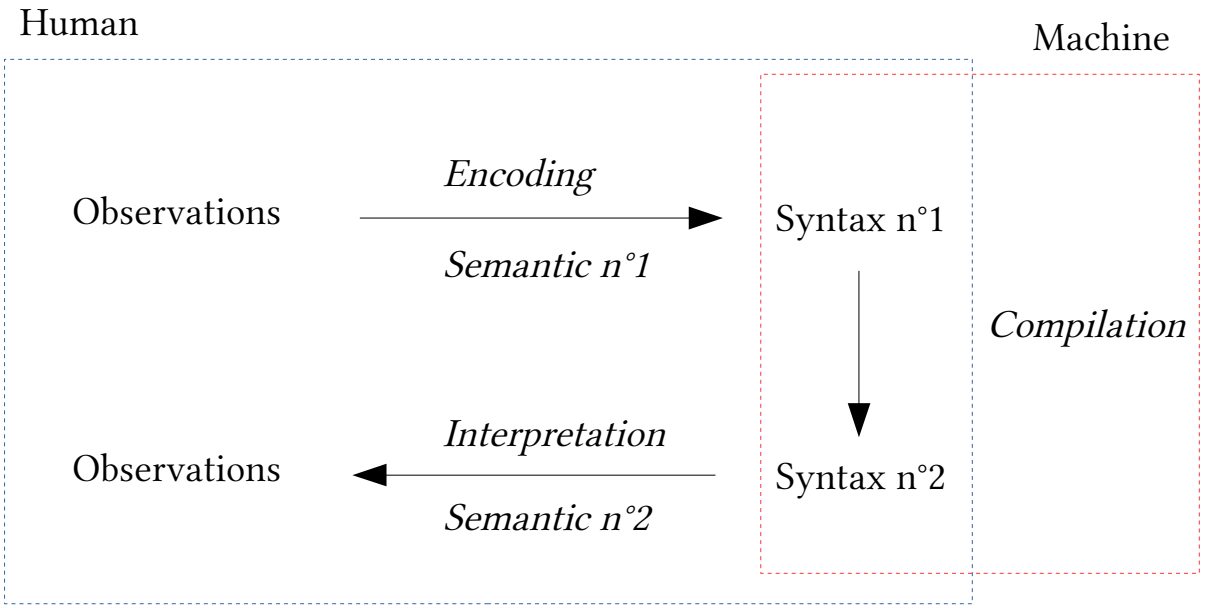
On Reo compiler

Intuition

Intuition


Human
Machine
Observations
Encoding
Semantic n°1
Syntax n°1
Compilation
Observations
Interpretation
Semantic n°2
Syntax n°2

Objectives

Reo is a language for modeling coordination and concurrency:

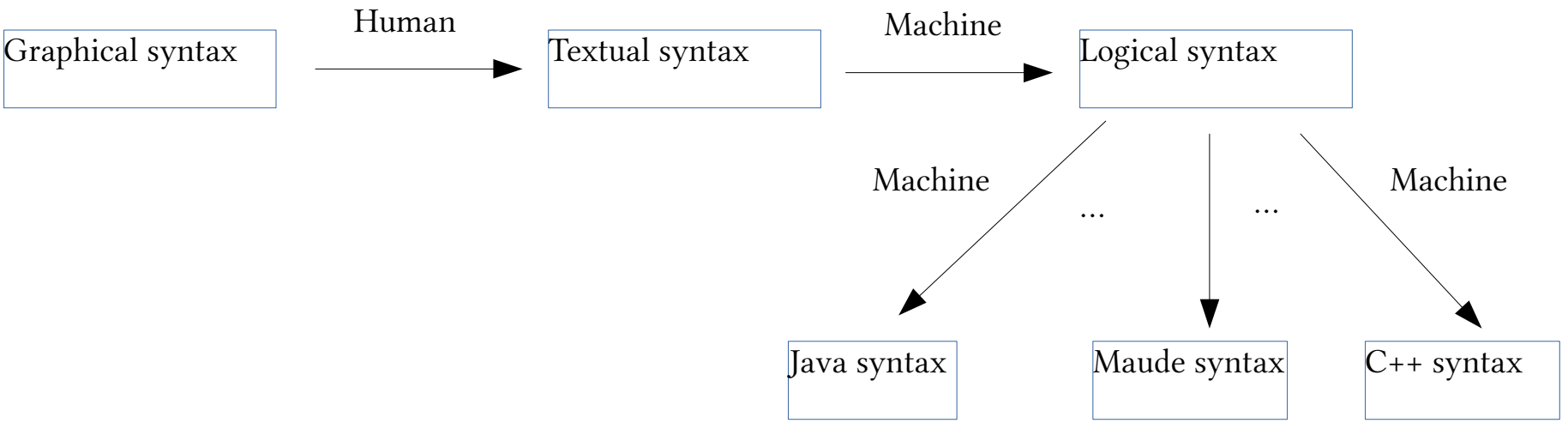
- Breaks complexity of concurrency with explicit interaction primitives.
- Breaks complexity of design with composition.

However, does not execute (as Java, C) or verify directly its specification.

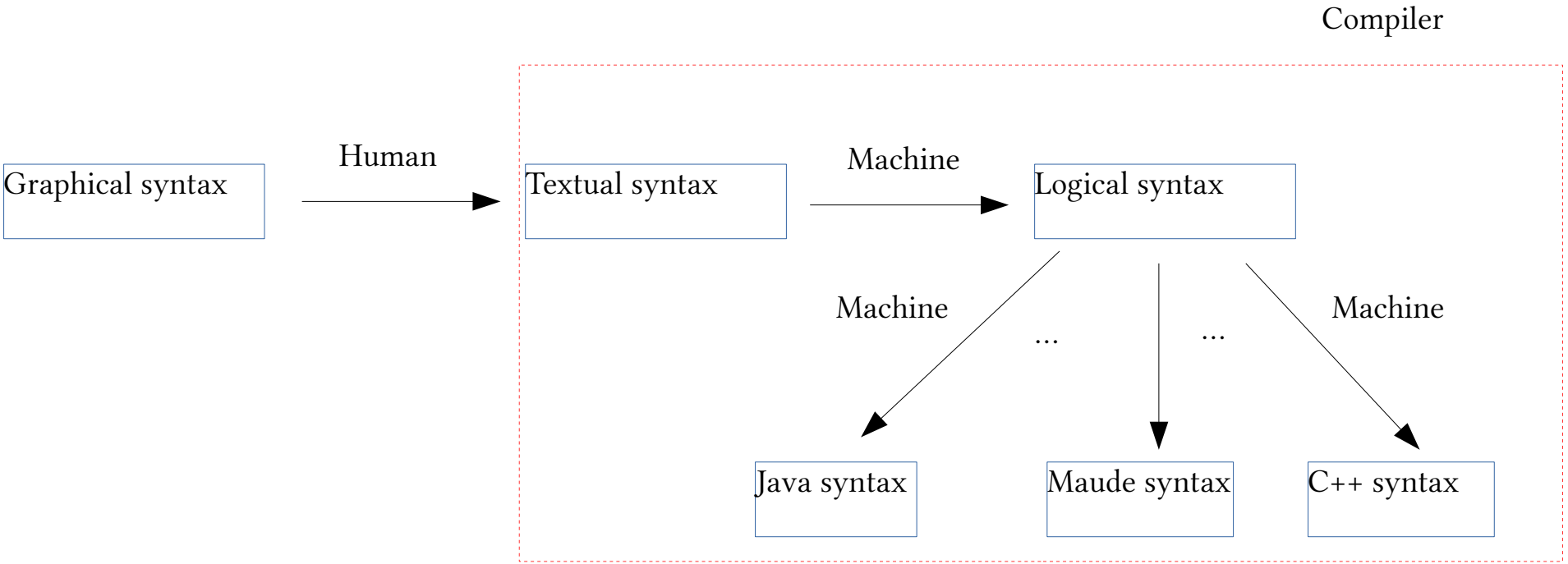
How to navigate between languages?

How can we preserve the meaning during translation?

A game of translation



A game of translation



Goal:

- all translations describe the same model: semantic preserving.

- use full power of each language to give information about the system.

Language: Reo

Grammar for textual syntax:

C := N <P> (S) { C* A* } | N <P> (S)	Component composite or atomic Component instance
N := STRING	Name
P := (STRING (, STRING)*)$^+$	Parameters
S := STRING (, STRING)*	Ports signature
A := User_Defined_Syntax	Semantic

- Recursive definition: components are **set** of composites or atomics

- STRING is an identifier for ports and parameters

Language: Reo

Example:

```
main() {
        alternator(a,b,c){
                sync(a,b){ #? }
                syncdrain(a,c) { #? }
                fifo(c,b) { #? } }
        green(a) { #? }
        blue(b) { #? }
        red(c) { #? } }
```

Syntax of fifo for Constraint Automaton:

fifo(c,b) { #CA
$q_0 \longrightarrow q_1$: { c , ~b} m'=c
$q_1 \longrightarrow q_0$: { ~c , b} m=b }

Syntax of fifo for Java:

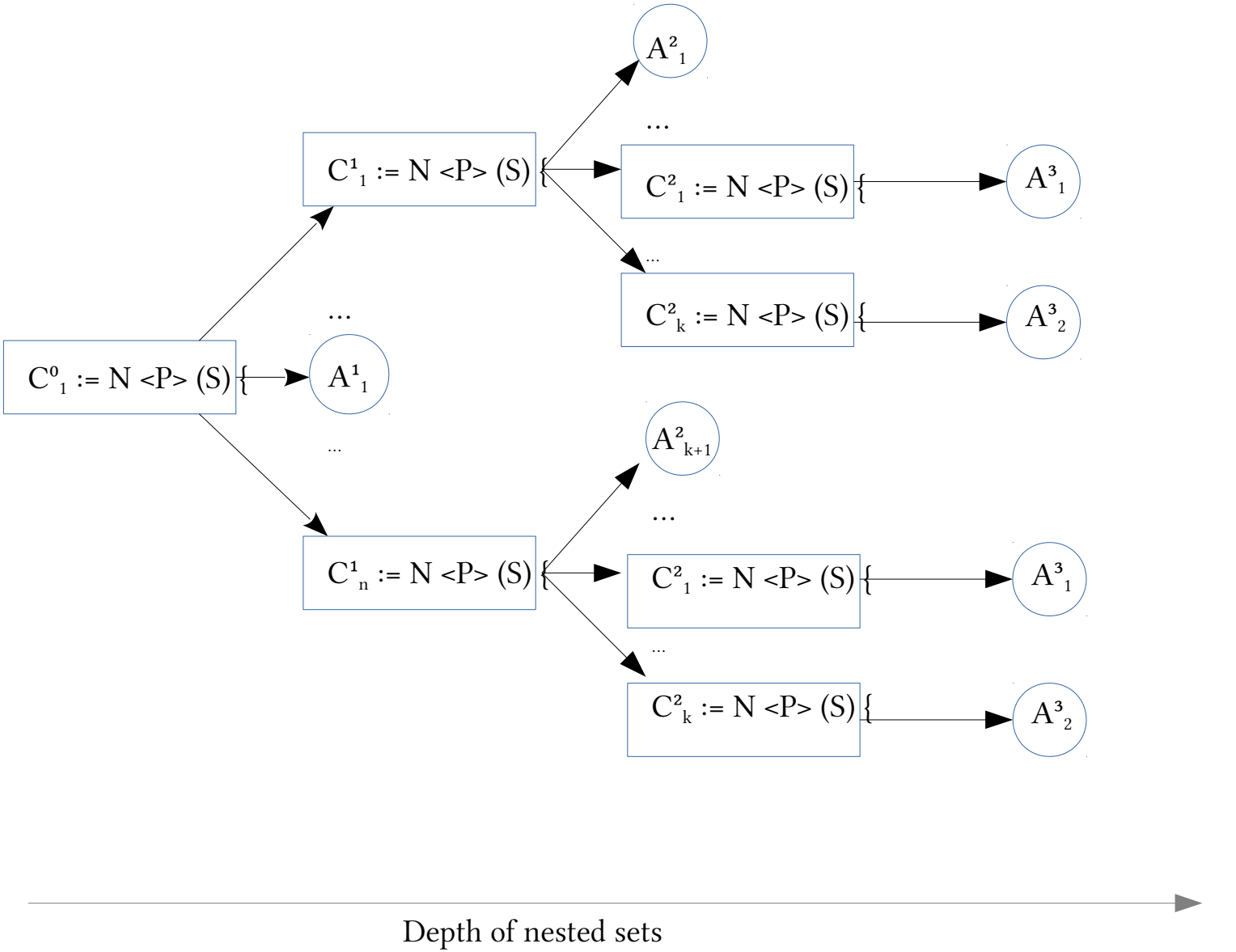
fifo(c,b) { #JAVA
"Function.fifo" }

What would be the syntax for fifo in Latex ?

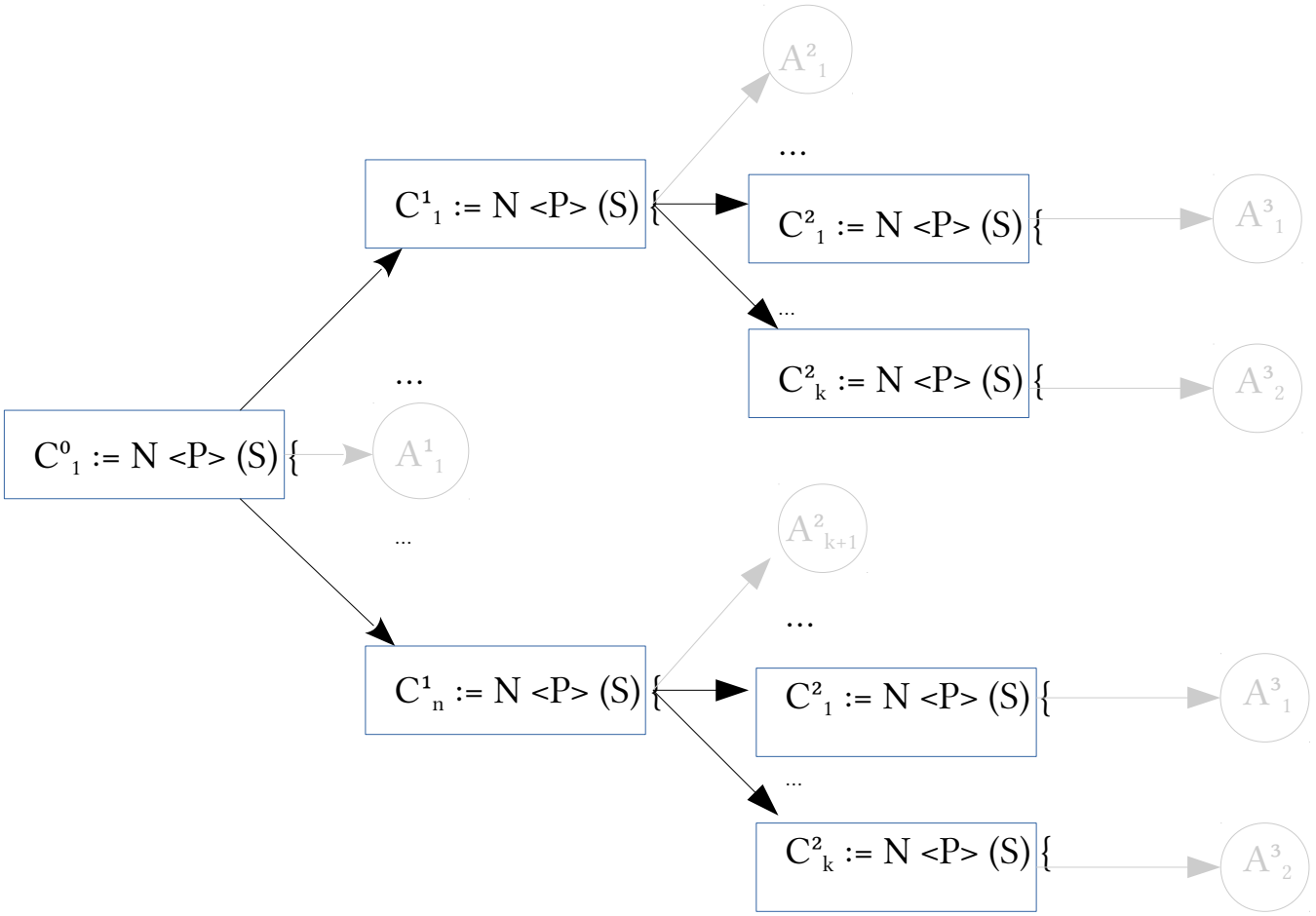
Remarks:
- Components in sets are instances of some definitions : we can save the definition and instantiate several components within a certain scope.

- How do we get the global behavior from a nested set component?

Machine representation: Parse Tree

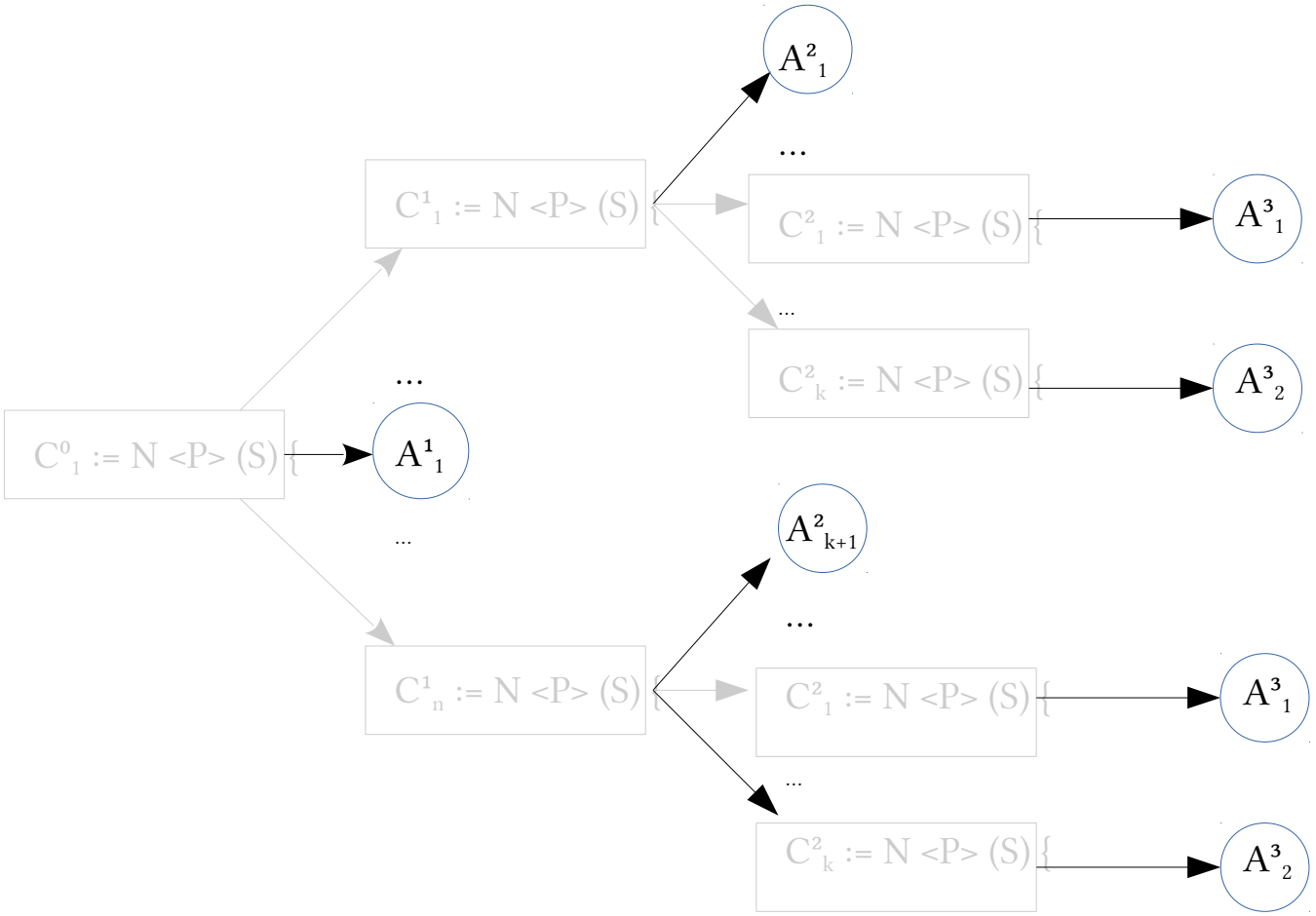


Machine representation: Parse Tree



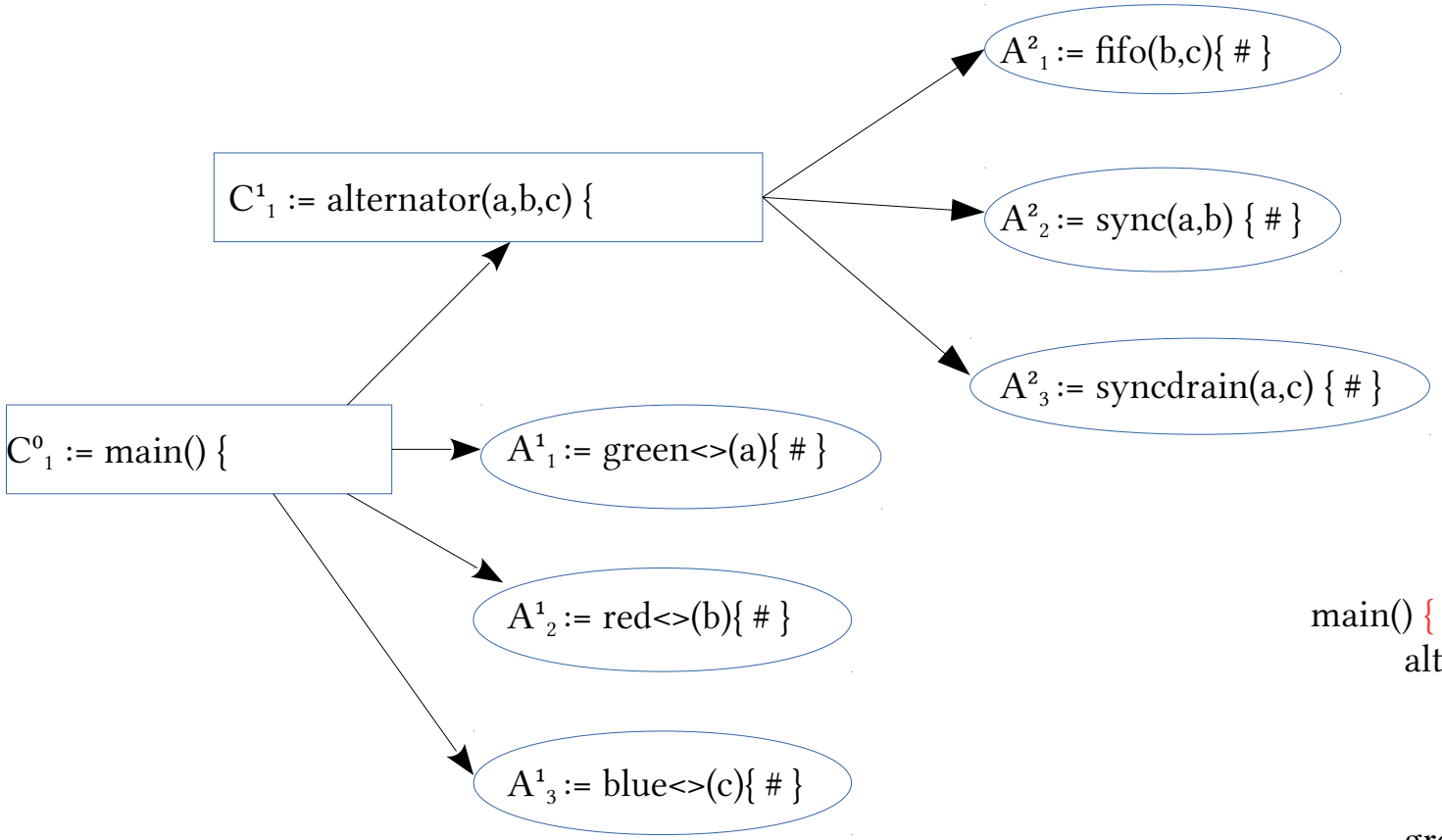
Protocol structure: no computation, just interaction

Machine representation: Parse Tree



Computational structure: computation, no interaction.

Machine representation: Parse Tree




```
main() {
        alternator(a,b,c){
                sync(a,b){ #? }
                syncdrain(a,c) { #? }
                fifo(c,b) { #? } }
        green(a) { #? }
        blue(b) { #? }
        red(c) { #? } }
```

Parse Tree for xrouter?

For unbounded fifo?

For alternator<k> ?

Parse Tree Evaluation

Evaluation of a parse tree expression regarding a scope :

$$\mathbf{M}[\ -\ ] : \text{ParseTreeExpression} \longrightarrow (\ \mathbb{D}\ )^{\Sigma}$$

$\mathbf{M}[\ -\ ]$: mapping function that evaluates an expression

$\mathbb{D}$: domain of the evaluated expression

Σ : scope to evaluate the expression

Evaluation is a recursive function and must handle local and global scopes

$$\text{eval} : \mathbf{M}[\ \text{ParseTreeExpression}\ ]\ \text{x}\ \Sigma \longrightarrow (\ \mathbb{D}\ )$$

Result after parsing:

Each leaves in the parsing tree is an atomic component, with an attached semantic.

A semantic must define a **composition operator** and **hiding operator**.

The evaluation of the parsing tree returns a set of instances, where renaming, hiding operations and insertion of nodes are implemented:

main = () {
A^1_1 := green(a){ # } ,
A^1_2 := red(b){ # } ,
A^1_3 := blue(c){ # } ,
A^2_1 := fifo(b,c_1){ # } ,
A^2_2 := sync(a,b) { # } ,
A^2_3 := syncdrain(a,c_2) { # } ,
A^2_3 := merger(c_1,c_2,c) { # }
}

Set of instances of atomic components described by the semantic #

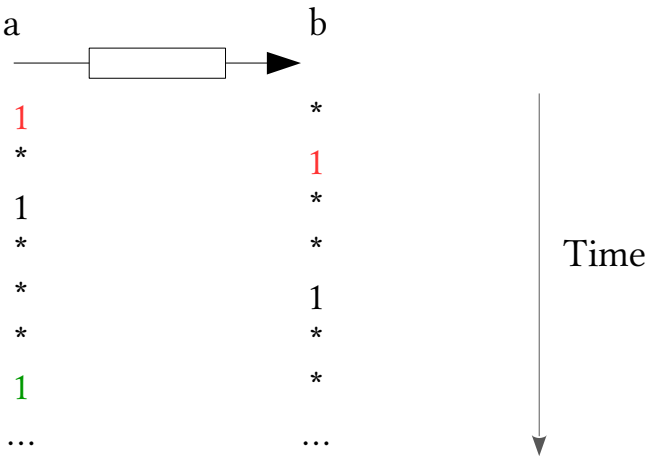
Given a semantic "#", how to interpret the resulting set of components into a composite behavior?

Rule Based Semantic:

Suppose we observe what happen at each ports: either you see a datum (1) or you do not see anything (*).

We assign for each ports its stream of observation.

How would you describe the relation on those streams?

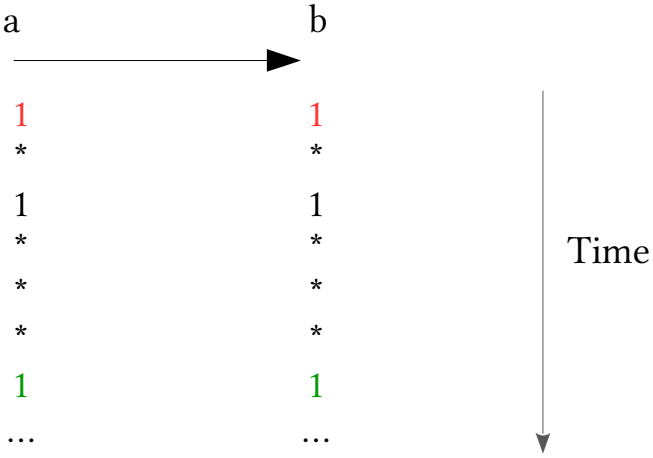


History:

Question investigated in 1948 : "A logical calculus of the idea immanent in nervous activity" by W. S. McCulloch and W. Pitts
and 1951: "Representation of events in nerve nets and finita automata" by S.C. Kleene.

Study of activation patterns in Nerve Nets: how output neurons firing is related to input neuron firing.

Leads to the definition of regular languages, Kleene star (as operation on set of tables) and beginning of machine learning..



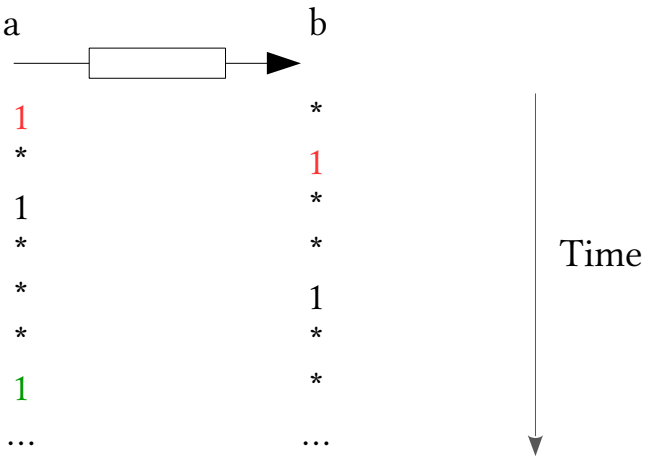
Rule Based:

$$fifo(a,b)\{ \quad \#RB$$

Rule1 $\quad (m' = a \wedge a \neq * \wedge b = * \wedge m = b) \vee$

Rule2 $\quad (m' = a \wedge a = * \wedge b \neq * \wedge m = b) \vee$

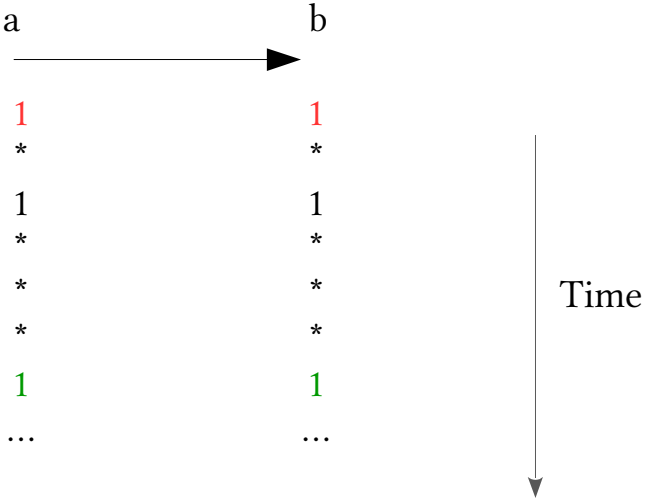
Rule3 $\quad (m' = m \wedge a = * \wedge b = *)\}$



A rule based formula describes a set of accepted stream on the port involved

$$sync(a,b)\{\#RB$$

Rule1 $\quad a = b\}$



Rule Based (formal):

Terms:

$$t \quad ::= \quad p \quad | \quad \underline{m} \quad | \quad \underline{m}' \quad | \quad f(t_1, ..., t_n) \quad | \quad *$$

Formulas:

$$\phi \quad ::= \quad \bot \quad | \quad t_1 = t_2 \quad | \quad Q(t_1, ..., t_n) \quad | \quad \neg\phi \quad | \quad \exists p\phi \quad | \quad \phi_1 \wedge \phi_2$$

Satisfiability relation given a model and an assignment map:

$$\begin{aligned}
\mathcal{M}, \gamma \models t_1 = t_2 &\quad \Leftrightarrow \quad t_1^{\mathcal{M}} = t_2^{\mathcal{M}}; \\
\mathcal{M}, \gamma \models Q(t_1, ..., t_n) &\quad \Leftrightarrow \quad (t_1^{\mathcal{M}}, ..., t_n^{\mathcal{M}}) \in Q^{\mathcal{M}}; \\
\mathcal{M}, \gamma \models \neg\phi &\quad \Leftrightarrow \quad \mathcal{M}, \gamma \not\models \phi; \\
\mathcal{M}, \gamma \models \exists p\phi &\quad \Leftrightarrow \quad \mathcal{M}, \gamma[p \mapsto x] \models \phi \text{ for some } x \in \Sigma; \\
\mathcal{M}, \gamma \models \phi_1 \wedge \phi_2 &\quad \Leftrightarrow \quad \mathcal{M}, \gamma \models \phi_1 \text{ and } \mathcal{M}, \gamma \models \phi_2.
\end{aligned}$$

Composition:

main(a,b,c) = {
A^1_1 := green(a){ #RB .. } ,
A^1_2 := red(b){ #RB .. } ,
A^1_3 := blue(c){ #RB .. } ,
A^2_1 := fifo(b,c_1){ #RB .. } ,
A^2_2 := sync(a,b) { #RB .. } ,
A^2_3 := syncdrain(a,c_2) { #RB .. } ,
A^2_3 := merger(c_1,c_2,c) { #RB .. }
}

$$main(a,b,c) = \phi_{green}(a) \wedge \phi_{red}(b) \wedge \phi_{blue}(c) \wedge \phi_{fifo}(b,c_1) \wedge \phi_{sync}(a,b) \wedge \phi_{syncdrain}(a,c_2) \wedge \phi_{merger}(c_1,c_2,c)$$

Each component is represented by a formula.

The **composition** of two components is represented by the **conjunction** of their respective formulas.

Hiding:

main = () {
A^1_1 := green(a){ #RB .. } ,
A^1_2 := red(b){ #RB .. } ,
A^1_3 := blue(c){ #RB .. } ,
A^2_1 := fifo(b,c_1){ #RB .. } ,
A^2_2 := sync(a,b) { #RB .. } ,
A^2_3 := syncdrain(a,c_2) { #RB .. } ,
A^2_3 := merger(c_1,c_2,c) { #RB .. }
}

$$main = \quad \exists a.b.c \quad \phi_{green}(a) \wedge \phi_{red}(b) \wedge \phi_{blue}(c) \wedge \phi_{fifo}(b, c_1) \wedge \phi_{sync}(a, b) \wedge \phi_{syncdrain}(a, c_2) \wedge \phi_{merger}(c_1, c_2, c)$$

Each open port is represented by a free variable.

Hiding a port bounds its corresponding variable with an existential quantifier.
No interaction is possible on this port

Formula and normal forms:

One particularity of logical formulas: different syntactical representation of the same object.

$$\phi_{main} = \bigwedge_{i,j} \phi_{A_{ij}}$$

Disjunctive normal form: get back the automaton

.

$$\phi_{main} = \bigvee_{i} \bigwedge_{j} r_{ij}$$ where r_{ij} is a rule from a component's formula

Conjunctive normal form: minimal representation of the system.

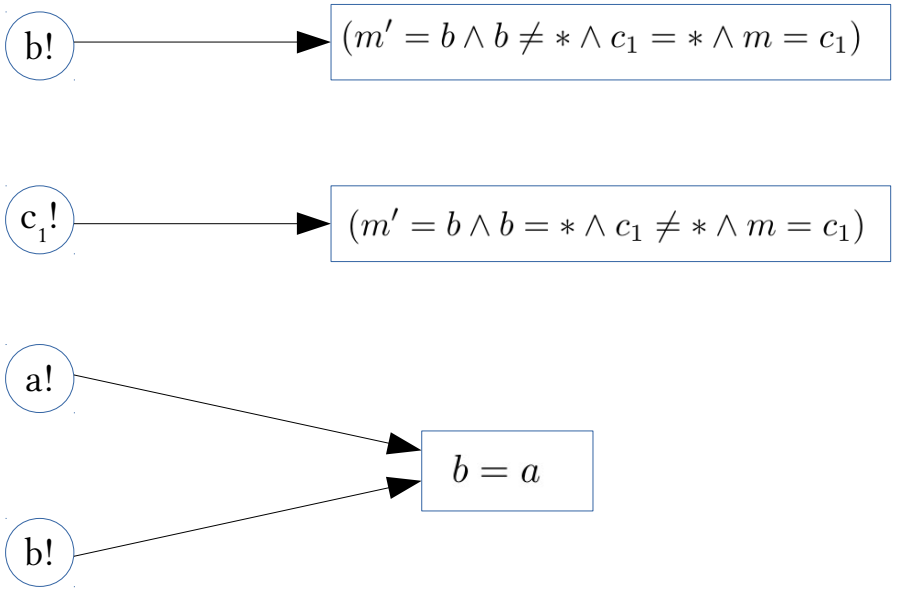
$$\phi_{main} = \bigwedge_{i} \bigvee_{j} r_{ij}$$ where r_{ij} is a rule from a component's formula

Conjunction of implication: if a port fires, then one of the rules in which the port is involved must be satisfied.

$$\phi_{main} = \bigwedge_{p} (p \neq * \rightarrow \bigvee_{p \in free(\phi)} \phi)$$

Formula and optimization:

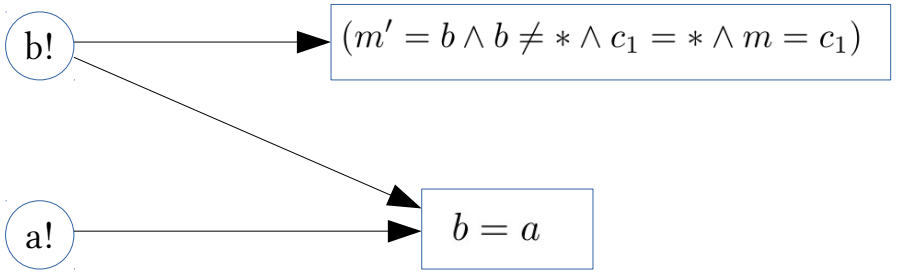
fifo(b,c_1)

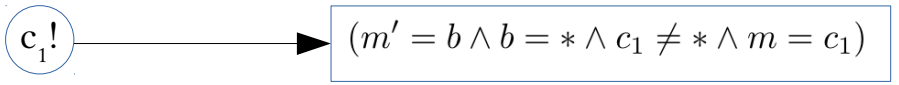


sync(a,b)

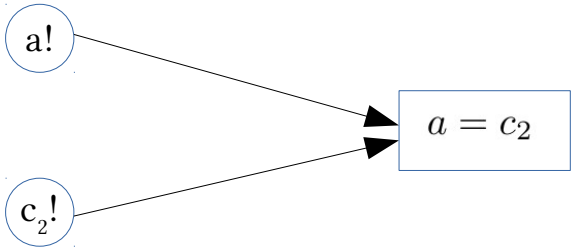
Formula and optimization:

fifo(b,c_1),
sync(a,b)





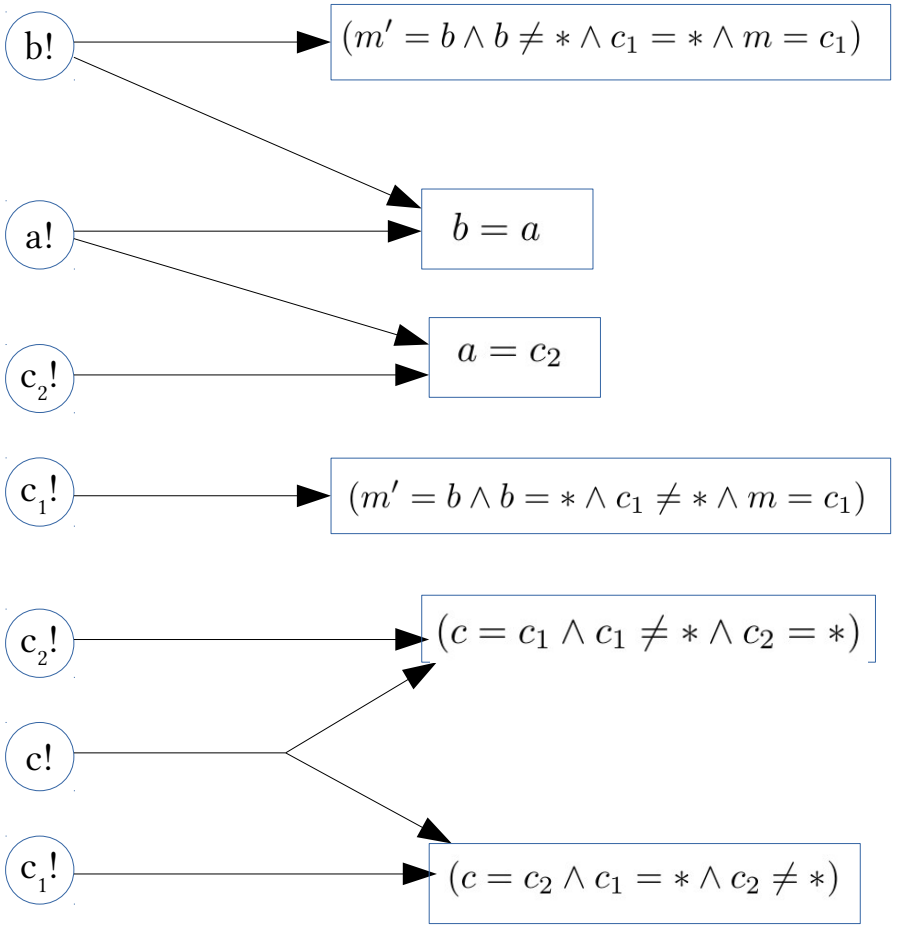
syncdrain(a,c_2)



Formula and optimization:

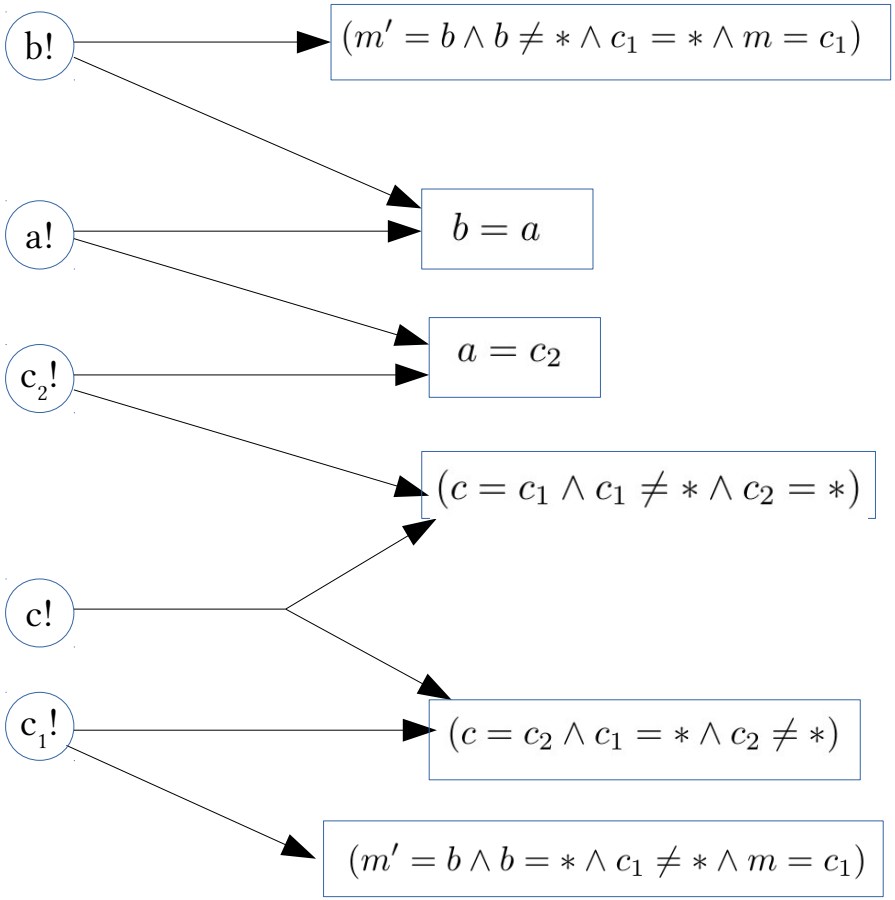
fifo(b,c_1),
sync(a,b),
syncdrain(a,c_2)

merger(c,c_1,c_2)



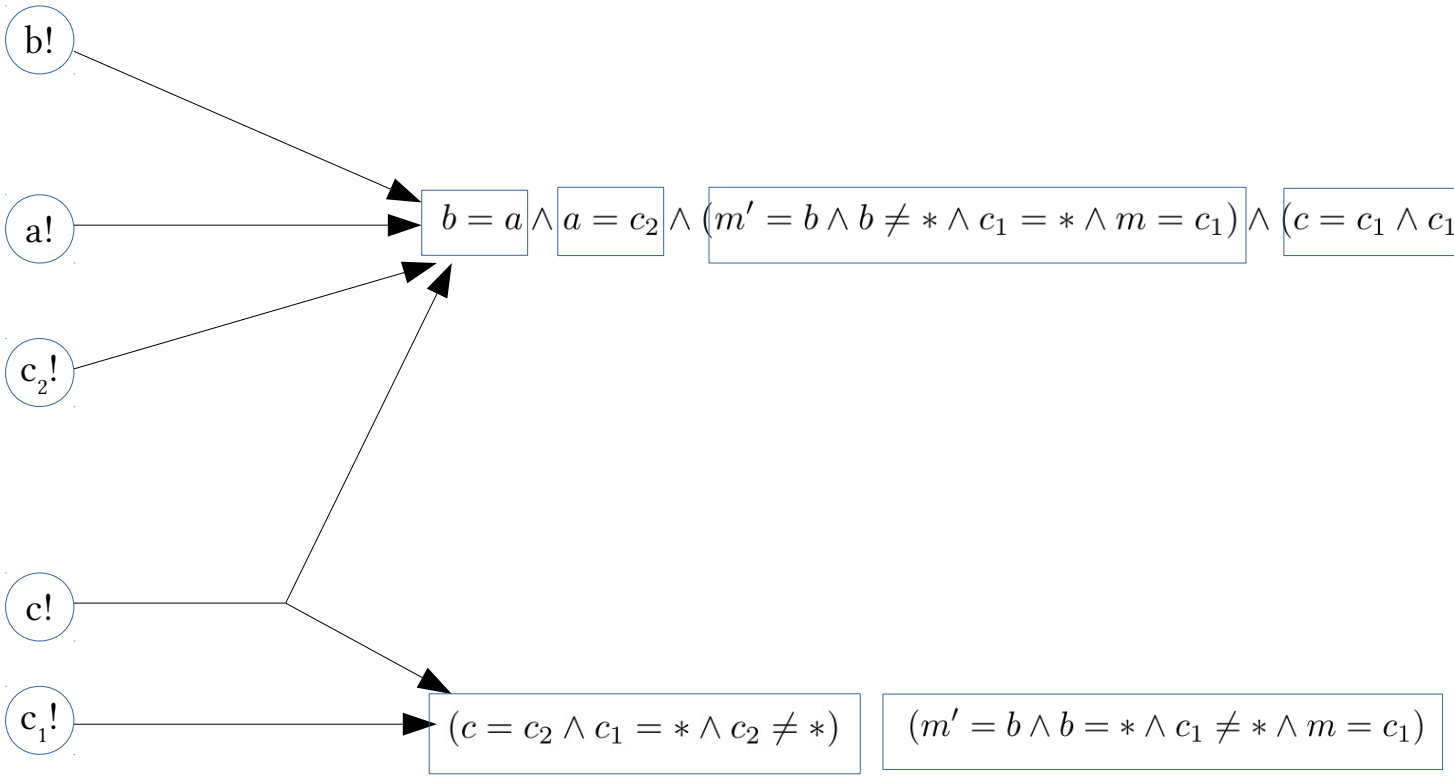
Formula and optimization:

fifo(b,c_1),
sync(a,b),
syncdrain(a,c_2),
merger(c,c_1,c_2)



Formula and optimization:

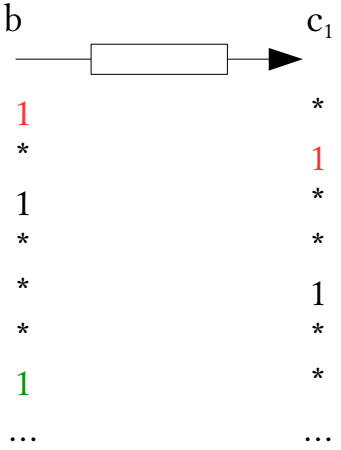
fifo(b,c_1),
sync(a,b),
syncdrain(a,c_2),
merger(c,c_1,c_2)



The minimal set of rule is used to build guarded commands.

Commandification:

Formula are direction-less: no specification of input and output, but only synchronization.

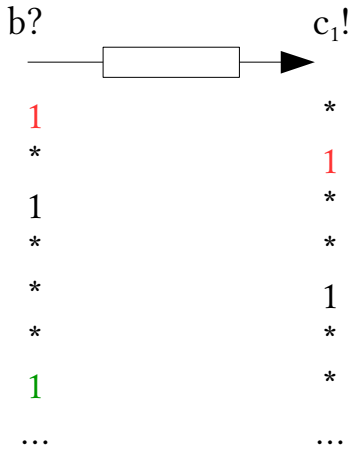


Implementation requires a direction for data flow. We assume each ports being either **input** or **output**.

Strategy:
Given an input/output specification for each port, we inspect the syntax of the formula to separate premise (guard) and conclusion (update).

Commandification:

Formula are direction-less: no specification of input and output, but only synchronization.



Implementation requires a direction for data flow. We assume each ports being either input or output.

Strategy:
Given an input/output specification for each port, we inspect the syntax of the formula to separate premise (guard) and conclusion (update).

	Guard	Command
$(m' = b \wedge b \neq * \wedge c_1 = * \wedge m = c_1)$	$\{m = *, b \neq *\}$	$\{m' = b\}$
$(m' = b \wedge b = * \wedge c_1 \neq * \wedge m = c_1)$	$\{m \neq *, c_1 \neq *\}$	$\{c_1 = m, m' = b\}$

Java target language:

A guarded command have an intuitive interpretation as if statement in Java.

For each guarded command (G,C), where G is the guard and C the command, generate the following set of statement:

$$\text{if}(\bigwedge_{g \in G} g)\{ \quad c_1; \quad c_2; \quad \ldots \quad c_N; \}$$

where $\{c_1,\ldots,c_N\}$ = C

Show semantic preserving:

Is time data stream behavior equivalent to Java execution?

Java runtime: port model

Fields

```
/** The prod. */
private Component prod;

/** The cons. */
private Component cons;
```

```
/** The put. */
private volatile T put;

/** The get. */
private volatile boolean get;
```

Put and get methods

```
public T get() {
        get = true;
        prod.activate();
        if (put == null)
                synchronized (cons) {
                        while (put == null)
                                try {
                                cons.wait();
                                } catch (InterruptedException e) {
                                }
                }
        T datum = put;
        put = null;
        get = false;
        prod.activate();
        return datum;
}
```

```
public void put(T datum) {
        if (datum == null)
                throw new NullPointerException();
        while (put != null) {
        }
        put = datum;
        cons.activate();
        if (!get && put != null)
                synchronized (prod) {
                        while (!get && put != null)
                                try {
                                prod.wait();
                                } catch (InterruptedException e) {
                                }
                }
}
```

Java protocol skeleton:

Each port is in a separated thread and is linked to a producer and consumer component.

The system is assumed to be closed: each port has a consumer and producer defined.

The protocol is defined in a thread and act as producer or consumer for ports in its interface.

```
public static void main(String[] args) {

    Port<String> $2 = new PortWaitNotify<String>();
…

    PortWindow1 PortWindow1 = new PortWindow1();
    $2.setProducer(PortWindow1);
    PortWindow1.$2 = $2;
…

    Protocol1 Protocol1 = new Protocol1();
    $2.setConsumer(Protocol1);
    Protocol1.$2 = $2;

    Thread thread_Protocol1 = new Thread(Protocol1);
    thread_PortWindow1.start();
…
    try {
        thread_PortWindow1.join();
        thread_Protocol1.join();
…
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

```
private static class PortWindow1 implements Component {

    public volatile Port<String> $2;


    public void activate() {
        synchronized (this) {
            notify();
        }
    }

    public void run() {
        Windows.producer("a", $2);
    }
}
```

Java protocol skeleton:

```
private static class Protocol1 implements Component {

    public volatile Port<String> $2;
…
    public void activate() {
        synchronized (this) {
            notify();
        }
    }

    public void run() {


    while (true) {
        if ($4.hasGet() && !(m1 == null)) {
            $4.put(m1);
            m1 = null;
        }
        if ($4.hasGet() && !($2.peek() == null) &&  !($3.peek() == null) && (!($2.peek() == null) && m1 == null && !($3.peek() == null))) {
            $4.put($2.peek());
            m1 = $3.peek();
            $2.get();
            $3.get();
       }
        synchronized (this) {
            while(true) {
                if ($4.hasGet() && !(m1 == null)) break;
                if ($4.hasGet() && !($2.peek() == null) &&  !($3.peek() == null) && (!($2.peek() == null) && m1 == null && !($3.peek() == null)))
break;
                try {
                    wait();
                } catch (InterruptedException e) { }
            }
        }
       }
```