PROTOCOL PROGRAMMING WITH AUTOMATA: (WHY AND) HOW?

Sung-Shik Jongmans[1,2(,3,4)] (Farhad Arbab[3,4])

[1]Open University, the Netherlands
[2]Radboud University, the Netherlands
[3]Centrum Wiskunde & Informatica, the Netherlands
[4]Leiden University, the Netherlands

29 September 2015

Previous talk: “We need a higher level of abstraction (than conventional stuff) for programming protocols.”

Previous talk: "We need a higher level of abstraction (than conventional stuff) for programming protocols."

This talk: "Here is one option."

Running example: **Producers** / **consumer** protocol

(Producers: Alice, Bob; consumer: Carol)

Properties: Asynchronous, reliable, unordered, transactional

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

Typical implementation (Ben-Ari's textbook)

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

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

  @Override
  public void run() {
    while (true) {
      Object datum = Thread.currentThread().getId();
      buffer.empty.acquire();
      buffer.content = datum;
      buffer.full.release();
} } }
```

Typical implementation (Ben-Ari's textbook)

```
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 datum = buffer.content;
      buffer.empty.release();
      System.out.println(datum);
} } }
```

Typical implementation (Ben-Ari's textbook)

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

    Producer alice = new Producer(buffer);
    Producer   bob = new Producer(buffer);
    Consumer carol = new Consumer(buffer);

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

Typical implementation (Ben-Ari's textbook)

Now, forget **everything** you know about sempahores, data races, shared memory, mutual exclusion, ...

Now, forget **everything** you know about
sempahores, data races, shared memory, mutual exclusion, ...

Let there be only **ports**.

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

- Every process **owns** a set of ports.
- Ports mark the **interface** between processes.
- **All** inter-process interaction occurs through ports.

- Every process **owns** a set of ports.
- Ports mark the **interface** between processes.
- **All** inter-process interaction occurs through ports.
- Processes perform **blocking** operations on ports.

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

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

- Every process **owns** a set of ports.
- Ports mark the **interface** between processes.
- **All** inter-process interaction occurs through ports.
- Processes perform **blocking** operations on ports.

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

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

- Processes are **oblivious** to data-flow.
  $\Rightarrow$ *That's what we have protocols for.*

```
public class Process {
  public static void Producer(InputPort port) {
    while (true) {
      Object datum = Thread.currentThread().getId();
      port.put(datum);
  } }

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

Port-based implementation

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



    Thread alice = new Thread() {
      public void run() { Process.Producer(A) } }
    Thread   bob = new Thread() {
      public void run() { Process.Producer(B) } }
    Thread carol = new Thread() {
      public void run() { Process.Consumer(C) } }

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

Port-based implementation

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

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

    Thread alice = new Thread() {
      public void run() { Process.Producer(A) } }
    Thread   bob = new Thread() {
      public void run() { Process.Producer(B) } }
    Thread carol = new Thread() {
      public void run() { Process.Consumer(C) } }

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

Port-based implementation

Q: Where does `Protocol(A,B,C)` come from?

A: `Protocol(A,B,C)` is specified in a DSL for protocols.

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

During a program run, `put`/`get` operations complete.

We call an *infinite* sequence of completions an **interaction**.

We call a set of *admissible* interactions a **protocol**.

An interaction is a function $w : \mathbb{N} \to \underbrace{\mathbb{P} \rightharpoonup \mathbb{D}}_{\text{multiactions}}$

Examples:

- $\{\mathtt{A} \mapsto \mathtt{0}\}, \{\mathtt{C} \mapsto \mathtt{0}\}, \{\mathtt{A} \mapsto \mathtt{1}\}, \{\mathtt{C} \mapsto \mathtt{1}\}, \{\mathtt{A} \mapsto \mathtt{2}\}, \{\mathtt{C} \mapsto \mathtt{2}\}, \ldots$

An interaction is a function $w : \mathbb{N} \to \underbrace{\mathbb{P} \rightharpoonup \mathbb{D}}_{\text{multiactions}}$

Examples:

- $\{\texttt{A} \mapsto \texttt{0}\}, \{\texttt{C} \mapsto \texttt{0}\}, \{\texttt{A} \mapsto \texttt{1}\}, \{\texttt{C} \mapsto \texttt{1}\}, \{\texttt{A} \mapsto \texttt{2}\}, \{\texttt{C} \mapsto \texttt{2}\}, \ldots$
- $\{\texttt{A} \mapsto \texttt{0}\}, \{\texttt{C} \mapsto \texttt{0}\}, \{\texttt{B} \mapsto \texttt{1}\}, \{\texttt{C} \mapsto \texttt{1}\}, \{\texttt{B} \mapsto \texttt{2}\}, \{\texttt{C} \mapsto \texttt{2}\}, \ldots$

An interaction is a function $w : \mathbb{N} \to \underbrace{\mathbb{P} \rightharpoonup \mathbb{D}}_{\text{multiactions}}$

Examples:

- $\{\mathtt{A} \mapsto \mathtt{0}\}, \{\mathtt{C} \mapsto \mathtt{0}\}, \{\mathtt{A} \mapsto \mathtt{1}\}, \{\mathtt{C} \mapsto \mathtt{1}\}, \{\mathtt{A} \mapsto \mathtt{2}\}, \{\mathtt{C} \mapsto \mathtt{2}\}, \ldots$
- $\{\mathtt{A} \mapsto \mathtt{0}\}, \{\mathtt{C} \mapsto \mathtt{0}\}, \{\mathtt{B} \mapsto \mathtt{1}\}, \{\mathtt{C} \mapsto \mathtt{1}\}, \{\mathtt{B} \mapsto \mathtt{2}\}, \{\mathtt{C} \mapsto \mathtt{2}\}, \ldots$
- $\{\mathtt{A} \mapsto \mathtt{0}, \mathtt{C} \mapsto \mathtt{0}\}, \{\mathtt{B} \mapsto \mathtt{1}, \mathtt{C} \mapsto \mathtt{1}\}, \ldots$

$$\text{An interaction is a function } w : \mathbb{N} \to \underbrace{\mathbb{P} \rightharpoonup \mathbb{D}}_{\text{multiactions}}$$

Examples:

- $\{\mathtt{A} \mapsto \mathtt{0}\}, \{\mathtt{C} \mapsto \mathtt{0}\}, \{\mathtt{A} \mapsto \mathtt{1}\}, \{\mathtt{C} \mapsto \mathtt{1}\}, \{\mathtt{A} \mapsto \mathtt{2}\}, \{\mathtt{C} \mapsto \mathtt{2}\}, \ldots$
- $\{\mathtt{A} \mapsto \mathtt{0}\}, \{\mathtt{C} \mapsto \mathtt{0}\}, \{\mathtt{B} \mapsto \mathtt{1}\}, \{\mathtt{C} \mapsto \mathtt{1}\}, \{\mathtt{B} \mapsto \mathtt{2}\}, \{\mathtt{C} \mapsto \mathtt{2}\}, \ldots$
- ~~$\{\mathtt{A} \mapsto \mathtt{0}, \mathtt{C} \mapsto \mathtt{0}\}, \{\mathtt{B} \mapsto \mathtt{1}, \mathtt{C} \mapsto \mathtt{1}\}, \ldots$~~ [synchronous]

An interaction is a function $w : \mathbb{N} \to \underbrace{\mathbb{P} \rightharpoonup \mathbb{D}}_{\text{multiactions}}$

Examples:

- $\{\texttt{A} \mapsto \texttt{0}\}, \{\texttt{C} \mapsto \texttt{0}\}, \{\texttt{A} \mapsto \texttt{1}\}, \{\texttt{C} \mapsto \texttt{1}\}, \{\texttt{A} \mapsto \texttt{2}\}, \{\texttt{C} \mapsto \texttt{2}\}, \ldots$
- $\{\texttt{A} \mapsto \texttt{0}\}, \{\texttt{C} \mapsto \texttt{0}\}, \{\texttt{B} \mapsto \texttt{1}\}, \{\texttt{C} \mapsto \texttt{1}\}, \{\texttt{B} \mapsto \texttt{2}\}, \{\texttt{C} \mapsto \texttt{2}\}, \ldots$
- ~~$\{\texttt{A} \mapsto \texttt{0}, \texttt{C} \mapsto \texttt{0}\}, \{\texttt{B} \mapsto \texttt{1}, \texttt{C} \mapsto \texttt{1}\}, \ldots$~~ [synchronous]
- $\{\texttt{A} \mapsto \texttt{0}\}, \{\texttt{C} \mapsto \texttt{4}\}, \{\texttt{B} \mapsto \texttt{1}\}, \{\texttt{C} \mapsto \texttt{1}\}, \ldots$

An interaction is a function $w : \mathbb{N} \to \underbrace{\mathbb{P} \rightharpoonup \mathbb{D}}_{\text{multiactions}}$

Examples:

- $\{\texttt{A} \mapsto \texttt{0}\}, \{\texttt{C} \mapsto \texttt{0}\}, \{\texttt{A} \mapsto \texttt{1}\}, \{\texttt{C} \mapsto \texttt{1}\}, \{\texttt{A} \mapsto \texttt{2}\}, \{\texttt{C} \mapsto \texttt{2}\}, \ldots$
- $\{\texttt{A} \mapsto \texttt{0}\}, \{\texttt{C} \mapsto \texttt{0}\}, \{\texttt{B} \mapsto \texttt{1}\}, \{\texttt{C} \mapsto \texttt{1}\}, \{\texttt{B} \mapsto \texttt{2}\}, \{\texttt{C} \mapsto \texttt{2}\}, \ldots$
- ~~$\{\texttt{A} \mapsto \texttt{0}, \texttt{C} \mapsto \texttt{0}\}, \{\texttt{B} \mapsto \texttt{1}, \texttt{C} \mapsto \texttt{1}\}, \ldots$~~ [synchronous]
- ~~$\{\texttt{A} \mapsto \texttt{0}\}, \{\texttt{C} \mapsto \texttt{4}\}, \{\texttt{B} \mapsto \texttt{1}\}, \{\texttt{C} \mapsto \texttt{1}\}, \ldots$~~ [unreliable]

$$\text{An interaction is a function } w : \mathbb{N} \to \underbrace{\mathbb{P} \rightharpoonup \mathbb{D}}_{\text{multiactions}}$$

Examples:

- $\{\mathtt{A} \mapsto \mathtt{0}\}, \{\mathtt{C} \mapsto \mathtt{0}\}, \{\mathtt{A} \mapsto \mathtt{1}\}, \{\mathtt{C} \mapsto \mathtt{1}\}, \{\mathtt{A} \mapsto \mathtt{2}\}, \{\mathtt{C} \mapsto \mathtt{2}\}, \ldots$
- $\{\mathtt{A} \mapsto \mathtt{0}\}, \{\mathtt{C} \mapsto \mathtt{0}\}, \{\mathtt{B} \mapsto \mathtt{1}\}, \{\mathtt{C} \mapsto \mathtt{1}\}, \{\mathtt{B} \mapsto \mathtt{2}\}, \{\mathtt{C} \mapsto \mathtt{2}\}, \ldots$
- ~~$\{\mathtt{A} \mapsto \mathtt{0}, \mathtt{C} \mapsto \mathtt{0}\}, \{\mathtt{B} \mapsto \mathtt{1}, \mathtt{C} \mapsto \mathtt{1}\}, \ldots$~~ [synchronous]
- ~~$\{\mathtt{A} \mapsto \mathtt{0}\}, \{\mathtt{C} \mapsto \mathtt{4}\}, \{\mathtt{B} \mapsto \mathtt{1}\}, \{\mathtt{C} \mapsto \mathtt{1}\}, \ldots$~~ [unreliable]
- $\{\mathtt{A} \mapsto \mathtt{0}\}, \{\mathtt{B} \mapsto \mathtt{1}\}, \{\mathtt{C} \mapsto \mathtt{0}\}, \{\mathtt{C} \mapsto \mathtt{1}\}, \ldots$

$$\text{An interaction is a function } w : \mathbb{N} \to \underbrace{\mathbb{P} \rightharpoonup \mathbb{D}}_{\text{multiactions}}$$

Examples:

- $\{\mathtt{A} \mapsto \mathtt{0}\}, \{\mathtt{C} \mapsto \mathtt{0}\}, \{\mathtt{A} \mapsto \mathtt{1}\}, \{\mathtt{C} \mapsto \mathtt{1}\}, \{\mathtt{A} \mapsto \mathtt{2}\}, \{\mathtt{C} \mapsto \mathtt{2}\}, \ldots$
- $\{\mathtt{A} \mapsto \mathtt{0}\}, \{\mathtt{C} \mapsto \mathtt{0}\}, \{\mathtt{B} \mapsto \mathtt{1}\}, \{\mathtt{C} \mapsto \mathtt{1}\}, \{\mathtt{B} \mapsto \mathtt{2}\}, \{\mathtt{C} \mapsto \mathtt{2}\}, \ldots$
- ~~$\{\mathtt{A} \mapsto \mathtt{0}, \mathtt{C} \mapsto \mathtt{0}\}, \{\mathtt{B} \mapsto \mathtt{1}, \mathtt{C} \mapsto \mathtt{1}\}, \ldots$~~ [synchronous]
- ~~$\{\mathtt{A} \mapsto \mathtt{0}\}, \{\mathtt{C} \mapsto \mathtt{4}\}, \{\mathtt{B} \mapsto \mathtt{1}\}, \{\mathtt{C} \mapsto \mathtt{1}\}, \ldots$~~ [unreliable]
- ~~$\{\mathtt{A} \mapsto \mathtt{0}\}, \{\mathtt{B} \mapsto \mathtt{1}\}, \{\mathtt{C} \mapsto \mathtt{0}\}, \{\mathtt{C} \mapsto \mathtt{1}\}, \ldots$~~ [nontransactional]

A protocol is a set $L \subseteq \mathbb{N} \to (\mathbb{P} \rightharpoonup \mathbb{D})$

Example:

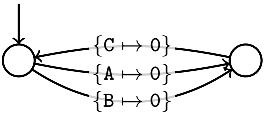
$$\left\{ w \;\middle|\; \begin{array}{l} w : \mathbb{N} \to (\mathbb{P} \rightharpoonup \mathbb{D}) \\ \textbf{and}\ \ \left[\begin{bmatrix} \quad\ \mathrm{Dom}(w(i)) = \{\mathtt{A}\} \\ \textbf{or}\ \ \mathrm{Dom}(w(i)) = \{\mathtt{B}\} \end{bmatrix}\ \textbf{for all}\ \ i \in \mathbb{N}_{\text{even}}\right]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

Interactions are **words**; Protocols are **languages**.

First attempt: An automaton is a tuple $(Q, P, \longrightarrow, q_0)$, where $\longrightarrow \subseteq Q \times (P \rightharpoonup \mathbb{D}) \times Q$

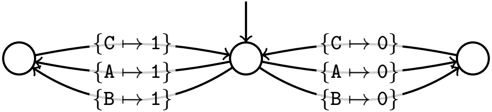
First attempt: An automaton is a tuple $(Q, P, \longrightarrow, q_0)$, where $\longrightarrow \subseteq Q \times (P \rightharpoonup \mathbb{D}) \times Q$

Example: Producers/consumer, $\mathbb{D} = \{\texttt{0}\}$



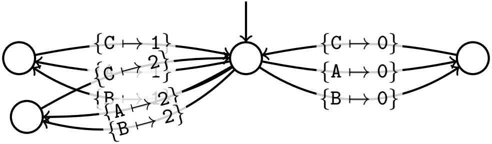
First attempt: An automaton is a tuple $(Q, P, \longrightarrow, q_0)$, where $\longrightarrow \subseteq Q \times (P \rightharpoonup \mathbb{D}) \times Q$

Example: Producers/consumer, $\mathbb{D} = \{0, 1\}$



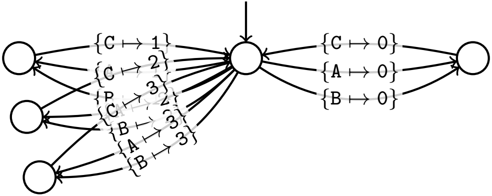
First attempt: An automaton is a tuple $(Q, P, \longrightarrow, q_0)$, where $\longrightarrow \subseteq Q \times (P \rightharpoonup \mathbb{D}) \times Q$

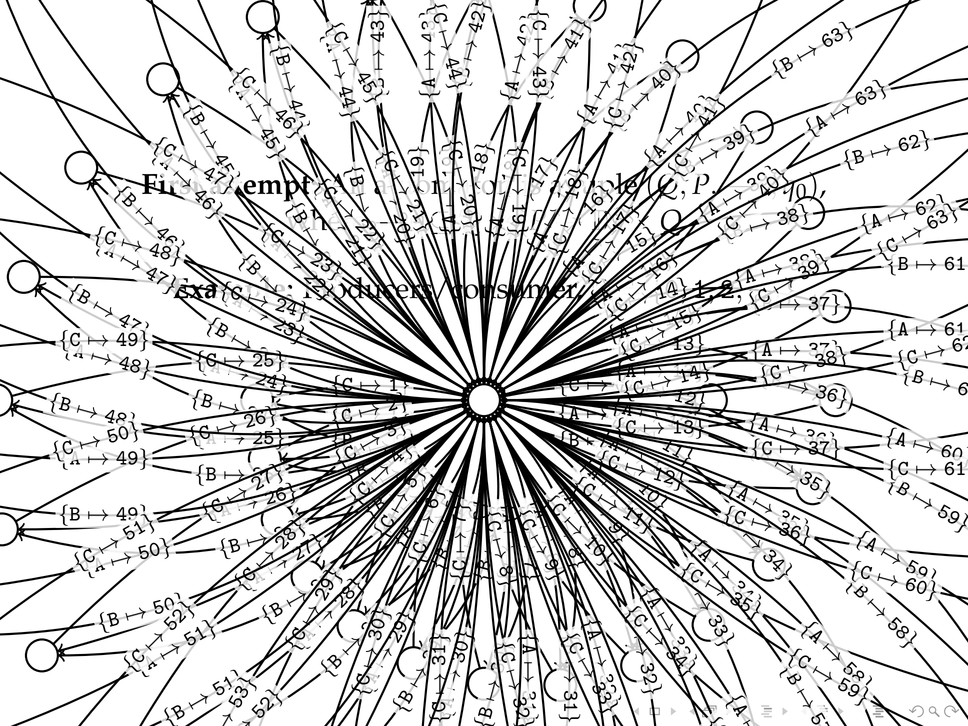
Example: Producers/consumer, $\mathbb{D} = \{\mathtt{0}, \mathtt{1}, \mathtt{2}, \ldots\}$



First attempt: An automaton is a tuple $(Q, P, \longrightarrow, q_0)$, where $\longrightarrow \subseteq Q \times (P \rightharpoonup \mathbb{D}) \times Q$

Example: Producers/consumer, $\mathbb{D} = \{\mathtt{0}, \mathtt{1}, \mathtt{2}, \ldots\}$

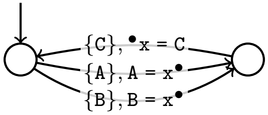




Second attempt: An automaton is a tuple $(Q, P, M, \longrightarrow, q_0, \mu_0)$, where $\longrightarrow \subseteq Q \times 2^P \times \mathbb{DC} \times Q$

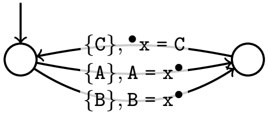
Second attempt: An automaton is a tuple $(Q, P, M, \longrightarrow, q_0, \mu_0)$, where $\longrightarrow \subseteq Q \times 2^P \times \mathbb{DC} \times Q$

Example: Producers/consumer, $\mathbb{D} = \{\texttt{0}\}$



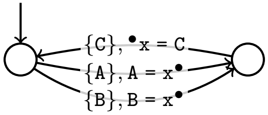
Second attempt: An automaton is a tuple $(Q, P, M, \longrightarrow, q_0, \mu_0)$, where $\longrightarrow \subseteq Q \times 2^P \times \mathbb{DC} \times Q$

Example: Producers/consumer, $\mathbb{D} = \{\mathtt{0}, \mathtt{1}\}$



Second attempt: An automaton is a tuple $(Q, P, M, \longrightarrow, q_0, \mu_0)$, where $\longrightarrow \subseteq Q \times 2^P \times \mathbb{DC} \times Q$

Example: Producers/consumer, $\mathbb{D} = \{\texttt{0}, \texttt{1}, \texttt{2}, \ldots\}$



Constraint automata may remind one of pushdown automata.

Constraint automata may remind one of 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 content of memory cells (“stack”)

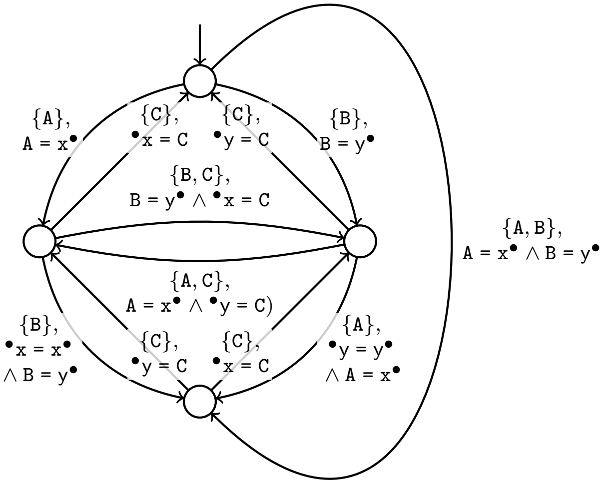
Constraint automata may remind one of pushdown automata.

- **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 content of memory cells ("stack")

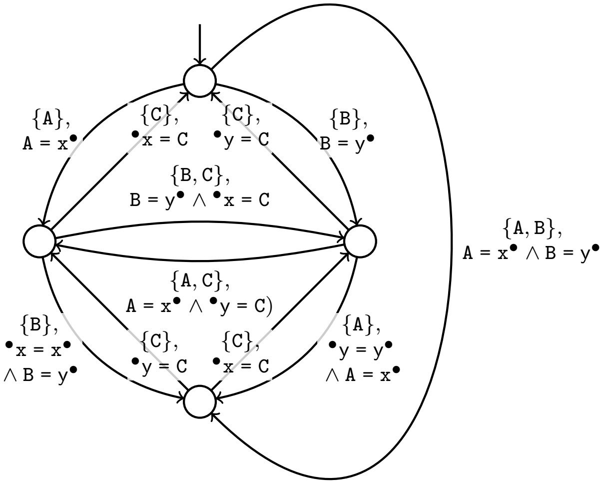
- **Move:** $(q, w, \mu') \vdash (q', w', \mu')$

Constraint automata may remind one of 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 content of memory cells ("stack")
- **Move:** $(q, w, \mu') \vdash (q', w', \mu')$
- **Language:** $\{w \mid (q_0, w, \mu_0) \vdash (q_1, w', \mu_1) \vdash (q_2, w'', \mu_2) \vdash \cdots\}$
  $\Rightarrow$ *Automata model protocols, operationally.*



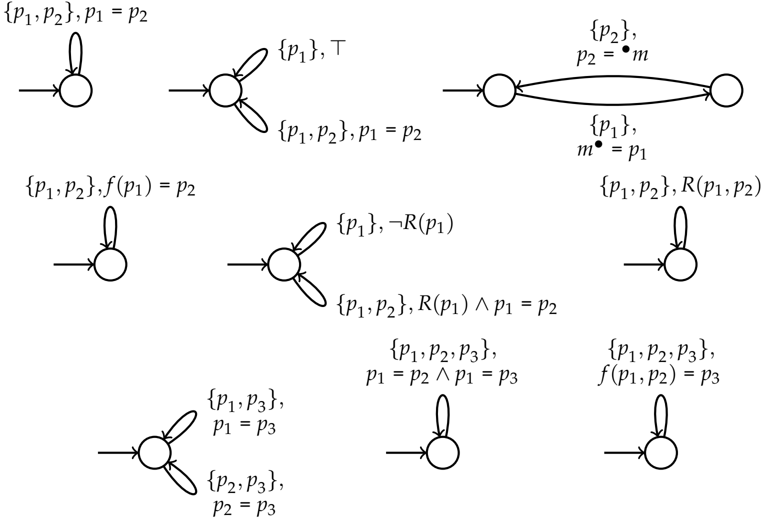
Asynchronous, reliable, unordered, **non**transactional



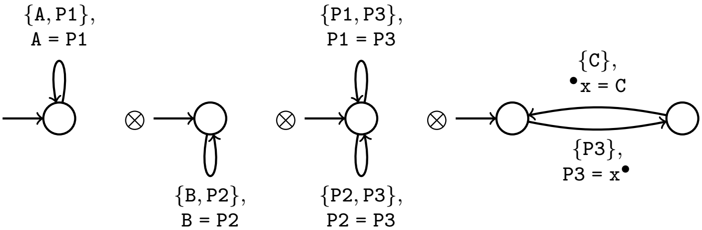
For k producers, 2^k states and $\mathcal{O}(k \cdot 2^k)$ transitions *per state*

Compositional construction, through **multiplication**

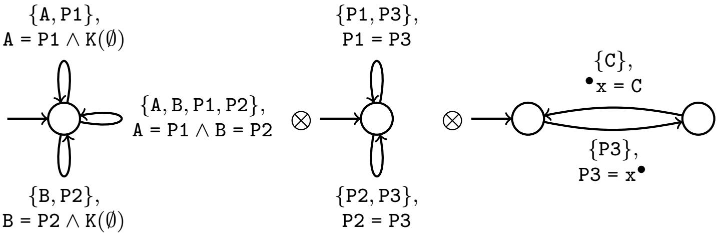
Compositional construction, through **multiplication**



Example:

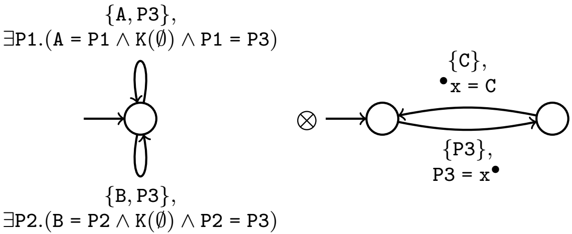


Example:

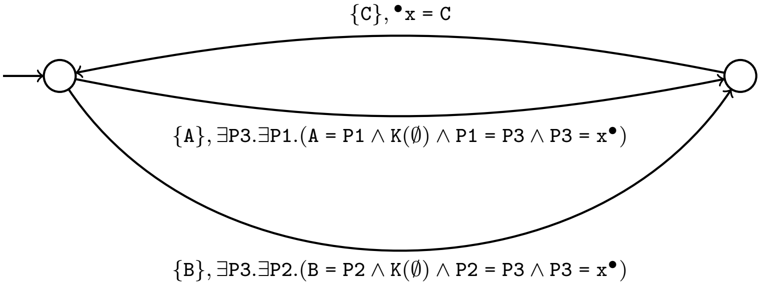


{A, P1},
A = P1 ∧ K(∅)
{A, B, P1, P2},
A = P1 ∧ B = P2
{B, P2},
B = P2 ∧ K(∅)
⊗
{P1, P3},
P1 = P3
{P2, P3},
P2 = P3
⊗
{C},
•x = C
{P3},
P3 = x•

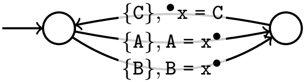
Example:



Example:



Example:



Definition:

$$\begin{pmatrix} Q_1, \\ P_1, \\ M_1, \\ \longrightarrow_1, \\ q_1^0, \\ \mu_1^0 \end{pmatrix} \otimes \begin{pmatrix} Q_2, \\ P_2, \\ M_2, \\ \longrightarrow_2, \\ q_2^0 \\ \mu_2^0 \end{pmatrix} = \begin{pmatrix} Q_1 \times Q_2, \\ P_1 \triangle P_2, \\ M_1 \cup M_2, \\ \longrightarrow, \\ (q_1^0, q_2^0), \\ \mu_1^0 \cup \mu_2^0 \end{pmatrix} \quad \textbf{if} \;\; M_1 \cap M_2 = \emptyset$$

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

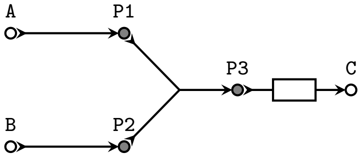
$$\frac{q_1 \xrightarrow{P_1^{\mathrm{tr}},\phi_1}_1 q_1' \ \ \textbf{and}\ \ q_2 \xrightarrow{P_2^{\mathrm{tr}},\phi_2}_2 q_2' \ \ \textbf{and}\ \ P_1 \cap P_2^{\mathrm{tr}} = P_2 \cap P_1^{\mathrm{tr}}}{q \xrightarrow{P_1^{\mathrm{tr}} \triangle P_2^{\mathrm{tr}},\exists P_1^{\mathrm{tr}} \cap P_2^{\mathrm{tr}}.\phi_1 \wedge \phi_2} q'}$$

$$\frac{q_1 \xrightarrow{P_1^{\mathrm{tr}},\phi_1}_1 q_1' \ \ \textbf{and}\ \ q_2 \in Q_2 \ \ \textbf{and}\ \ P_2 \cap P_1^{\mathrm{tr}} = \emptyset}{(q_1, q_2) \xrightarrow{P_1^{\mathrm{tr}},\phi_1 \wedge \mathtt{K}(M_2)} (q_1', q_2)}$$

$$\frac{q_2 \xrightarrow{P_2^{\mathrm{tr}},\phi_2}_2 q_2' \ \ \textbf{and}\ \ q_1 \in Q_1 \ \ \textbf{and}\ \ P_1 \cap P_2^{\mathrm{tr}} = \emptyset}{(q_1, q_2) \xrightarrow{P_2^{\mathrm{tr}},\phi_2 \wedge \mathtt{K}(M_1)} (q_1, q_2')}$$

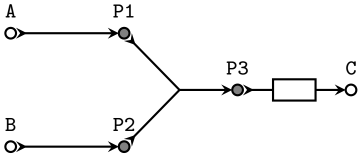
Syntax for multiplication expressions:

- **Graphical:** Every **edge** denotes an automaton.

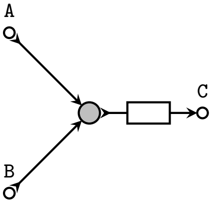


Syntax for multiplication expressions:

- **Graphical:** Every **edge** denotes an automaton.

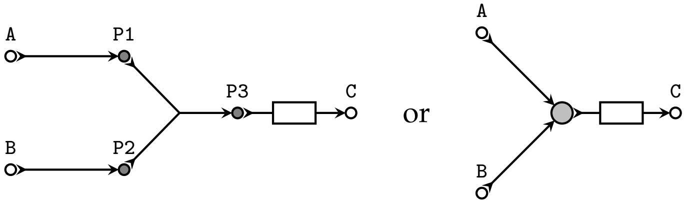


or



Syntax for multiplication expressions:

- **Graphical:** Every **edge** denotes an automaton.



- **Textual:** Every **signature** denotes an automaton.

```
LateAsyncMerger(a,b;c) =
  Sync(a;P1) mult Sync(b;P2) mult Merger2(P1,P2;P3) mult Fifo(P3;c)

main = LateAsyncMerger(A,B;C)
```

Demo

Compilation:

1. Extract a list of **“small”** automata from the syntax.
2. Multiply those automata into one **“big”** automaton.
3. Translate that automaton into state machine code.

Compilation:

1. Extract a list of **“small”** automata from the syntax.
2. Multiply those automata into one **“big”** automaton.
3. Translate that automaton into state machine code.

Explanation fits on less than half a slide :).

Compilation:

1. Extract a list of **“small”** automata from the syntax.
2. Multiply those automata into one **“big”** automaton.
3. Translate that automaton into state machine code.

Explanation fits on less than half a slide :).

It does not work very well :(.

Compilation:

1. Extract a list of **"small"** automata from the syntax.
2. Multiply those automata into one **"big"** automaton.
3. Translate that automaton into state machine code.

Explanation fits on less than half a slide :).

It does not work very well :(.

I need **four chapters** of optimizations to get *some* performance.

That's it.

Questions?