation of synchrony} \\ \textbf{and}\ \ (P_1^{\text{in}} \cup P_1^{\text{out}}) \cap P_2 = (P_2^{\text{in}} \cup P_2^{\text{out}}) \cap P_1 \end{array}\right]$$

$$\textbf{implies}\ \ \left[P_1 \cap P_2 = \emptyset\ \ \textbf{or}\ \ P_1 \subseteq P_2\ \ \textbf{or}\ \ P_2 \subseteq P_1\right]$$

- **Theorem:**

$$\begin{bmatrix} \boldsymbol{a}_1 \text{ or } \boldsymbol{a}_2 \text{ requires no propagation of synchrony} \\ \textbf{and} \;\; (P_1^{\text{in}} \cup P_1^{\text{out}}) \cap P_2 = (P_2^{\text{in}} \cup P_2^{\text{out}}) \cap P_1 \end{bmatrix}$$

$$\textbf{implies} \;\; \left[P_1 \cap P_2 = \emptyset \;\; \textbf{or} \;\; P_1 \subseteq P_2 \;\; \textbf{or} \;\; P_2 \subseteq P_1\right]$$

- **Corollary:**

$$\left[\boldsymbol{a}_1 \text{ or } \boldsymbol{a}_2 \text{ requires no propagation of synchrony}\right]$$

$$\textbf{implies} \;\; \boldsymbol{a}_1 \otimes \boldsymbol{a}_2 \simeq \boldsymbol{a}_1 \odot \boldsymbol{a}_2$$

- **Theorem:**

$$\begin{bmatrix}\boldsymbol{a}_1 \text{ or } \boldsymbol{a}_2 \text{ requires no propagation of synchrony} \\ \textbf{and}\ \ (P_1^{\text{in}} \cup P_1^{\text{out}}) \cap P_2 = (P_2^{\text{in}} \cup P_2^{\text{out}}) \cap P_1\end{bmatrix}$$

$$\textbf{implies}\ \ \left[P_1 \cap P_2 = \emptyset\ \ \textbf{or}\ \ P_1 \subseteq P_2\ \ \textbf{or}\ \ P_2 \subseteq P_1\right]$$

- **Corollary:**

$$\left[\boldsymbol{a}_1 \text{ or } \boldsymbol{a}_2 \text{ requires no propagation of synchrony}\right]$$

$$\textbf{implies}\ \ \boldsymbol{a}_1 \otimes \boldsymbol{a}_2 \simeq \boldsymbol{a}_1 \odot \boldsymbol{a}_2$$

- **Corollary:**

$$\begin{bmatrix}\{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n\} \text{ are small automata} \\ \textbf{and}\ \ \{B_1, \ldots, B_m\} \text{ is a partition}\end{bmatrix}\ \textbf{implies}$$

$$\boldsymbol{a}_1 \otimes \cdots \otimes \boldsymbol{a}_n \simeq (\bigotimes B_1) \odot \cdots \odot (\bigotimes B_m)$$

- Previous observations:
  - Centralized approach suffers from compile-time state-space explosion and run-time oversequentialization
  - Distributed approach suffers from run-time overparallelization
- Simulating automata that require propagation of synchrony generally amounts to *useless parallelism*
- The hybrid approach tries to maximize *useful parallelism*

Optimization II

Centralized execution:

1. Protocol thread α awakes to handle event from process thread on port p
2. For all transitions $q^{\text{curr}} \xrightarrow{P,\varphi} q'$ such that $p \in P$:
    1. α checks synchronization constraint P:
        - Are "neighboring" process threads "behind" public ports in P ready for data-flow through those ports?
        - ~~Are "neighboring" protocol threads "behind" private ports in P ready for data-flow through those ports? *(New events!)*~~
    2. α checks data constraint φ:
        - Do data to be exchanged through ports in P satisfy φ?
    3. ~~$\alpha_\imath$ commits to make transition, and informs θ~~
    4. α ~~awaits confirmation from θ, and~~ makes transition
    5. α breaks the loop
3. α goes back to sleep

- Constraint solving for φ, with free variables $x_1, \ldots, x_k$:
  1. "Guess" a solution $\sigma = \{x_1 \mapsto d_1, \ldots, x_k \mapsto d_k\}$
  2. Check $\sigma \models \varphi$

     On failure, go back to the previous step and guess again

- Constraint solving for φ, with free variables $x_1, \ldots, x_k$:
  1. "Guess" a solution $\sigma = \{x_1 \mapsto d_1, \ldots, x_k \mapsto d_k\}$
  2. Check $\sigma \models \varphi$
     On failure, go back to the previous step and guess again

- **Fortunately:** Much more advanced techniques exist
- **Unfortunately:** Constraint solving over finite domains is NP-complete

- Constraint solving for φ, with free variables $x_1, \ldots, x_k$:
  1. "Guess" a solution $\sigma = \{x_1 \mapsto d_1, \ldots, x_k \mapsto d_k\}$
  2. Check $\sigma \models \varphi$
     On failure, go back to the previous step and guess again

- **Fortunately:** Much more advanced techniques exist
- **Unfortunately:** Constraint solving over finite domains is NP-complete

- Checking data constraints, for *every* transition, for *every* event, is an expensive sequential bottleneck
- **Challenge:** How to speed-up checking data constraints?

What a programmer would do:

$$q \xrightarrow{\{\texttt{A;B}\},\texttt{A=B}} q'$$

What a programmer would do:

$$\begin{gathered} q \xrightarrow{\{\texttt{A;B}\},\texttt{A=B}} q' \\ \Downarrow \\ \texttt{B := A} \end{gathered}$$

What a programmer would do:

$$q \xrightarrow{\{\mathtt{A;B,C}\},\mathtt{A=B}\wedge\mathtt{A=C}} q'$$

What a programmer would do:

$$q \xrightarrow{\{\mathtt{A};\mathtt{B},\mathtt{C}\},\mathtt{A}=\mathtt{B}\wedge\mathtt{A}=\mathtt{C}} q'$$

$$\Downarrow$$

```
B := A ; C := A
```

What a programmer would do:

$$q \xrightarrow{\{\texttt{A},\texttt{B};\texttt{C}\},\texttt{A=B}\wedge\texttt{A=C}} q'$$

What a programmer would do:

$$q \xrightarrow{\{\texttt{A},\texttt{B};\texttt{C}\},\texttt{A=B}\wedge\texttt{A=C}} q'$$

$$\Downarrow$$

$$\texttt{if A = B then C := A}$$

Idea:

- At compile-time: Translate every *declarative* data constraint φ into an *imperative* **data command** $\dagger(\varphi)$ (cf. programmers)
- At run-time: Execute $\dagger(\varphi)$ instead of solving φ
- (See demo)

Syntax of data commands

$$P \quad ::= \quad \texttt{skip} \mid x \texttt{ := } t \mid \texttt{if } \varphi \texttt{ -> } P \mid P \texttt{ ; } P$$

Semantics of data commands

- A **configuration** (P, σ) is a pair of a data command P and a state σ to execute P in

Semantics of data commands

- A **configuration** (P, σ) is a pair of a data command P and a state σ to execute P in

- $\Longrightarrow$ denotes the transition relation on configurations

$$\frac{}{(\texttt{skip}, \sigma) \Longrightarrow (\varepsilon, \sigma)} \qquad \frac{}{(x \texttt{ := } t, \sigma) \Longrightarrow (\varepsilon, \sigma[x := \mathsf{eval}_\sigma(t)])}$$

$$\frac{\sigma \models \varphi}{(\texttt{if } \varphi \texttt{ -> } P, \sigma) \Longrightarrow (P, \sigma)} \qquad \frac{\sigma \not\models \varphi}{(\texttt{if } \varphi \texttt{ -> } P, \sigma) \Longrightarrow (\varepsilon, \mathsf{fail})}$$

$$\frac{(P, \sigma) \Longrightarrow (P', \sigma')}{(P \ ; \ P'', \sigma) \Longrightarrow (P' \ ; \ P'', \sigma')}$$

Semantics of data commands

- Partial correctness semantics

$$\mathcal{M}(P, \Sigma) = \{\sigma' \mid \sigma \in \Sigma \ \textbf{and} \ (P, \sigma) \Longrightarrow^* (\varepsilon, \sigma')\}$$

Semantics of data commands

- Partial correctness semantics

$$\mathcal{M}(P,\Sigma) = \{\sigma' \mid \sigma \in \Sigma \ \ \textbf{and} \ \ (P,\sigma) \Longrightarrow^* (\varepsilon,\sigma')\}$$

- Total correctness semantics

$$\begin{aligned}\mathcal{M}_{\text{tot}}(P,\Sigma) = {} & \mathcal{M}(P,\Sigma) \\ & \cup \{\mathsf{fail} \mid \sigma \in \Sigma \ \ \textbf{and} \ \ (P,\sigma) \Longrightarrow^* (\varepsilon,\mathsf{fail})\}\end{aligned}$$

Find a translation † such that, for every data constraint φ:

Find a translation † such that, for every data constraint φ:

$$\underbrace{\mathcal{M}(\dagger(\varphi), \Sigma) \subseteq \{\sigma \mid \sigma \models \varphi\}}_{\text{soundness}}$$

where $\Sigma = \mathbb{X} \rightharpoonup \mathbb{D}$

Find a translation † such that, for every data constraint φ:

$$\underbrace{\mathcal{M}(\dagger(\varphi), \Sigma) \subseteq \{\sigma \mid \sigma \models \varphi\}}_{\text{soundness}} \quad \textbf{and} \quad \underbrace{\mathcal{M}_{\text{tot}}(\dagger(\varphi), \Sigma) \subseteq \Sigma}_{\text{completeness}}$$

where $\Sigma = \mathbb{X} \rightharpoonup \mathbb{D}$

Syntax of data constraints′:

$$\begin{array}{lcll}
t & ::= & x \mid d \mid f(t, \ldots, t) & \text{(data terms)} \\
a & ::= & \top \mid \bot \mid t = t \mid \mathtt{Keep}(M) \mid R(t, \ldots, t) & \text{(data atoms)} \\
\ell & ::= & a \mid \neg a & \text{(data literals)} \\
\varphi & ::= & \ell \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \exists x . \varphi & \text{(data constraints)}
\end{array}$$

Semantics of data constraints′:

$$\begin{array}{lcl}
\sigma \models t_1 = t_2 & \textbf{iff} & \mathsf{eval}_\sigma(t_1) = \mathsf{eval}_\sigma(x_2) \\
\sigma \models \mathtt{Keep}(M) & \textbf{iff} & \sigma \models {}^\bullet m = m^\bullet \ \textbf{ for all } \ m \in M \\
\sigma \models R(t_1, \ldots, t_k) & \textbf{iff} & (\mathsf{eval}_\sigma(t_1), \ldots, \mathsf{eval}_\sigma(t_k)) \in R
\end{array}$$

where eval : $(\mathbb{X} \rightharpoonup \mathbb{D}) \times \{t \mid t \text{ is a data term}\} \rightarrow \mathbb{D}$

- **Theorem:** Every automaton can be translated to a congruent automaton with only data constraints of the form $\ell_1 \wedge \cdots \wedge \ell_k$
- Henceforth, assume all data constraints to be in this form

Question: How to construct P that computes σ that satisfies φ?

Observations:

- Some literals in of the form $x = t$ (or $t = x$) can be turned into an assignment statement:

$$x := t$$

- Other literals ℓ in φ need to be translated into a guarded failure statement:

$$\texttt{if } \ell \texttt{ -> skip}$$

- The order in which assignments and guarded failures follow each other crucially matters

✓ $$\begin{aligned} \dagger(\texttt{C = add}(\texttt{A}, \texttt{B}) \land \neg\texttt{Odd}(\texttt{C})) &= \texttt{C := add}(\texttt{A}, \texttt{B})\ \texttt{;} \\ &\ \texttt{if } \neg\texttt{Odd}(\texttt{C})\ \texttt{-> skip} \end{aligned}$$

✗ $$\begin{aligned} \dagger(\texttt{C = add}(\texttt{A}, \texttt{B}) \land \neg\texttt{Odd}(\texttt{C})) &= \texttt{if } \neg\texttt{Odd}(\texttt{C})\ \texttt{-> skip ;} \\ &\ \texttt{C := add}(\texttt{A}, \texttt{B}) \end{aligned}$$

Approach:

1. Extract a linear precedence relation $\sqsubset$ on literals in φ from a data-flow graph for φ
2. Translate literals to statements according to $\sqsubset$

Explanation by example:

$$\varphi = \texttt{A = 0} \wedge \texttt{B = 1} \wedge \texttt{C = add}(\texttt{A}, \texttt{B}) \wedge \texttt{C = D} \wedge \texttt{C = E} \wedge \neg\texttt{Odd}(\texttt{D})$$

$\neg\mathtt{Odd}(\mathtt{D})$

$\mathtt{A} = \mathtt{0}$

$\mathtt{C} = \mathtt{D}$

$\mathtt{C} = \mathtt{add}(\mathtt{A}, \mathtt{B})$

$\mathtt{B} = \mathtt{1}$

$\mathtt{C} = \mathtt{E}$

All literals in φ are vertices

$\mathtt{0 = A}$ $\mathtt{1 = B}$ $\neg\mathtt{Odd}(\mathtt{D})$

$\mathtt{A = 0}$ $\mathtt{C = D}$ $\mathtt{D = C}$

$\mathtt{C = add}(\mathtt{A}, \mathtt{B})$

$\mathtt{B = 1}$ $\mathtt{C = E}$ $\mathtt{E = C}$

$\mathtt{add}(\mathtt{A}, \mathtt{B}) \mathtt{= C}$

Also “symmetric equalities” are vertices

0 = A   1 = B   ¬Odd(D)

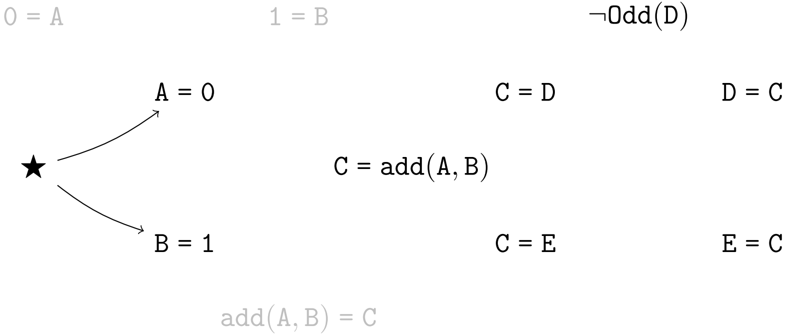
A = 0   C = D   D = C

★   C = add(A, B)

B = 1   C = E   E = C

add(A, B) = C

Also ★ is a vertex



0 = A

1 = B

¬Odd(D)

A = 0

C = D

D = C

★

C = add(A, B)

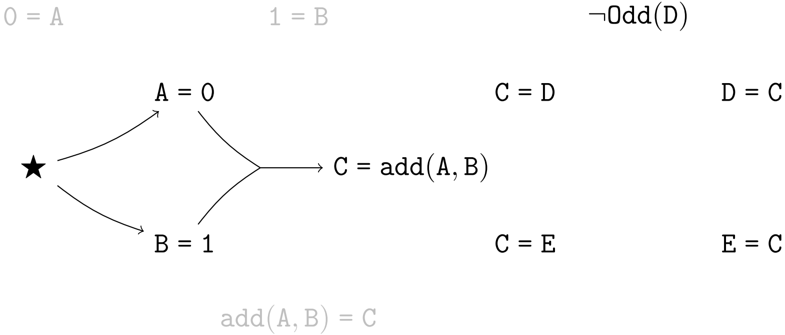
B = 1

C = E

E = C

add(A, B) = C

Hyperarcs represent dependencies among literals



0 = A

1 = B

¬Odd(D)

A = 0

C = D

D = C

★

C = add(A, B)

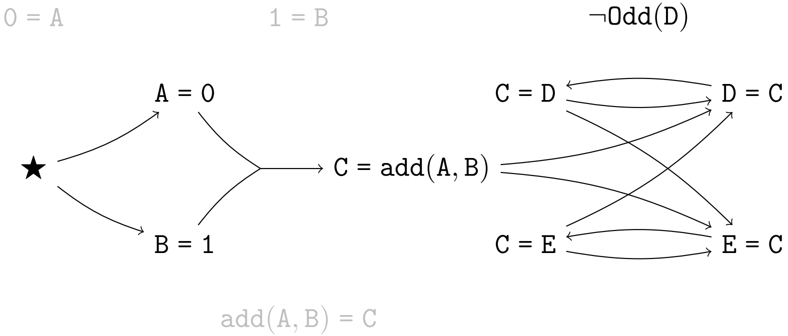
B = 1

C = E

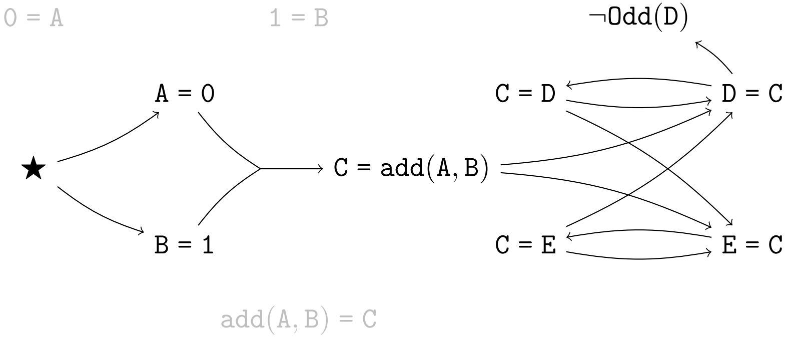
E = C

add(A, B) = C

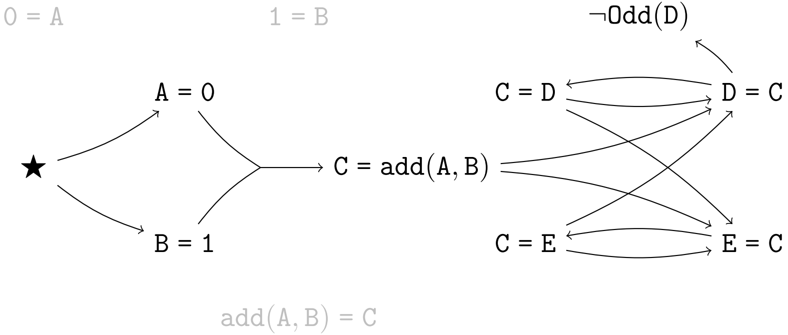
Hyperarcs represent dependencies among literals



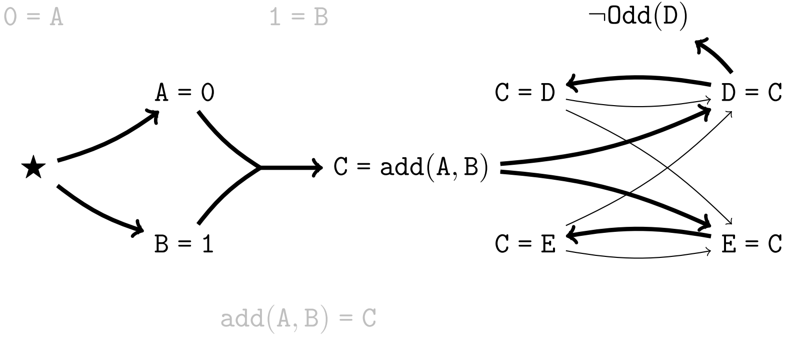
Hyperarcs represent dependencies among literals



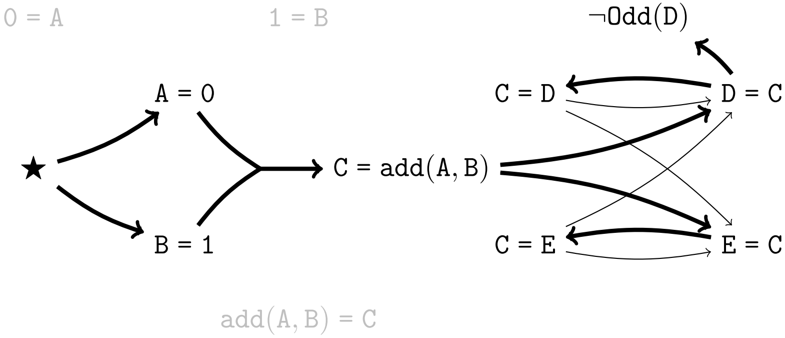
Hyperarcs represent dependencies among literals



(Many hyperarcs missing from this figure)



Compute an **arborescence** on the hypergraph



(An arborescence not always exists...)

$\mathtt{A = 0}$

$\sqsubset \mathtt{B = 1}$

$\sqsubset \mathtt{C = add(A, B)}$

$\sqsubset \mathtt{D = C}$

$\sqsubset \mathtt{C = D}$

$\sqsubset \neg\mathtt{Odd(D)}$

$\sqsubset \mathtt{E = C}$

$\sqsubset \mathtt{C = E}$

The arborescence induces a strict total order

```
A = 0
 ⊏ B = 1
 ⊏ C = add(A, B)
 ⊏ D = C
 ⊏ C = D
 ⊏ ¬Odd(D)
 ⊏ E = C
 ⊏ C = E
```

```
A := 0 ;
```

A simple algorithm iterates over this order

`A = 0`	`A := 0 ;`
$\sqsubset$ `B = 1`	`B := 1 ;`
$\sqsubset$ `C = add(A, B)`	
$\sqsubset$ `D = C`	
$\sqsubset$ `C = D`	
$\sqsubset$ $\neg$`Odd(D)`	
$\sqsubset$ `E = C`	
$\sqsubset$ `C = E`	

A simple algorithm iterates over this order

```
A = 0
 ⊏ B = 1
 ⊏ C = add(A, B)
 ⊏ D = C
 ⊏ C = D
 ⊏ ¬Odd(D)
 ⊏ E = C
 ⊏ C = E
```

```
A := 0 ;
B := 1 ;
C := add(A, B) ;
```

A simple algorithm iterates over this order

$$\begin{array}{ll}
\mathtt{A} = \mathtt{0} & \mathtt{A} := \mathtt{0}\ ; \\
\sqsubset \mathtt{B} = \mathtt{1} & \mathtt{B} := \mathtt{1}\ ; \\
\sqsubset \mathtt{C} = \mathtt{add}(\mathtt{A}, \mathtt{B}) & \mathtt{C} := \mathtt{add}(\mathtt{A}, \mathtt{B})\ ; \\
\sqsubset \mathtt{D} = \mathtt{C} & \mathtt{D} := \mathtt{C}\ ; \\
\sqsubset \mathtt{C} = \mathtt{D} & \\
\sqsubset \neg\mathtt{Odd}(\mathtt{D}) & \\
\sqsubset \mathtt{E} = \mathtt{C} & \\
\sqsubset \mathtt{C} = \mathtt{E} &
\end{array}$$

A simple algorithm iterates over this order

```
A = 0
 ⊏ B = 1
 ⊏ C = add(A, B)
 ⊏ D = C
 ⊏ C = D
 ⊏ ¬Odd(D)
 ⊏ E = C
 ⊏ C = E
```

```
A := 0 ;
B := 1 ;
C := add(A, B) ;
D := C ;
if C = D -> skip ;
```

A simple algorithm iterates over this order

```
A = 0
 ⊏ B = 1
 ⊏ C = add(A, B)
 ⊏ D = C
 ⊏ C = D
 ⊏ ¬Odd(D)
 ⊏ E = C
 ⊏ C = E
```

```
A := 0 ;
B := 1 ;
C := add(A, B) ;
D := C ;
if C = D -> skip ;
if ¬Odd(D) -> skip ;
```

A simple algorithm iterates over this order

```
A = 0
 ⊏ B = 1
 ⊏ C = add(A, B)
 ⊏ D = C
 ⊏ C = D
 ⊏ ¬Odd(D)
 ⊏ E = C
 ⊏ C = E
```

```
A := 0 ;
B := 1 ;
C := add(A, B) ;
D := C ;
if C = D -> skip ;
if ¬Odd(D) -> skip ;
E := C ;
```

A simple algorithm iterates over this order

```
A = 0                    A := 0 ;
 ⊏ B = 1                 B := 1 ;
 ⊏ C = add(A, B)         C := add(A, B) ;
 ⊏ D = C                 D := C ;
 ⊏ C = D                 if C = D -> skip ;
 ⊏ ¬Odd(D)               if ¬Odd(D) -> skip ;
 ⊏ E = C                 E := C ;
 ⊏ C = E                 if C = E -> skip ;
```

A simple algorithm iterates over this order

Find a translation † such that, for every data constraint φ:

$$\underbrace{\mathcal{M}(\dagger(\varphi), \Sigma) \subseteq \{\sigma \mid \sigma \models \varphi\}}_{\text{soundness}} \quad \textbf{and} \quad \underbrace{\mathcal{M}_{\text{tot}}(\dagger(\varphi), \Sigma) \subseteq \Sigma}_{\text{completeness}}$$

where $\Sigma = \mathbb{X} \to \mathbb{D}$.

- Proof using Hoare logic
- **Theorem:** If the hypergraph for a satisfiable data constraint φ has an arborescence, the algorithm yields a data command P such that:

$$\underbrace{\vdash_{\text{part}} \{\top\}\ P\ \{\varphi\}}_{\text{soundness}} \quad \textbf{and} \quad \underbrace{\vdash_{\text{tot}} \{\top\}\ P\ \{\top\}}_{\text{completeness}}$$

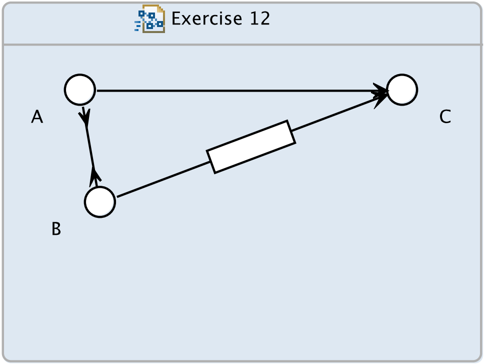
  where $\vdash_{\text{part}}$ and $\vdash_{\text{tot}}$ are proof systems for partial and total correctness.

- **Theorem:** Replacing a data constraint with an equivalent data constraint yields a congruent automaton
- **Corollary:** $\boldsymbol{a} \simeq (\!|\boldsymbol{a}|\!)$ (where $(\!|\cdot|\!)$) denotes commandification)

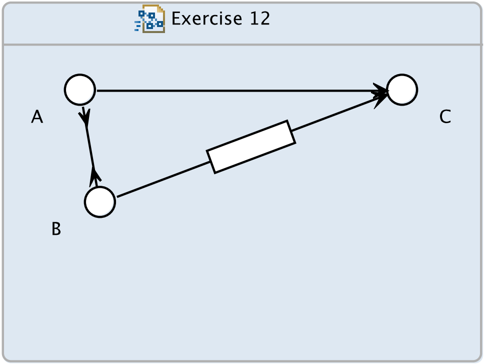
Constraint solving folklore according to Apt:

> *"If domain specific methods are available they should be applied instead of the general methods"*

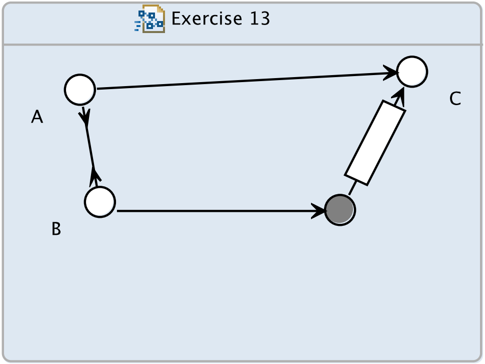
More exercises



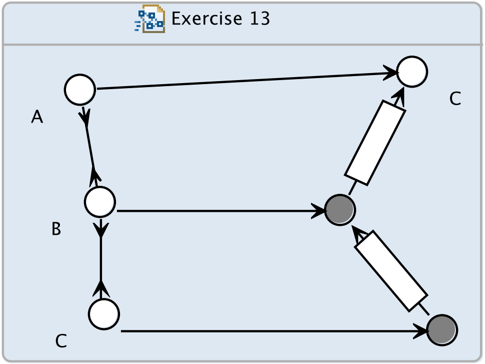
Exercise 12: Describe the protocol specified by this circuit (in natural language or as an automaton)



Exercise 13: Extend the circuit from Exercise 12 for a protocol among 3 producers

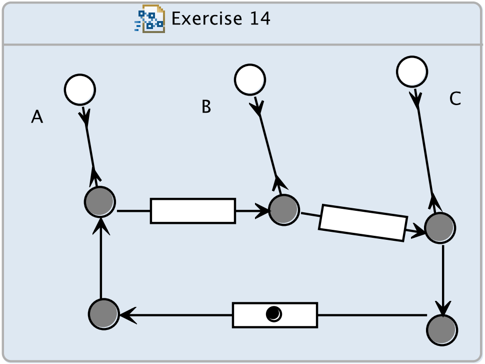


Exercise 13: Extend the circuit from Exercise 12 for a protocol among 3 producers



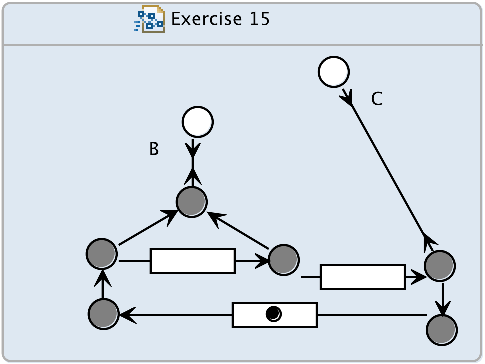
Exercise 13: Extend the circuit from Exercise 12 for a protocol among 3 producers

Exercise 14: Design a circuit for a protocol among 3 producers that send messages in sequence (all messages are lost)

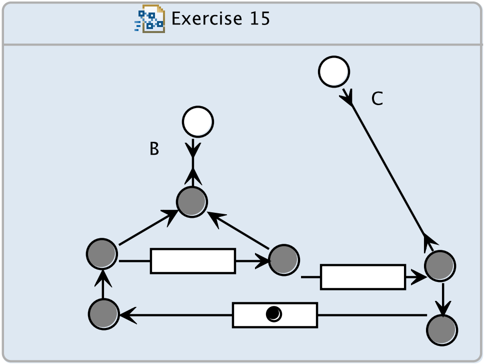


Exercise 14: Design a circuit for a protocol among 3 producers that send messages in sequence (all messages are lost)

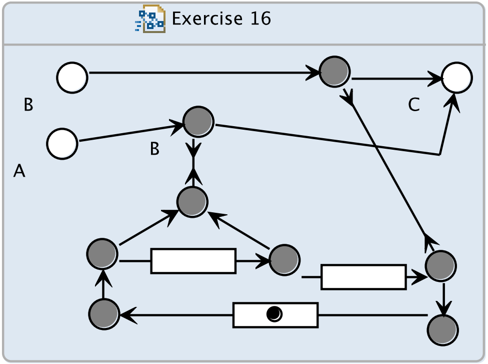
Exercise 15: Design a circuit for a protocol among 2 producers that send messages in sequence, where the first producer sends two messages (all messages are lost)



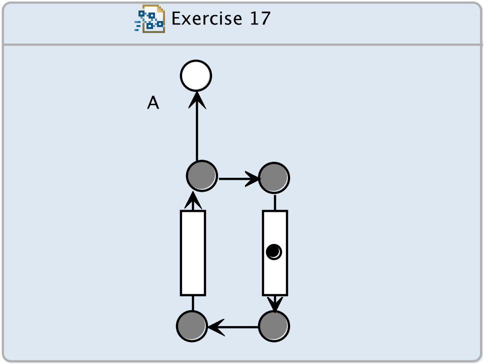
Exercise 15: Design a circuit for a protocol among 2 producers that send messages in sequence, where the first producer sends two messages (all messages are lost)



Exercise 16: Extend the circuit from Exercise 15 such that a consumer receives all messages

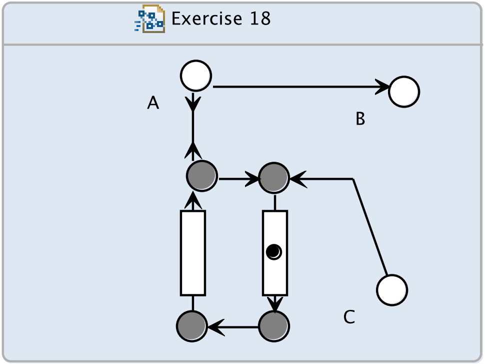


Exercise 16: Extend the circuit from Exercise 15 such that a consumer receives all messages

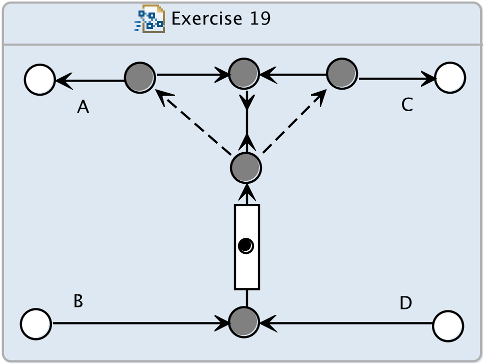


Exercise 17: Describe the protocol specified by this circuit (in natural language or as an automaton)

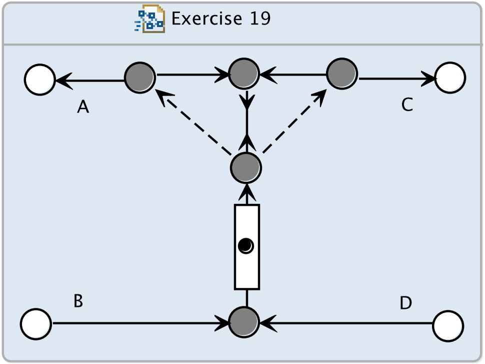
Exercise 18: Design a circuit for a protocol among a producer, a consumer, and a regulator, where the producer sends messages to the consumer until the regulator sends a signal



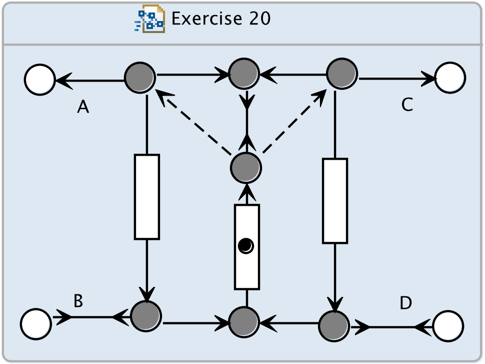
Exercise 18: Design a circuit for a protocol among a producer, a consumer, and a regulator, where the producer sends messages to the consumer until the regulator sends a signal



Exercise 19: Describe the protocol specified by this circuit (in natural language or as an automaton)



Exercise 20: Extend the circuit from Exercise 19 to a lock



Exercise 20: Extend the circuit from Exercise 19 to a lock

Summary:

- Basic compilation: Distributed approach and centralized approach
- Optimizations:
  - Hybrid approach (*middle ground* between sequentiality and parallelism)
  - Translating data constraints to data commands

  More optimizations exist! (E.g., automatic queue inference)
- Correctness criteria: *preservation of behavior* (language equivalence through congruence)

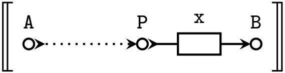
Are we happy with LossySync?

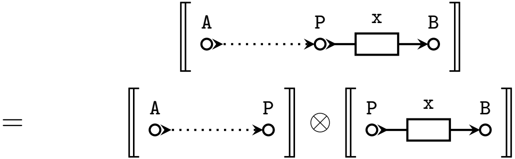
(Maybe not..!)

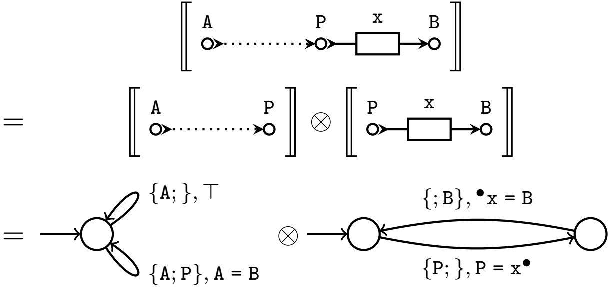
- LossySync loses data nondeterministically
- It makes more sense for LossySync to lose data *only if* they have nowhere to go

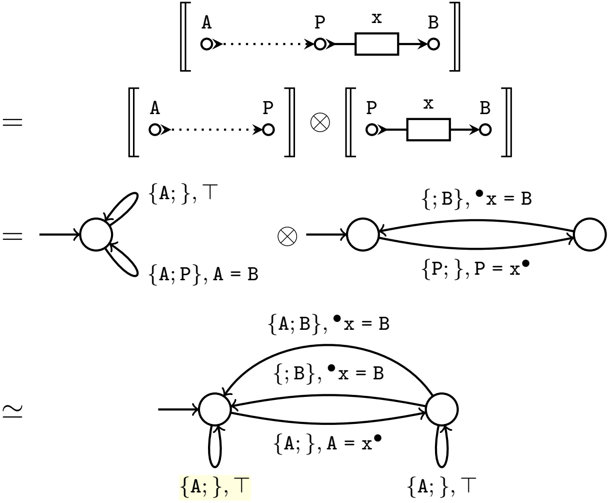
- LossySync loses data nondeterministically
- It makes more sense for LossySync to lose data *only if* they have nowhere to go

- LossySync should exhibit context-sensitivity
- Generally, context-sensitivity means that behavior depends on whether the "context" is ready to accept/offer data


A
P
x
B


A
P
x
B
=
A
P
⊗
P
x
B


A
P
x
B
=
A
P
⊗
P
x
B
=
{A; }, ⊤
{A; P}, A = B
⊗
{; B}, •x = B
{P; }, P = x•


A
P
x
B
=
A
P
⊗
P
x
B
=
{A; }, ⊤
{A; P}, A = B
⊗
{; B}, •x = B
{P; }, P = x•
≃
{A; B}, •x = B
{; B}, •x = B
{A; }, A = x•
{A; }, ⊤
{A; }, ⊤

$$\frac{\begin{array}{l} q_1 \xrightarrow{P_1}_1 q_1' \ \ \textbf{and} \ \ q_2 \xrightarrow{P_2}_2 q_2' \\ \textbf{and} \ \ (P_1^{\text{in}} \cup P_1^{\text{out}}) \cap P_2 = (P_2^{\text{in}} \cup P_2^{\text{out}}) \cap P_1 \end{array}}{q \xrightarrow{(P_1 \cup P_2) \setminus (P_1 \cap P_2)} q'}$$

$$\frac{\begin{array}{l} q_1 \xrightarrow{P_1}_1 q_1' \ \ \textbf{and} \ \ q_2 \in Q_2 \\ \textbf{and} \ \ (P_2^{\text{in}} \cup P_2^{\text{out}}) \cap P_1 = \emptyset \end{array}}{(q_1, q_2) \xrightarrow{P_1} (q_1', q_2)} \qquad \frac{\begin{array}{l} q_2 \xrightarrow{P_2}_2 q_2' \ \ \textbf{and} \ \ q_1 \in Q_1 \\ \textbf{and} \ \ (P_1^{\text{in}} \cup P_1^{\text{out}}) \cap P_2 = \emptyset \end{array}}{(q_1, q_2) \xrightarrow{P_2} (q_1, q_2')}$$

Question: Redefine the LossySync automaton to make it context-sensitive (i.e., redefine such that the product of LossySync and Fifo yields an automaton *without* $(q_0, \{\texttt{A;}\}, \top, q_0)$)

Question: Redefine the LossySync automaton to make it context-sensitive (i.e., redefine such that the product of LossySync and Fifo yields an automaton *without* $(q_0, \{\mathtt{A};\}, \top, q_0)$)

Answer: There is none

- The automata considered so far are insufficiently expressive
- To elegantly support context-sensitivity, more information need to be captured

- The automata considered so far are insufficiently expressive
- To elegantly support context-sensitivity, more information need to be captured

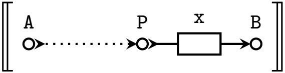
- Extend transitions from $q \xrightarrow{P,\varphi} q'$ to $q \xrightarrow{P^-,P,\varphi} q'$
  - P^- is the unreadiness constraint: P^- contains those ports that are unready to participate in the transition
  - P is the synchronization constraint
  - φ is the data constraint

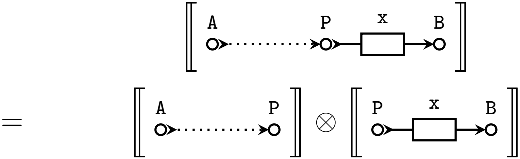
  such that $P^- \cap P = \emptyset$

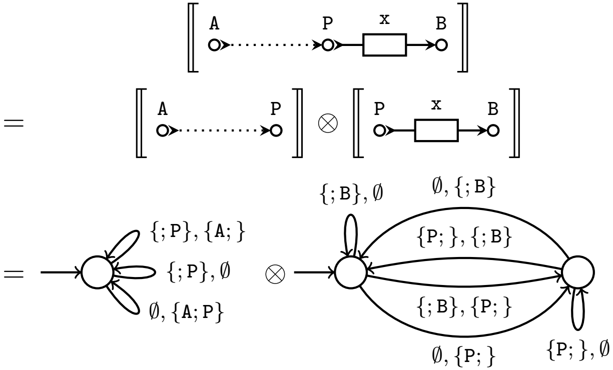
$$\frac{\begin{array}{l} q_1 \xrightarrow{P_1^-,P_1}_1 q_1' \;\; \textbf{and} \;\; q_2 \xrightarrow{P_2^-,P_2}_2 q_2' \\ \textbf{and} \;\; (P_1^{\text{in}} \cup P_1^{\text{out}}) \cap P_2^- = (P_2^{\text{in}} \cup P_2^{\text{out}}) \cap P_1^- \\ \textbf{and} \;\; (P_1^{\text{in}} \cup P_1^{\text{out}}) \cap P_2 = (P_2^{\text{in}} \cup P_2^{\text{out}}) \cap P_1 \end{array}}{q \xrightarrow{(P_1^- \cup P_2^-) \setminus (P_1^- \cap P_2^-),(P_1 \cup P_2) \setminus (P_1 \cap P_2)} q'}$$

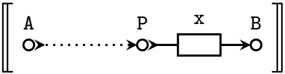
$$\frac{\begin{array}{l} q_1 \xrightarrow{P_1^-,P_1}_1 q_1' \;\; \textbf{and} \;\; q_2 \in Q_2 \\ \textbf{and} \;\; (P_2^{\text{in}} \cup P_2^{\text{out}}) \cap P_1^- = \emptyset \\ \textbf{and} \;\; (P_2^{\text{in}} \cup P_2^{\text{out}}) \cap P_1 = \emptyset \end{array}}{(q_1, q_2) \xrightarrow{P_1^-,P_1} (q_1', q_2)}$$

$$\frac{\begin{array}{l} q_2 \xrightarrow{P_2}_2 q_2' \;\; \textbf{and} \;\; q_1 \in Q_1 \\ \textbf{and} \;\; (P_1^{\text{in}} \cup P_1^{\text{out}}) \cap P_2^- = \emptyset \\ \textbf{and} \;\; (P_1^{\text{in}} \cup P_1^{\text{out}}) \cap P_2 = \emptyset \end{array}}{(q_1, q_2) \xrightarrow{P_2^-,P_2} (q_1, q_2')}$$

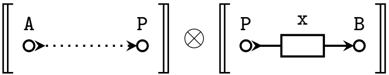

A
P
x
B


A
P
x
B
=
A
P
⊗
P
x
B

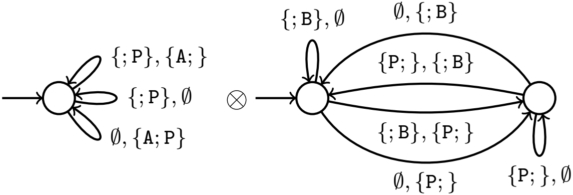

A
P
x
B
=
A
P
⊗
P
x
B
=
{;P}, {A;}
{;P}, ∅
∅, {A;P}
⊗
{;B}, ∅
∅, {;B}
{P;}, {;B}
{;B}, {P;}
∅, {P;}
{P;}, ∅



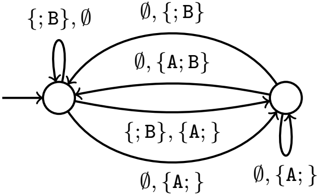
$=$



$=$



$\simeq$



Summary:

- Context-sensitivity seems a desirable semantic feature to support
- Automata with unreadiness constraints are just *one* possible model that supports context-sensitivity
- *Many* others exist:
    - 3-coloring semantics
    - Intentional automata
    - Guarded automata
    - Action constraint automata
    - ...
- Context-sensitivity is, and has been, an important topic in the Reo community

Final slides

Other topics:

- Other semantic models for Reo
- Verification and analysis (model checking, quantitative analysis, QoS reasoning)
- Other applications:
    - Web service composition
    - Business process modeling
    - Multi-agent systems
    - Biological systems

Thank you!