# Reo Compiler

Benjamin Lion
lion@cwi.nl

# On Reo

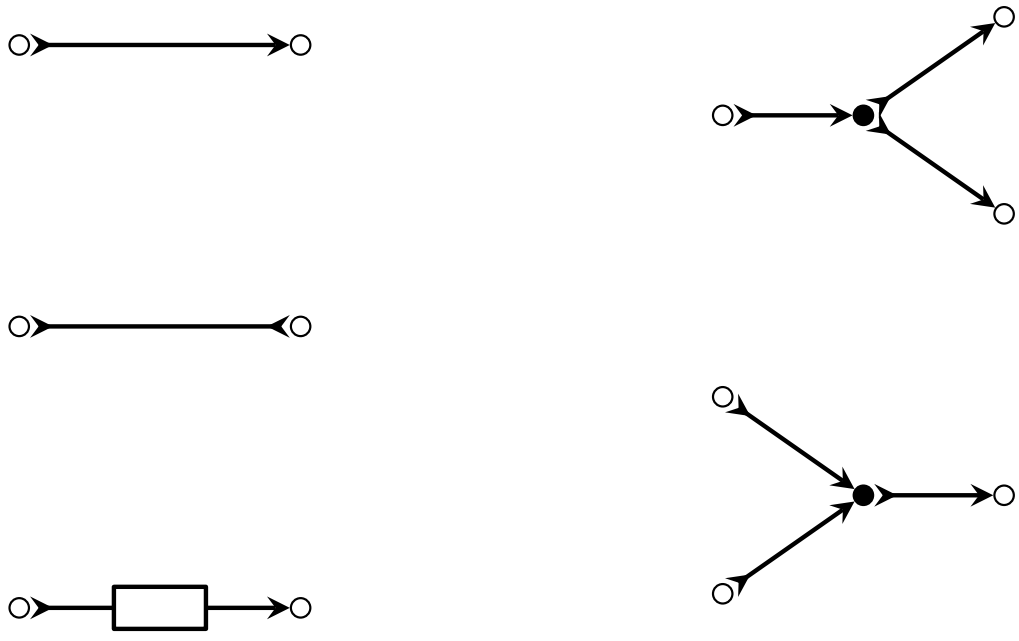
Quick recap:

- Some Reo primitives
- Their behavior.
- Formal semantics.
- Relationship between semantics.

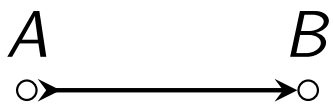
Compiling Treo:

- From Treo to a formal representation
- Reduction.
- From the reduced form to an output language.

# Some Reo primitives



# Behavior for primitives



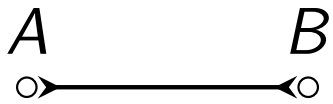

| A | B |
|---|---|
| $d_1$ | $d_1$ |
| $d_2$ | $d_2$ |
| $d_3$ | $d_3$ |
| $*$ | $*$ |
| ... | ... |

| A | B |
|---|---|
| $*$ | $*$ |
| $d_2$ | $d_2$ |
| $d_1$ | $d_1$ |
| $*$ | $*$ |
| ... | ... |

...

$d_i$: data element, and $*$: absence of data.

# Behavior for primitives



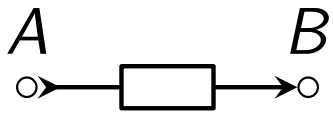

| A | B |
|---|---|
| $d_1$ | $d_2$ |
| $*$ | $*$ |
| $d_3$ | $d_4$ |
| $*$ | $*$ |
| ... | ... |

| A | B |
|---|---|
| $*$ | $*$ |
| $d_2$ | $d_2$ |
| $d_1$ | $d_1$ |
| $*$ | $*$ |
| ... | ... |

...

$d_i$: data element, and $*$: absence of data.

# Behavior for primitives



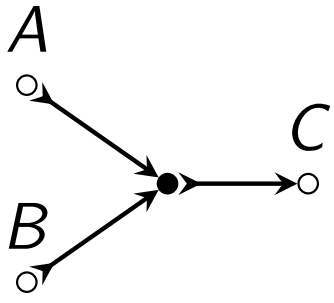

| A | B |
|---|---|
| $d_1$ | $*$ |
| $*$ | $d_1$ |
| $*$ | $*$ |
| $d_2$ | $*$ |
| ... | ... |

| A | B |
|---|---|
| $*$ | $*$ |
| $d_2$ | $*$ |
| $*$ | $*$ |
| $*$ | $d_2$ |
| ... | ... |

...

$d_i$: data element, and $*$: absence of data.

# Behavior for primitives



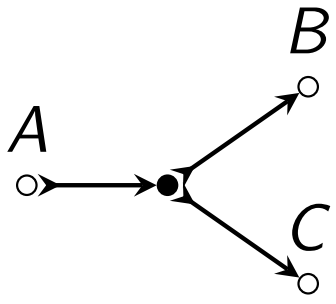

| A | B | C |
|---|---|---|
| $*$ | $*$ | $*$ |
| $d_2$ | $*$ | $d_2$ |
| $*$ | $d_1$ | $d_1$ |
| $*$ | $d_2$ | $d_2$ |
| ... | ... | |

| A | B | C |
|---|---|---|
| $d_1$ | $*$ | $d_1$ |
| $d_2$ | $*$ | $d_2$ |
| $*$ | $d_3$ | $d_3$ |
| $*$ | $d_4$ | $d_4$ |
| ... | ... | ... |

...

$d_i$: data element, and $*$: absence of data.

# Behavior for primitives



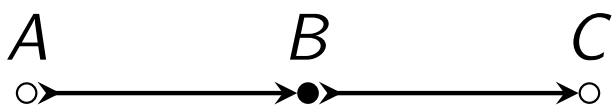

| A | B | C |
|---|---|---|
| $*$ | $*$ | $*$ |
| $d_2$ | $d_2$ | $d_2$ |
| $d_1$ | $d_1$ | $d_1$ |
| $d_2$ | $d_2$ | $d_2$ |
| ... | ... | |

| A | B | C |
|---|---|---|
| $d_1$ | $d_1$ | $d_1$ |
| $d_2$ | $d_2$ | $d_2$ |
| $*$ | $*$ | $*$ |
| $d_4$ | $d_4$ | $d_4$ |
| ... | ... | ... |

...

$d_i$: data element, and $*$: absence of data.

# Composition




| A | B |
|---|---|
| $d_1$ | $d_1$ |
| $d_2$ | $d_2$ |
| $d_3$ | $d_3$ |
| $*$ | $*$ |
| $\dots$ | $\dots$ |

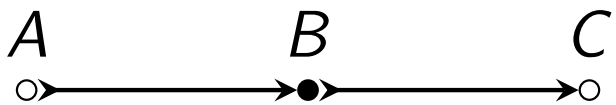
$\bowtie$

| B | C |
|---|---|
| $d_1$ | $d_1$ |
| $d_2$ | $d_2$ |
| $d_3$ | $d_3$ |
| $*$ | $*$ |
| $\dots$ | $\dots$ |

$=$

| A | B | C |
|---|---|---|
| $d_1$ | $d_1$ | $d_1$ |
| $d_2$ | $d_2$ | $d_2$ |
| $d_3$ | $d_3$ | $d_3$ |
| $*$ | $*$ | $*$ |
| $\dots$ | $\dots$ | $\dots$ |

# Composition



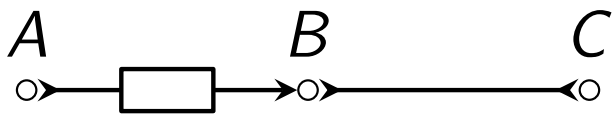

| A | B |
|---|---|
| $d_1$ | $d_1$ |
| $d_2$ | $d_2$ |
| $d_3$ | $d_3$ |
| $*$ | $*$ |
| $\dots$ | $\dots$ |

$\bowtie$

| B | C |
|---|---|
| $d_2$ | $d_2$ |
| $d_2$ | $d_2$ |
| $d_3$ | $d_3$ |
| $*$ | $*$ |
| $\dots$ | $\dots$ |

$= \emptyset$

# Composition




| A | B |
|---|---|
| $d_1$ | $*$ |
| $*$ | $d_1$ |
| $*$ | $*$ |
| $d_2$ | $*$ |
| $\ldots$ | $\ldots$ |

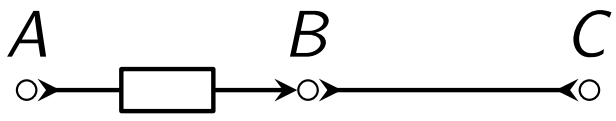
$\bowtie$

| B | C |
|---|---|
| $*$ | $*$ |
| $d_1$ | $d_3$ |
| $*$ | $*$ |
| $*$ | $*$ |
| $\ldots$ | $\ldots$ |

$=$

| A | B | C |
|---|---|---|
| $d_1$ | $*$ | $*$ |
| $*$ | $d_1$ | $d_3$ |
| $*$ | $*$ | $*$ |
| $d_2$ | $*$ | $*$ |
| $\ldots$ | $\ldots$ | $\ldots$ |

# Composition



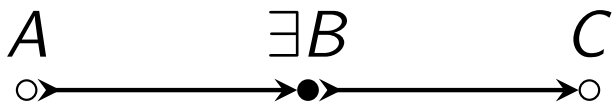

| A | B |
|---|---|
| $d_1$ | $*$ |
| $*$ | $d_1$ |
| $*$ | $*$ |
| $d_2$ | $*$ |
| $\ldots$ | $\ldots$ |

$\bowtie$

| B | C |
|---|---|
| $*$ | $*$ |
| $d_4$ | $d_3$ |
| $*$ | $*$ |
| $*$ | $*$ |
| $\ldots$ | $\ldots$ |

$= \emptyset$

# Hiding

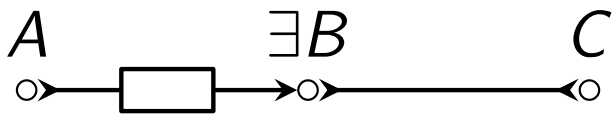



| A | ∃B | C |
|---|---|---|
| $d_1$ | $d_1$ | $d_1$ |
| $d_2$ | $d_2$ | $d_2$ |
| $d_3$ | $d_3$ | $d_3$ |
| $*$ | $*$ | $*$ |
| $\dots$ | $\dots$ | $\dots$ |

$=$

| A | C |
|---|---|
| $d_1$ | $d_1$ |
| $d_2$ | $d_2$ |
| $d_3$ | $d_3$ |
| $*$ | $*$ |
| $\dots$ | $\dots$ |

# Hiding




| A | ∃ B | C |
|---|---|---|
| $d_1$ | $*$ | $*$ |
| $*$ | $d_1$ | $d_3$ |
| $*$ | $*$ | $*$ |
| $d_2$ | $*$ | $*$ |
| ... | ... | ... |

$=$

| A | C |
|---|---|
| $d_1$ | $*$ |
| $*$ | $d_3$ |
| $*$ | $*$ |
| $d_2$ | $*$ |
| ... | ... |

# What we want:

1. Make the notion of data constraint precise.

2. Make the notion of composition and hiding precise.

3. So that a machine can compute the result.

4. Verification, informal proof.

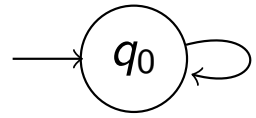
5. Code generated can be used as protocol.

# Where to look?

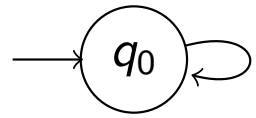
1. Automaton based specification: primitives as constraint automata.[**?**]

2. Logical specification: primitives as predicates.[**?**]

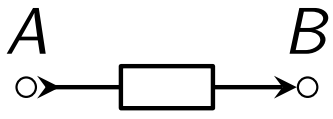
# Constraint automata


$d_A \neq * \wedge d_A = d_B$

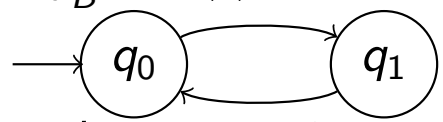




$d_A \neq * \wedge d_B \neq *$




$d_A \neq * \wedge m^\bullet = d_A \wedge$

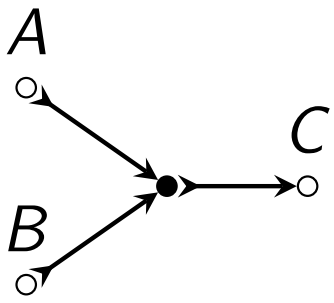
$d_B = * \wedge^\bullet m = *$

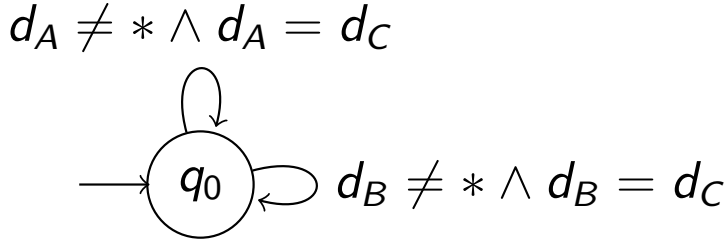


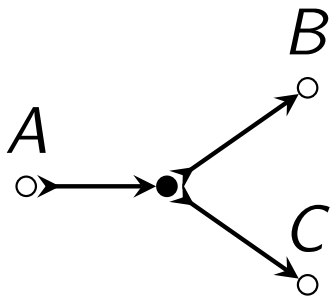

$d_A = * \wedge m^\bullet = * \wedge$

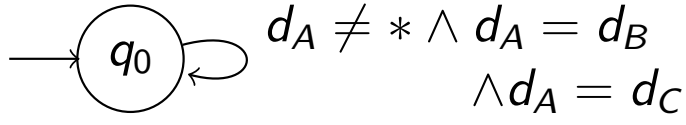
$d_B =^\bullet m \wedge^\bullet m \neq *$

# Constraint automata



# Constraints

$DC(D, V \cup M)$ is the set of guards $g$ defined over the data elements $d \in D$, port variables $v \in V$ and memory variables $m \in M$, such that:

$$t ::= d \mid v \mid^{\bullet} m \mid m^{\bullet} \mid *$$

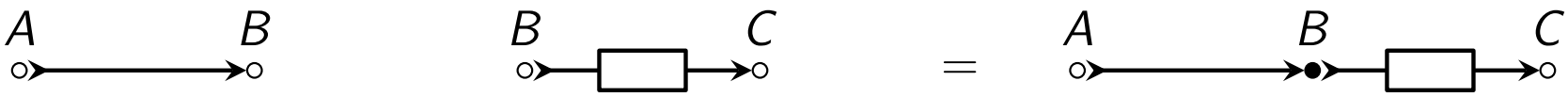
$$g ::= t_1 = t_2 \mid t_1 \neq t_2 \mid g_1 \wedge g_2 \mid \exists v.g$$

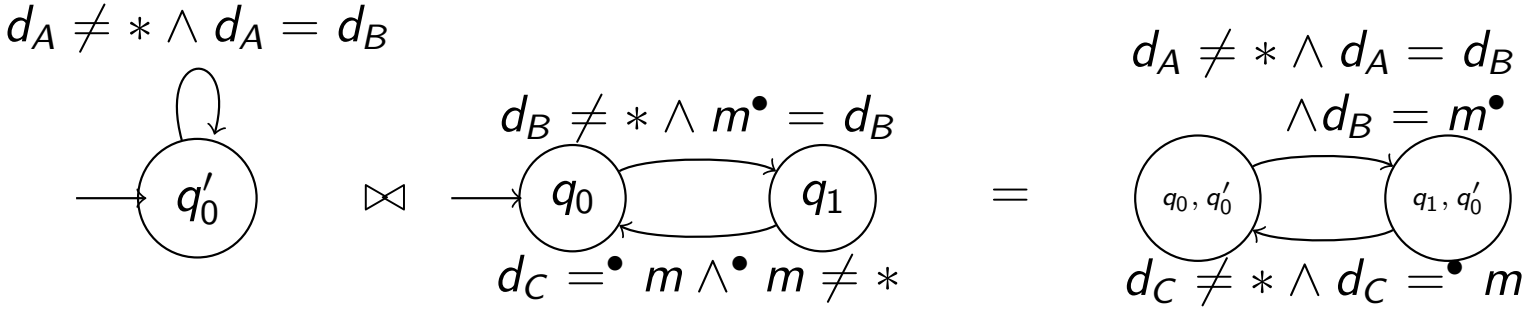
# Constraint automata

$$\mathcal{A} = (Q, \mathcal{V}, \rightarrow, Q_0)$$

$Q$ : set of states.
$\mathcal{V}$ : set of ports and memories.
$\rightarrow$ : subset of $Q \times DC(D, V \cup M) \times Q$.
$Q_0$ : set of initial states, subset of $Q$.

# Constraint automata (composition)



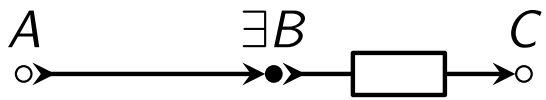
# Constraint automata (composition)

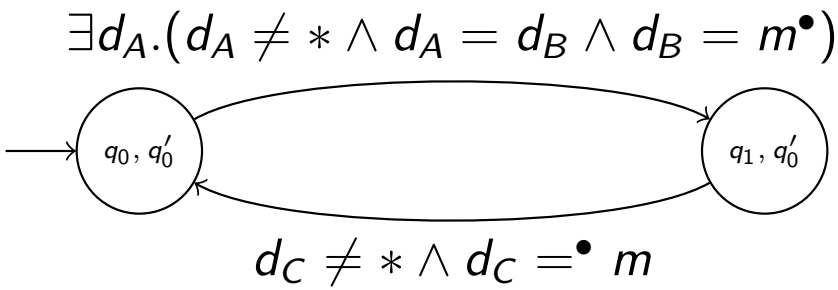
$\mathcal{A}_1 = (Q_1, \mathcal{P}_1, \rightarrow_1, Q_{10})$ and $\mathcal{A}_2 = (Q_2, \mathcal{P}_2, \rightarrow_2, Q_{20})$

$$\mathcal{A}_1 \bowtie \mathcal{A}_2 = (Q_1 \times Q_2, \mathcal{P}_1 \cup \mathcal{P}_2, \rightarrow, Q_{10} \cup Q_{20})$$

where, given $q_1 \xrightarrow{g_1} p_1 \in \rightarrow_1$ and $q_2 \xrightarrow{g_2} p_2 \in \rightarrow_2$,
$(q_1, q_2) \xrightarrow{g_1 \wedge g_2} (p_1, p_2) \in \rightarrow$.

# Constraint automata (hiding)



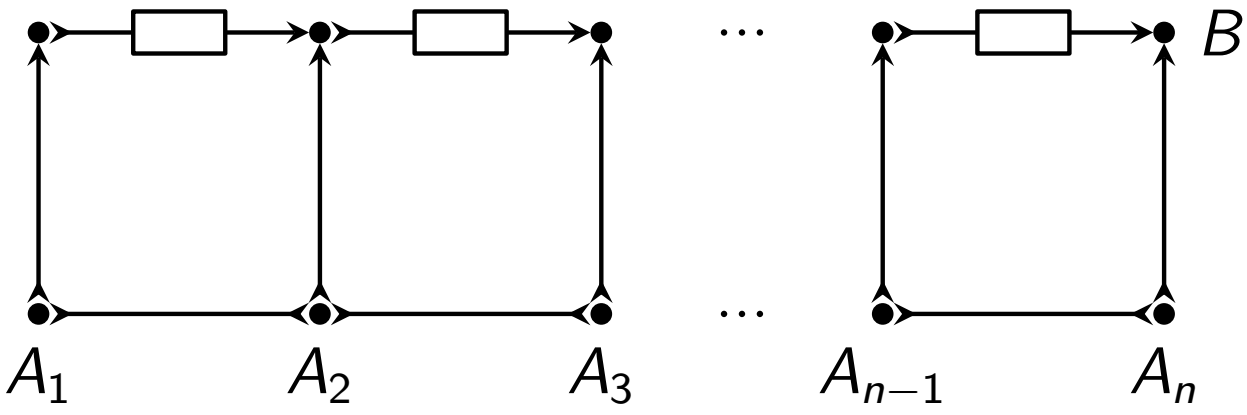



# Constraint automata (limitations)

State space explosion for some compositions:

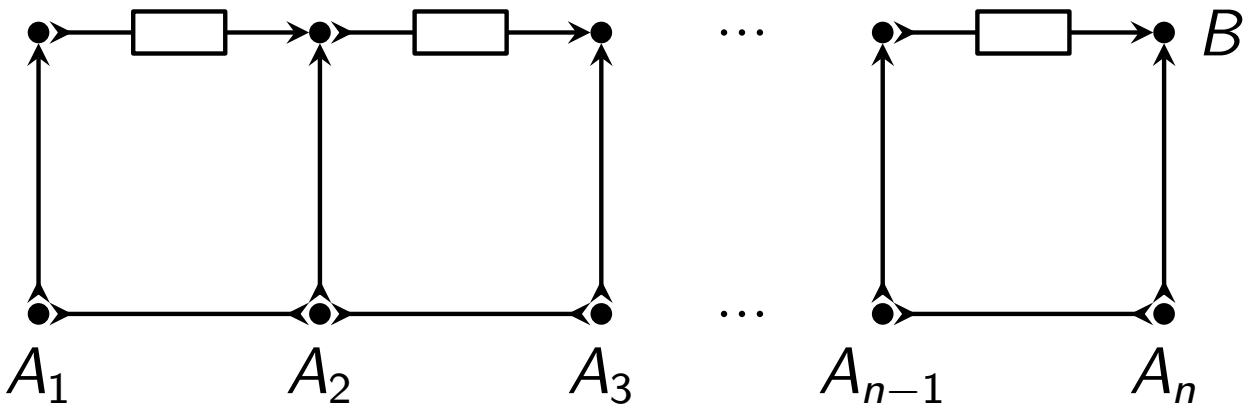
# Constraint automata (limitations)

State space explosion for some compositions:



# Constraint automata (limitations)

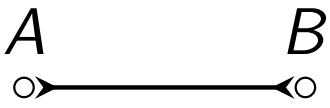
State space explosion for some compositions:



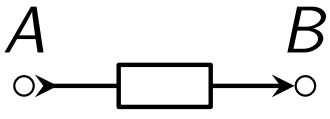

Assumption that every states have a silent transition.

# Logical specification


$$\forall t.A(t) = B(t)$$

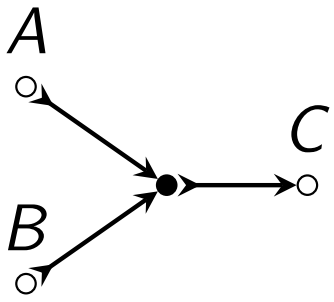



$$\forall t.(A(t) \neq * \wedge B(t) \neq * \\ A(t) = * \wedge B(t) = *)$$

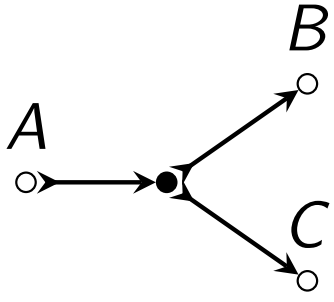



$$\forall t.(A(t) \neq * \wedge m^{\bullet} = A(t) \wedge B(t) = * \wedge^{\bullet} m = * \vee \\ A(t) = * \wedge m^{\bullet} = * \wedge B(t) \neq * \wedge^{\bullet} m = B(t) \vee \\ A(t) = * \wedge m^{\bullet} =^{\bullet} m \wedge B(t) = *)$$

# Logical specification




$$\forall t.(A(t) \neq * \wedge A(t) = C(t) \wedge B(t) = *$$
$$B(t) \neq * \wedge B(t) = C(t) \wedge A(t) = *$$
$$B(t) = * \wedge A(t) = * \wedge C(t) = *)$$




$$\forall t.(A(t) \neq * \wedge A(t) = B(t) \wedge A(t) = C(t)$$
$$A(t) = * \wedge B(t) = * \wedge C(t) = *)$$

# Logic

$DC(D, V \cup M)$ is the set of guards $g$ defined over the data elements $d \in D$, port stream variables $A \in V$ and memory variables $m \in M$, such that:

$$t ::= A(n) \mid d \mid^{\bullet} m \mid m^{\bullet} \mid *$$

$$\phi ::= t_1 = t_2 \mid t_1 \neq t_2 \mid \phi_1 \wedge \phi_2 \mid \exists A.\phi \mid \forall n.g \mid \phi_1 \vee \phi_2$$

# Logical specification (operations)

*Composition* is conjunction: $\phi(c_1 \circ c_2) = \phi(c_1) \wedge \phi(c_2)$, where $c_1$ and $c_2$ are two Reo components.

*Hiding* is existential quantification: $\phi(\exists A.c) = \exists A.\phi(c)$, where $c$ is a Reo components and $A$ is a port.

# Relationship between both semantics

Every constraint automata can be encoded as a formula in the logical specification.
Can we express more than regular languages?

# Synthesis

1. For any Reo circuit, there exists a constraint automaton.
For any constraint automaton, there exists a Reo circuit.

# Synthesis

1. For any Reo circuit, there exists a constraint automaton. For any constraint automaton, there exists a Reo circuit.

2. For any Reo circuit, there exists a logical formula. Is the converse true? (Open question)

# Treo (Composite component)

Ports in the same set *synchronize*.
Ports not present in the interface are *hidden*.

```
import alternator;
import green;
import red;
import blue;

main() {
    alternator(a,b,c)
    green(a)
    blue(b)
    red(c)
}
```

# Treo (Composite component)

Components can be nested in Treo.

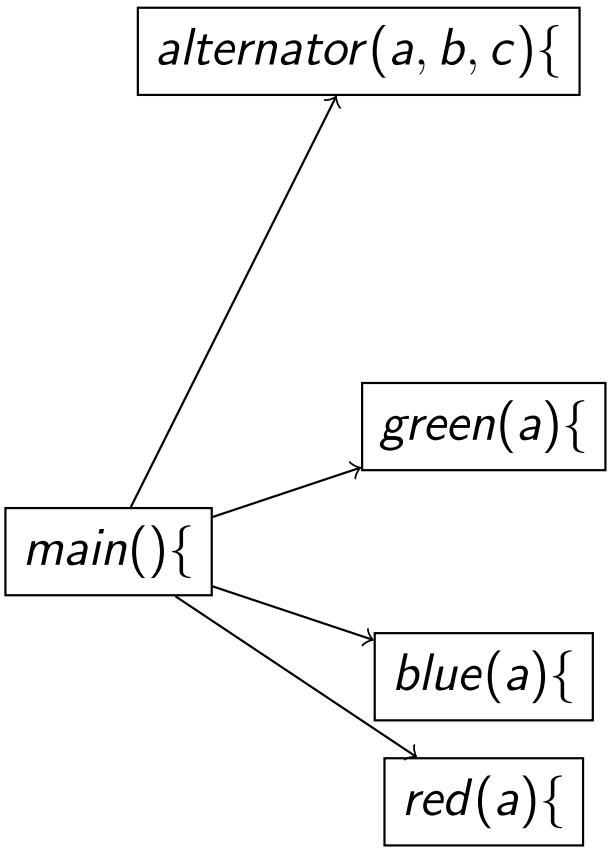
```
main() {
    alternator(a,b,c){
        sync(a,b){ #? }
        syncdrain(a,c) { #? }
        fifo(c,b) { #? }
    }
    green(a){ #? }
    blue(b){ #? }
    red(c){ #? }
}
```

# Treo (Atomic component)

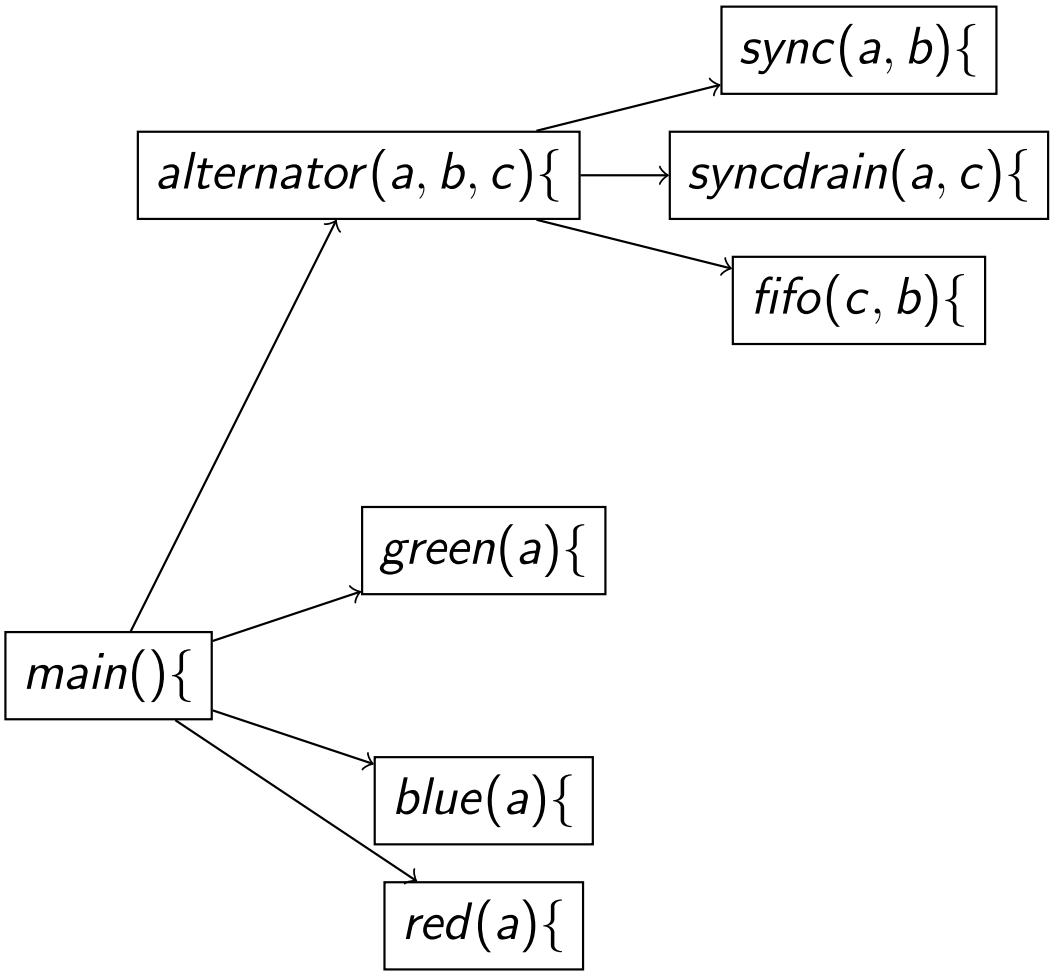
Atomic components have their own semantic.

```
sync(a,b){
    #CA
    q0 -> q1 : a!= *, a=b
}

syncdrain(a,c) {
    #CA
    q0 -> q1 : a!=*, b!=*
}

green(a){
    #Java
    'MyFunction.green'
}
```

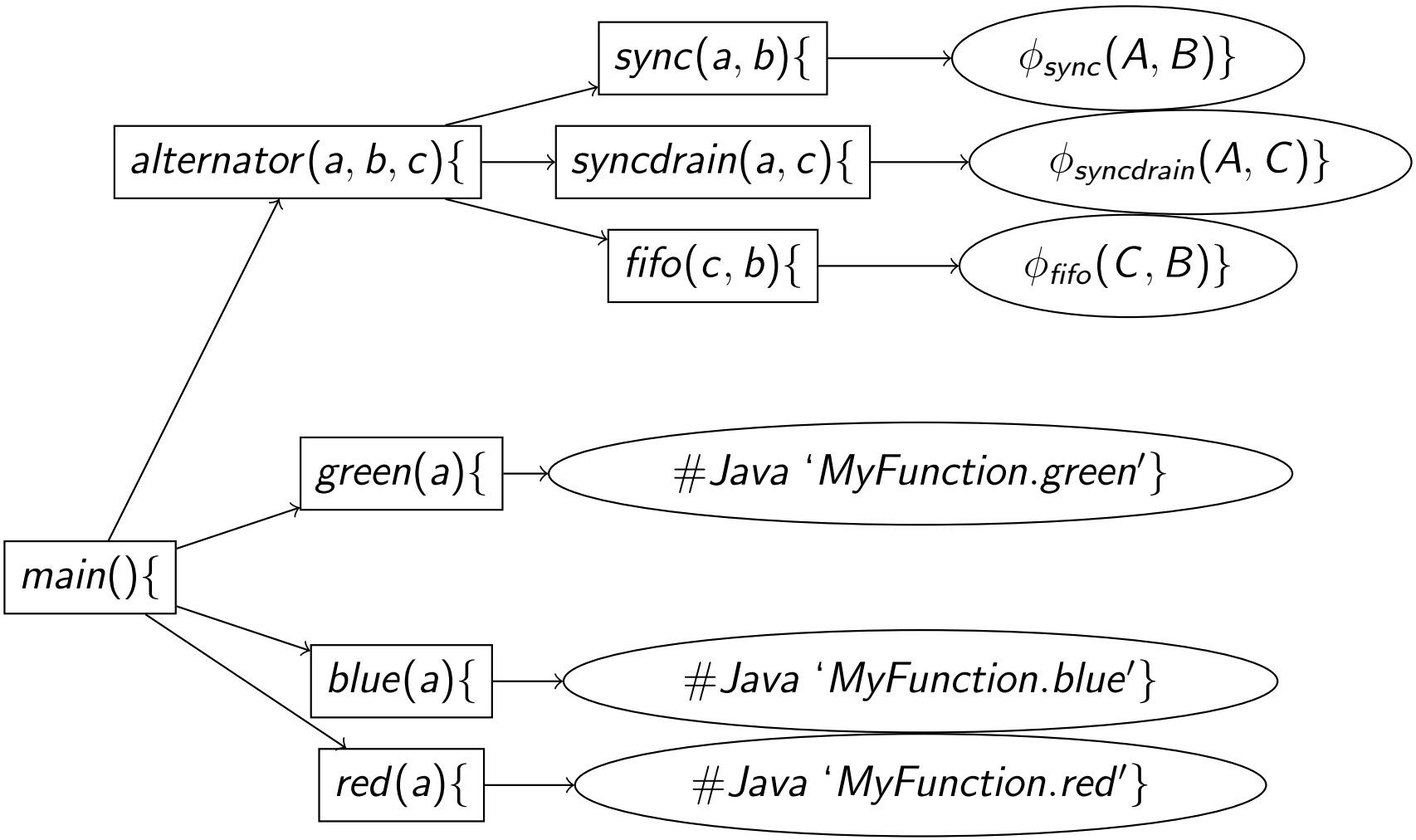
# Parse tree



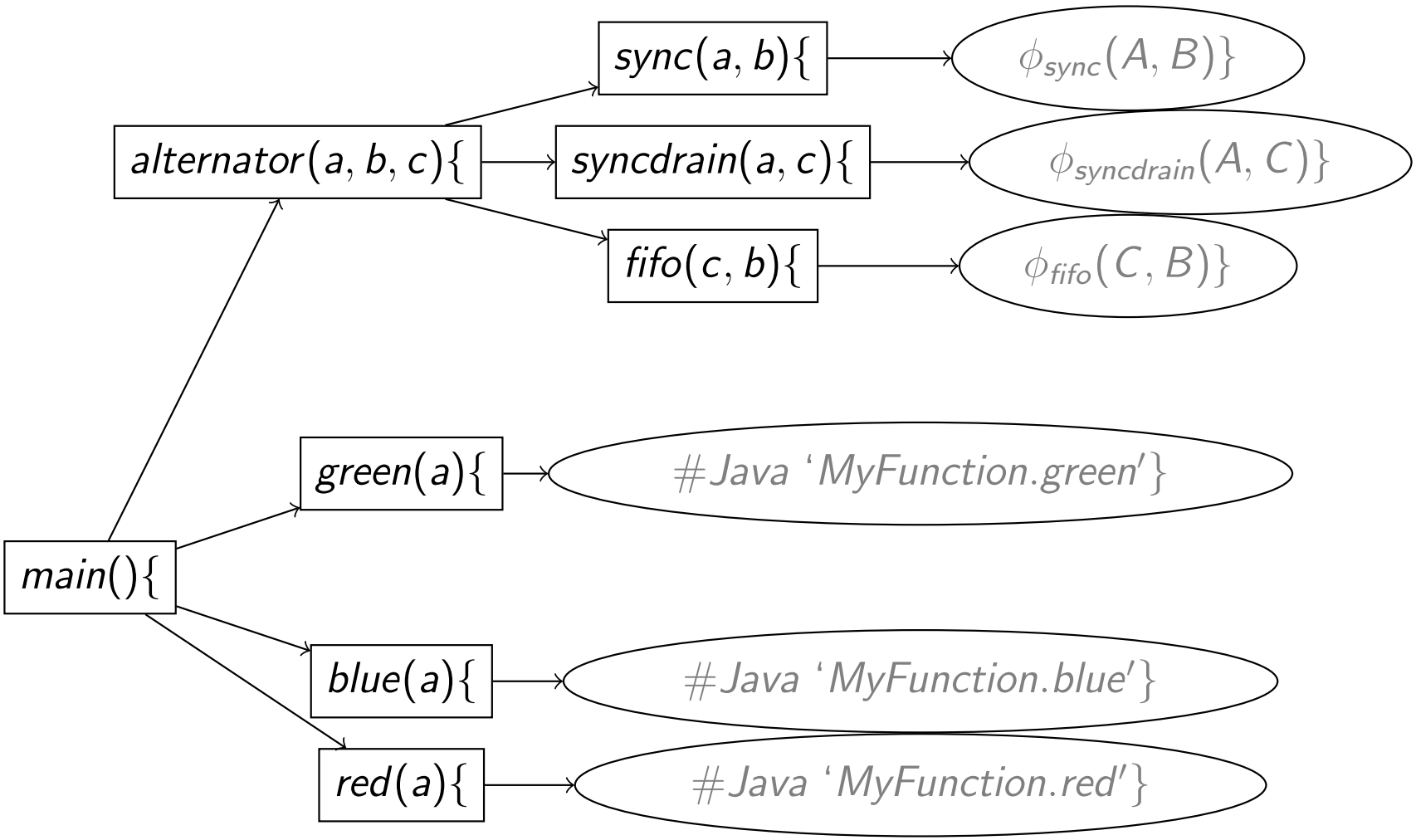
# Parse tree


sync(a, b){
alternator(a, b, c){
syncdrain(a, c){
fifo(c, b){
green(a){
main(){
blue(a){
red(a){

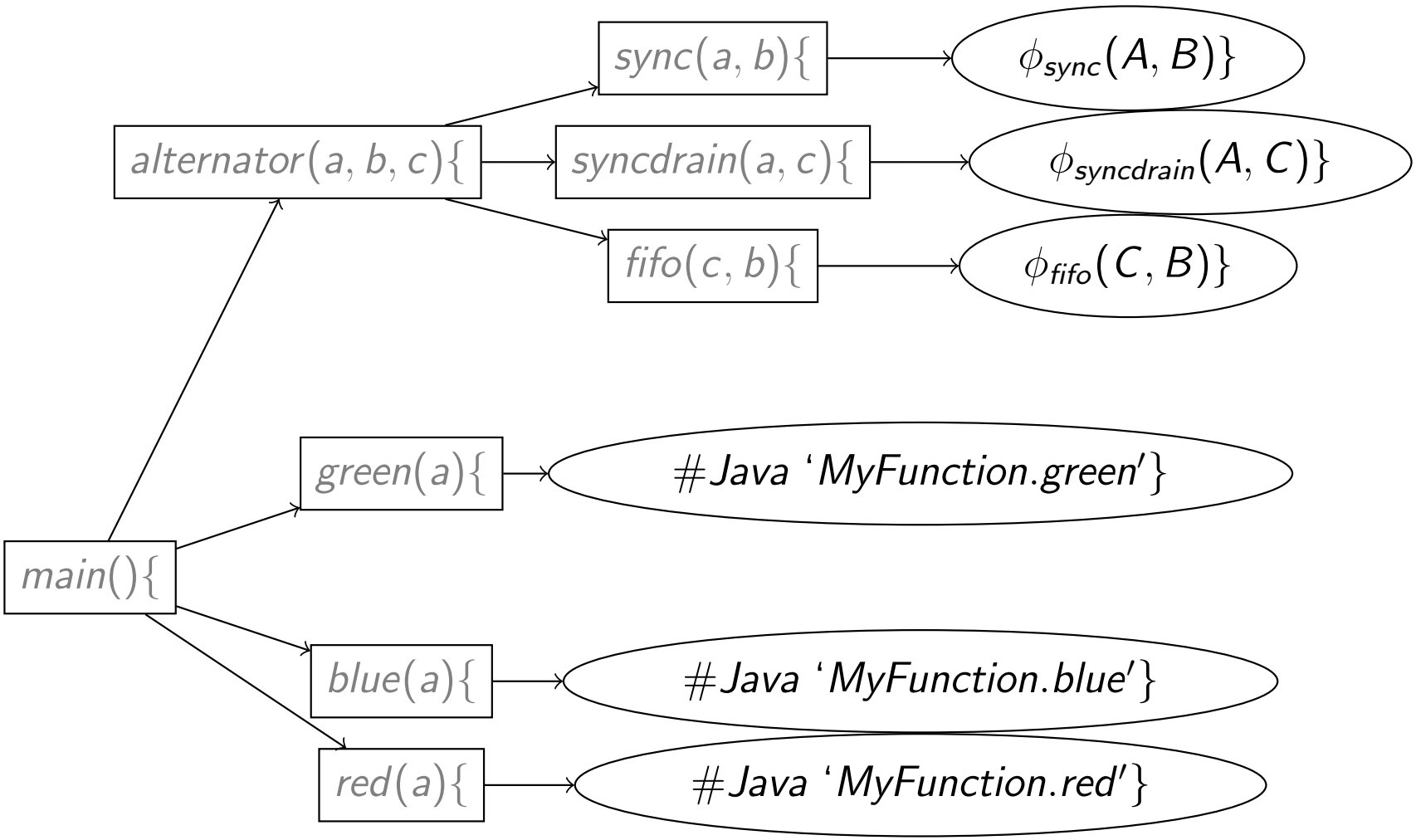
# Parse tree


sync(a, b){
$\phi_{sync}(A, B)\}$
alternator(a, b, c){
syncdrain(a, c){
$\phi_{syncdrain}(A, C)\}$
fifo(c, b){
$\phi_{fifo}(C, B)\}$
green(a){
#Java 'MyFunction.green'}
main(){
blue(a){
#Java 'MyFunction.blue'}
red(a){
#Java 'MyFunction.red'}

# Protocol


sync(a, b){
φsync(A, B)}
alternator(a, b, c){
syncdrain(a, c){
φsyncdrain(A, C)}
fifo(c, b){
φfifo(C, B)}
green(a){
#Java 'MyFunction.green'}
main(){
blue(a){
#Java 'MyFunction.blue'}
red(a){
#Java 'MyFunction.red'}

# Composition



# Flattening

Instantiation of all definition, and renaming of ports according to the protocol definition.

```
main() {
    sync(a,b){ #? }
    syncdrain(a,c) { #? }
    fifo(c,b) { #? }

    green(a){ #? }
    blue(b){ #? }
    red(c){ #? }
}
```

$$\begin{aligned}\phi_{main} = &\phi_{sync}(A, B) \wedge \\ &\phi_{syncdrain}(A, C) \wedge \\ &\phi_{fifo}(C, B) \wedge \\ &\phi_{green}(A) \wedge \\ &\phi_{blue}(B) \wedge \\ &\phi_{red}(C)\end{aligned}$$

# Node insertion

Insertion of a merger for a port used twice as input.
Insertion of a replicator for a port used twice as output.

```
main() {
sync(a,b){ #? }
syncdrain(a,c) { #? }
fifo(c,b) { #? }

green(a){ #? }
blue(b){ #? }
red(c){ #? }
}
```

$$\phi_{main} = \phi_{sync}(A, B_1) \wedge$$
$$\phi_{syncdrain}(A, C) \wedge$$
$$\phi_{fifo}(C, B_2) \wedge$$
$$\phi_{merger}(B_1, B_2, B) \wedge$$
$$\phi_{green}(A) \wedge$$
$$\phi_{blue}(B) \wedge$$
$$\phi_{red}(C)$$

# Hiding

Hide all ports except those in the interface of main().

```
main() {
sync(a,b){ #? }
syncdrain(a,c) { #? }
fifo(c,b) { #? }

green(a){ #? }
blue(b){ #? }
red(c){ #? }
}
```

$$\begin{aligned}\phi_{main} = &\exists A.\exists B.\exists B_1.\exists B_2.\exists C.\\ &(\phi_{sync}(A, B_1) \wedge\\ &\phi_{syncdrain}(A, C) \wedge\\ &\phi_{fifo}(C, B_2) \wedge\\ &\phi_{merger}(B_1, B_2, B) \wedge\\ &\phi_{green}(A) \wedge\\ &\phi_{blue}(B) \wedge\\ &\phi_{red}(C))\end{aligned}$$

# Composition

Disjunctive normal form:

$$\phi_{main} = \bigvee_i \phi_i \qquad \text{where } \phi_i \text{ are conjuncts.}$$

Conjunctive normal form:

$$\phi_{main} = \bigwedge_i \phi_i \qquad \text{where } \phi_i \text{ are disjuncts.}$$

# Composition

Disjunctive normal form:

$$\phi_{main} = \bigvee_i \phi_i \qquad \text{where } \phi_i \text{ are conjuncts.}$$

Conjunctive normal form:

$$\phi_{main} = \bigwedge_i \phi_i \qquad \text{where } \phi_i \text{ are disjuncts.}$$

Commandification:

$$\phi_{main} = \bigwedge_{P \in \mathcal{P}} (P(t) \neq * \implies \bigvee_i \phi_i) \qquad \text{where } \phi_i \text{ are conjuncts.}$$

# Composition

$B_1(t) \neq *$

$A(t) \neq *$

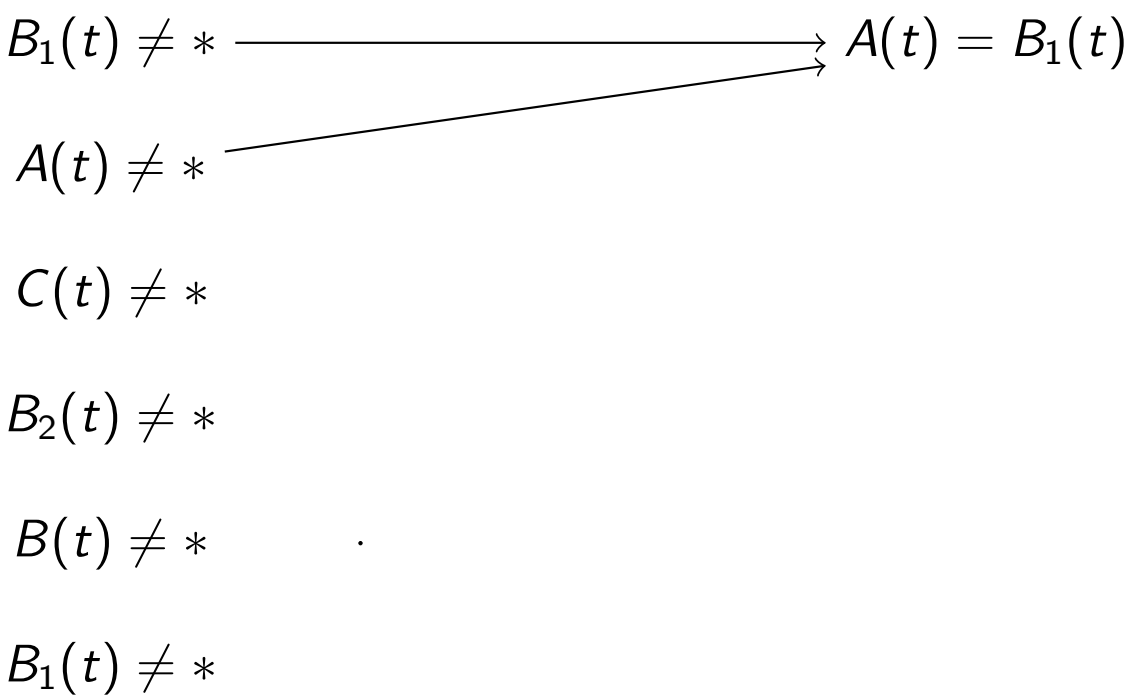
$C(t) \neq *$

$B_2(t) \neq *$

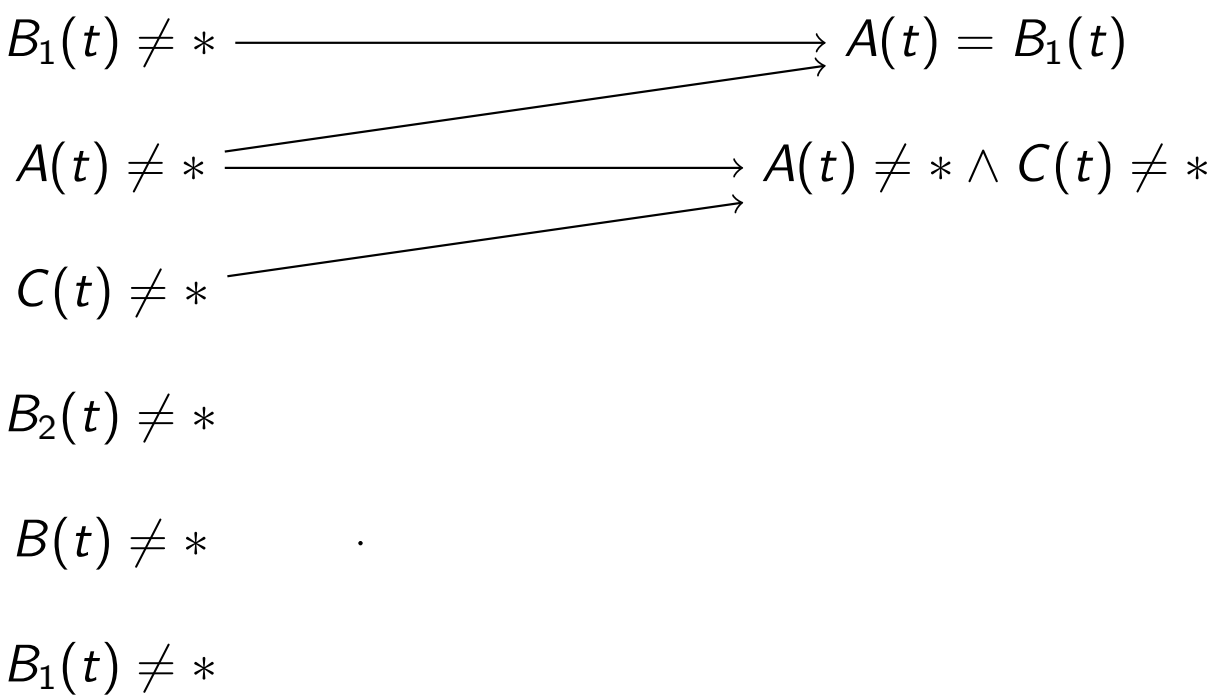
$B(t) \neq *$ .

$B_1(t) \neq *$

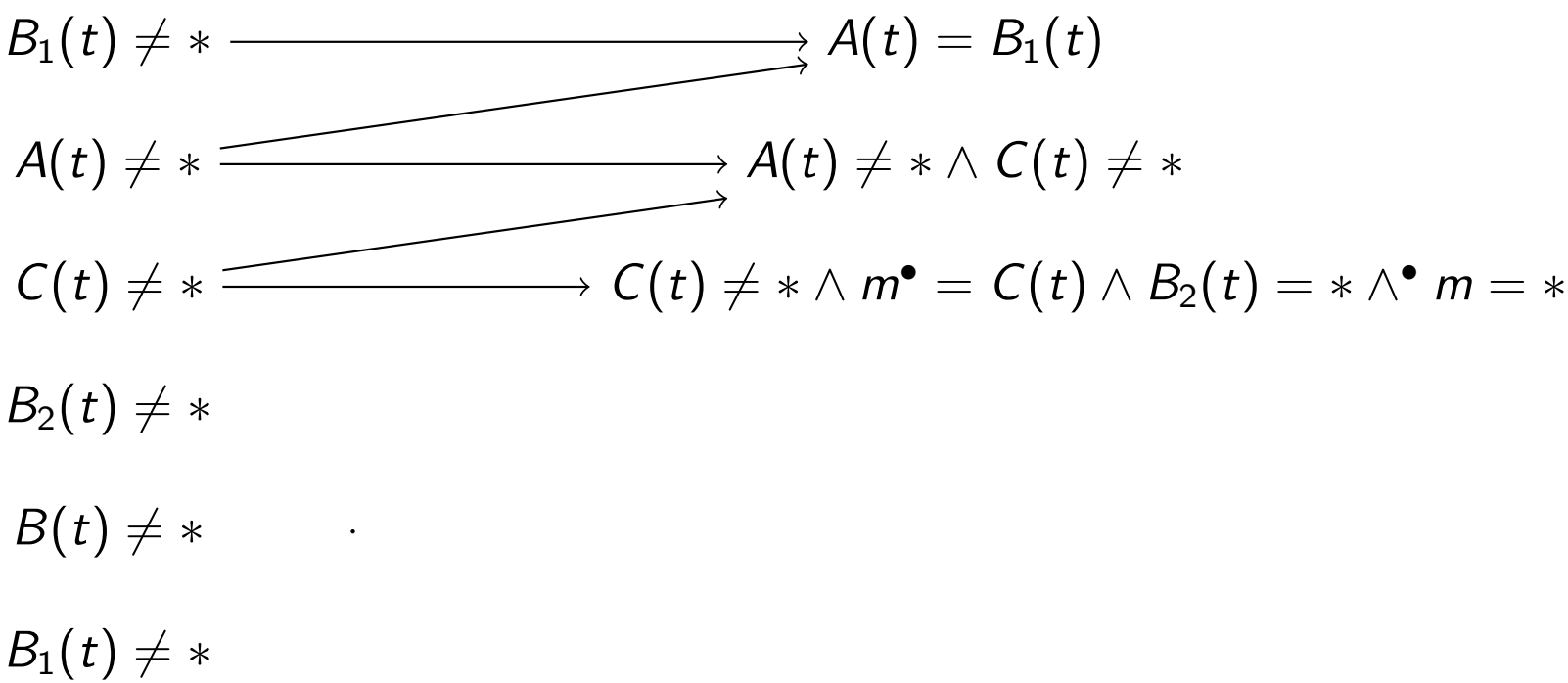
# Composition



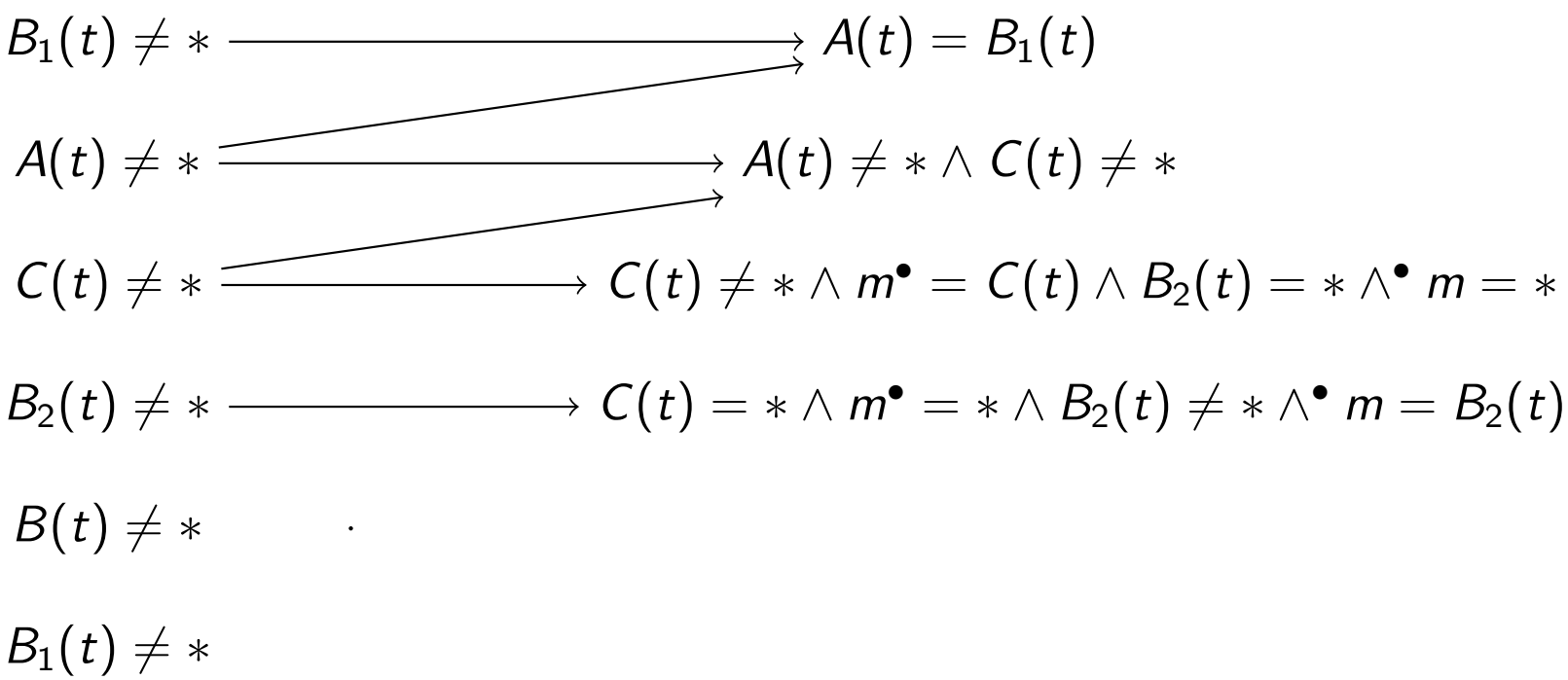
# Composition



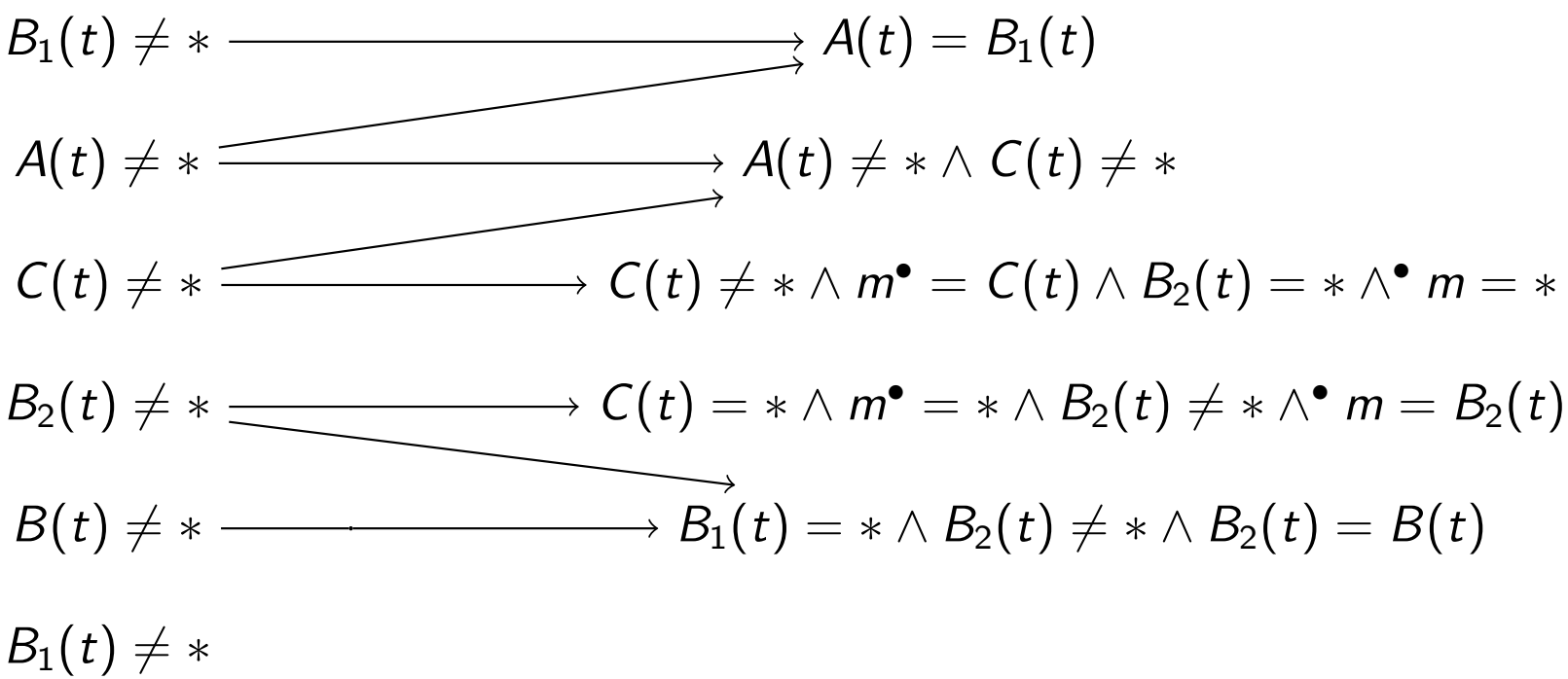
# Composition



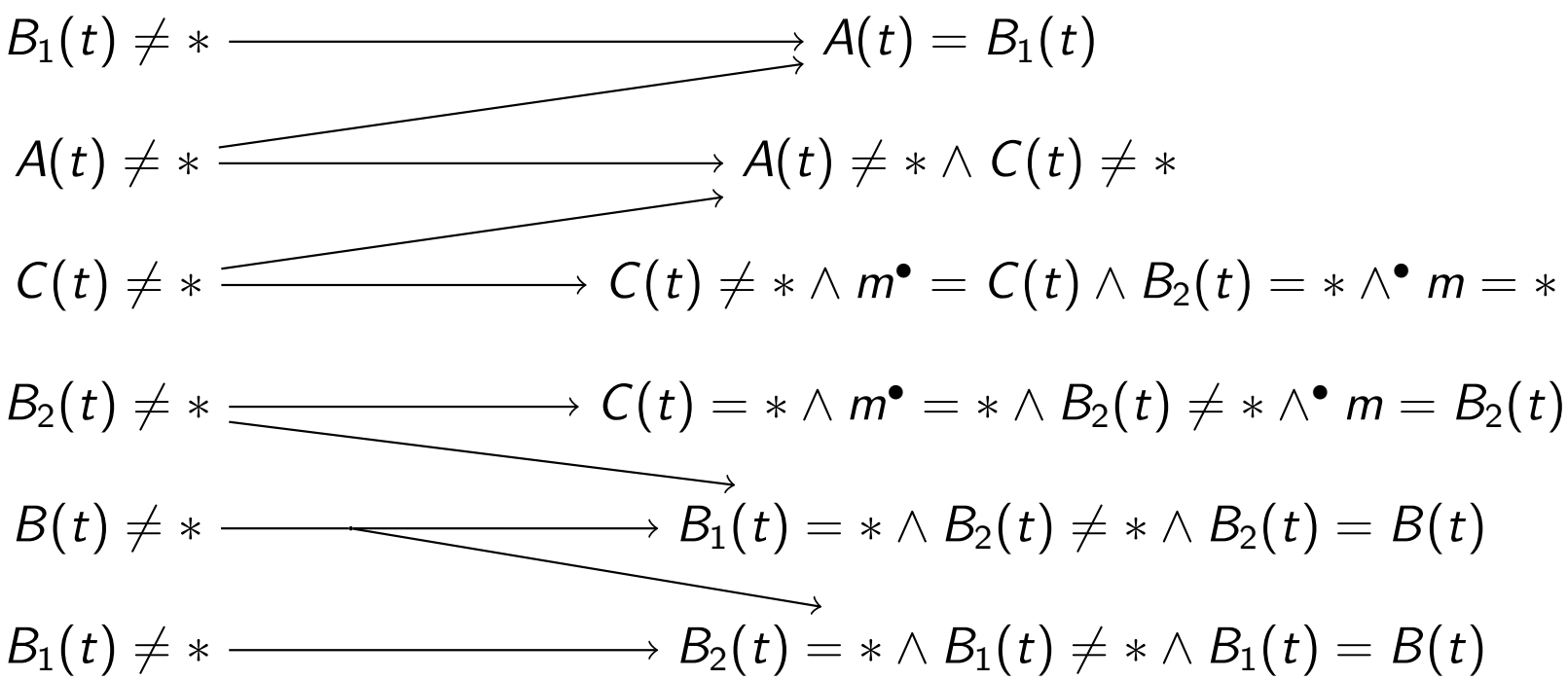
# Composition



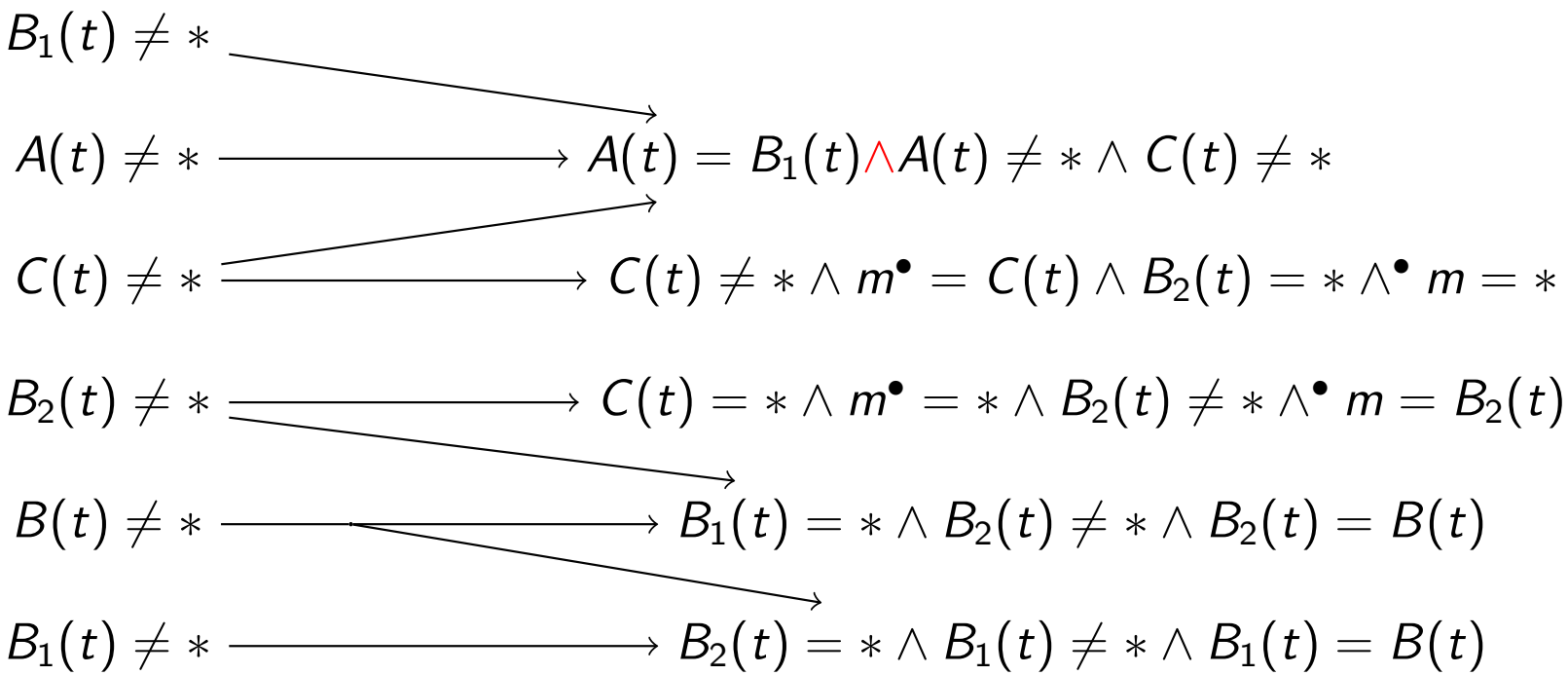
# Composition



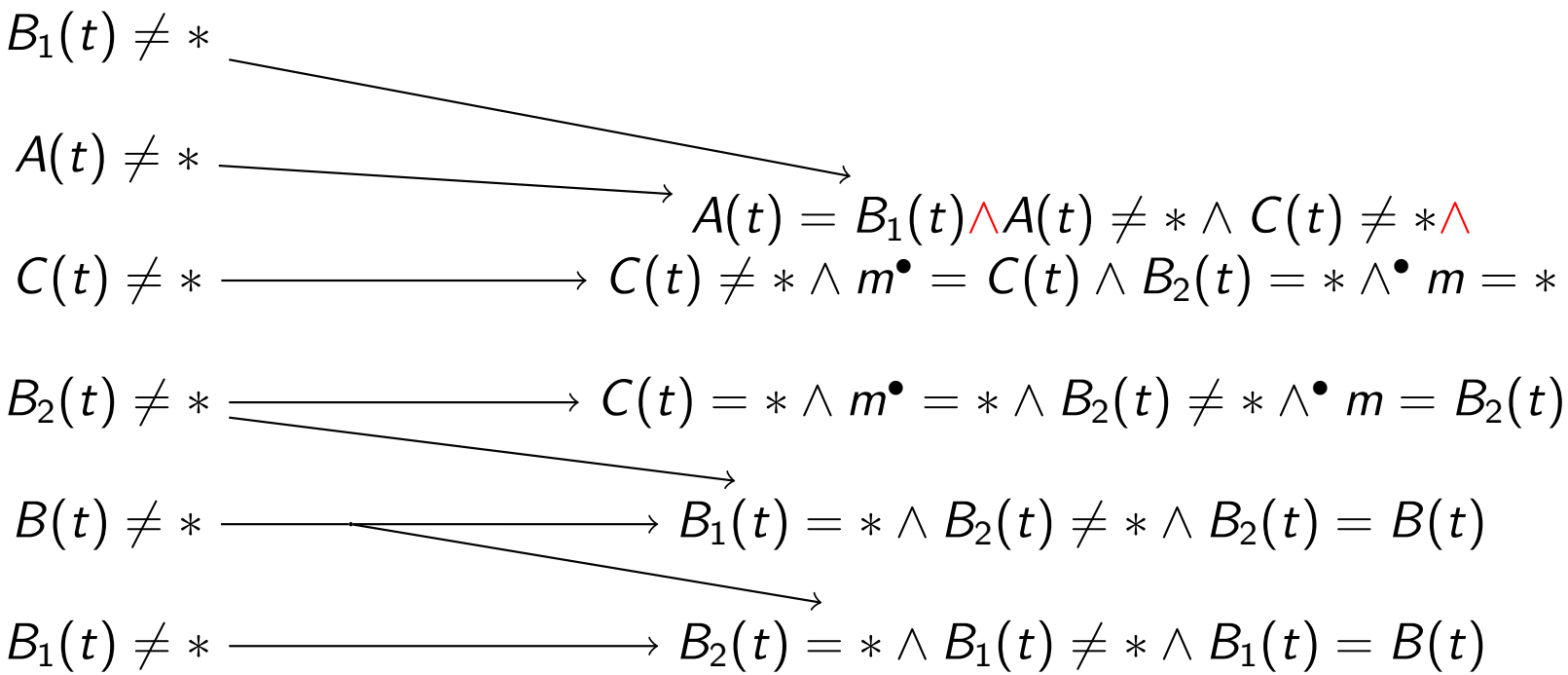
# Composition



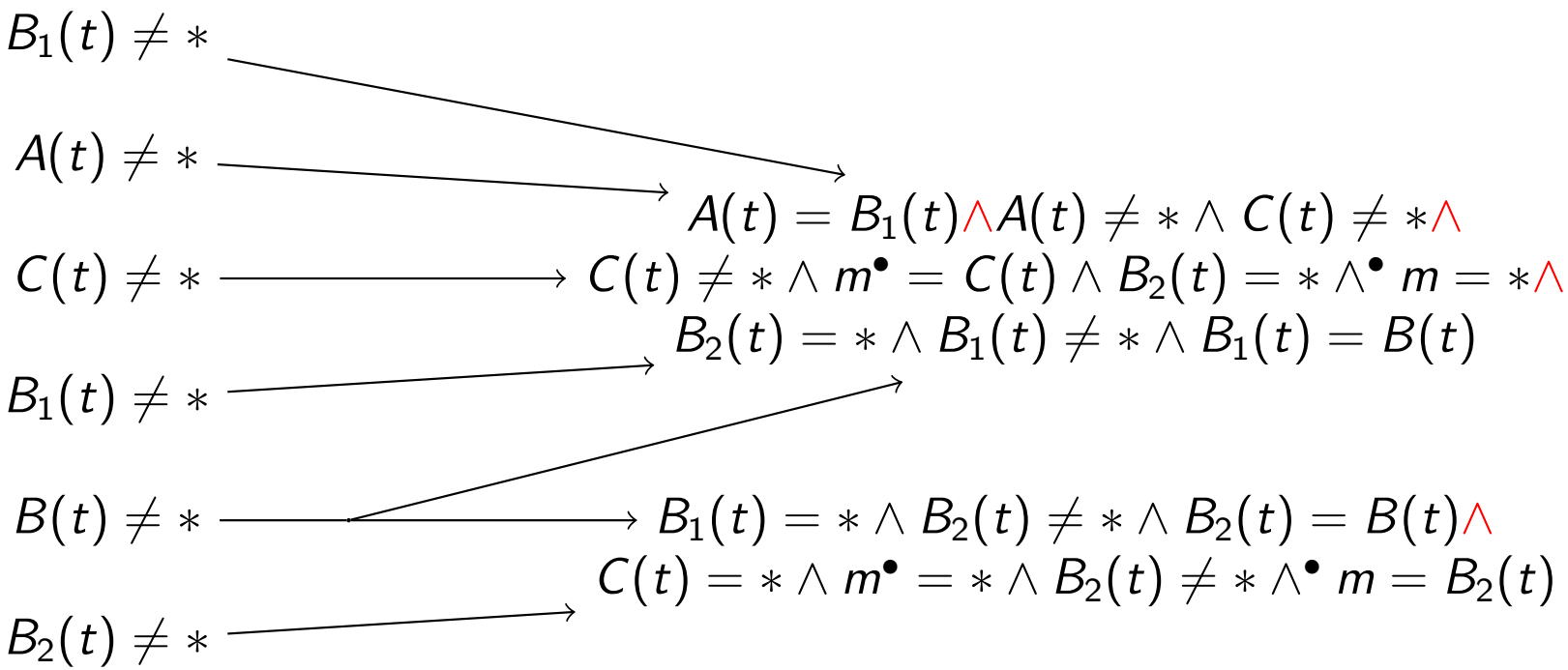
# Simplification



# Simplification



# Simplification



# Commandification

From the simplification step, let $\mathcal{R}$ be the set of synchronous rules.
Providing the input/output type of a port, we transform each rule $r \in \mathcal{R}$ into

- a predicate on the state of the boundary ports and memory: **guard**.
- a predicate on data constraint: **command**.

# Commandification (example)

Formula:

$$C(t) = * \wedge m^{\bullet} = * \wedge B_2(t) \neq * \wedge^{\bullet} m = B_2(t)$$

# Commandification (example)

Formula:

$$C(t) = * \wedge m^{\bullet} = * \wedge B_2(t) \neq * \wedge^{\bullet} m = B_2(t)$$

Guard:

$$B_2(t) \neq * \wedge^{\bullet} m \neq *$$

# Commandification (example)

Formula:

$$C(t) = * \wedge m^{\bullet} = * \wedge B_2(t) \neq * \wedge^{\bullet} m = B_2(t)$$

Guard:

$$B_2(t) \neq * \wedge^{\bullet} m \neq *$$

Command:

$$m = B_2(t) \wedge m^{\bullet} = *$$

# To target language (Java)

Let $GC$ be the set of guarded command and $(g, c) \in GC$ a guarded command.

Intuitive interpretation in Java as if statement and updates.

# To target language (Java)

Let $GC$ be the set of guarded command and $(g, c) \in GC$ a guarded command.

Intuitive interpretation in Java as if statement and updates.

```
while(true){
    if(g1){
        c1;
    }

    ...

    if(gn){
        cn;
    }
}
```

# Properties as component

If the port $A$ fires, then eventually the $D$ fires.

$$\exists t, A(t) \neq * \implies \exists k > t, D(k) \neq *$$

Never port $A$ and port $B$ fire together.

$$\forall t, A(t) \neq * \implies B(t) = *$$

If the port $D$ fires, then one of the port $A$ or $B$ or $C$ fires.

$$\forall t, D(t) \neq * \implies A(t) \neq * \vee B(t) \neq * \vee C(t) \neq *$$

# Conclusion

Start with a design language: Reo.

# Conclusion

Start with a design language: Reo.

Make it precise with some formal semantics: constraint automata, predicate logic.

# Conclusion

Start with a design language: Reo.

Make it precise with some formal semantics: constraint automata, predicate logic.

Define formally simplifications of our design: composition.

# Conclusion

Start with a design language: Reo.

Make it precise with some formal semantics: constraint automata, predicate logic.

Define formally simplifications of our design: composition.

Translate our formal representation to different back-ends: Treo to Java, ... .

# Conclusion

Start with a design language: Reo.

Make it precise with some formal semantics: constraint automata, predicate logic.

Define formally simplifications of our design: composition.

Translate our formal representation to different back-ends: Treo to Java, ... .

Prove some properties: semantics preserving.