ALLOY 6

EXERCISE LECTURE

Authors:
Luca Padalino
Francesca Pia Panaccione
Francesco Santambrogio

CONCURRENCY: a property of a system in which multiple processes or threads can run simultaneously or appear to be running simultaneously.

Scenarios that deal with **CONCURRENCY**:

- **distributed systems**
- multi-threading/multi-tasking applications
- data migrations…

CONCURRENCY: a property of a system in which multiple processes or threads can run simultaneously or appear to be running simultaneously.

Scenarios that deal with **CONCURRENCY**:

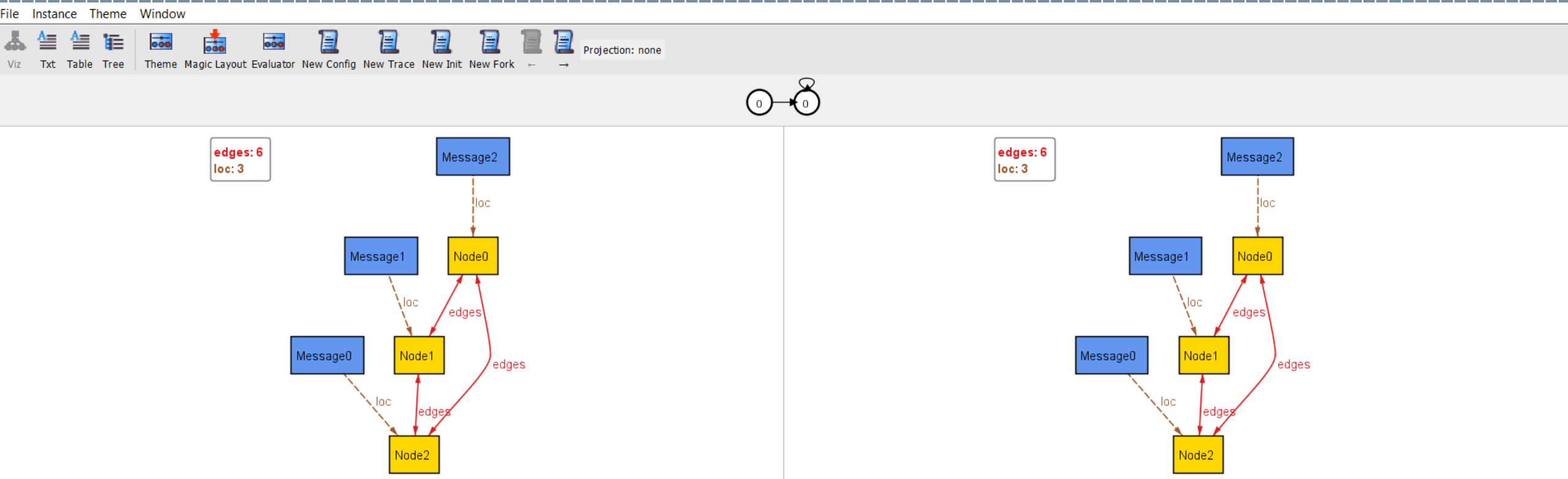
- **distributed systems:** a collection of independent components located on different systems, sharing messages in order to operate as a single unit.

EXERCISE 1
Concurrent communication in a distributed system

- Collection of **independent components**
- Sharing **messages**
- To operate as a **Single unit**
- + There can be only **one message** in a node

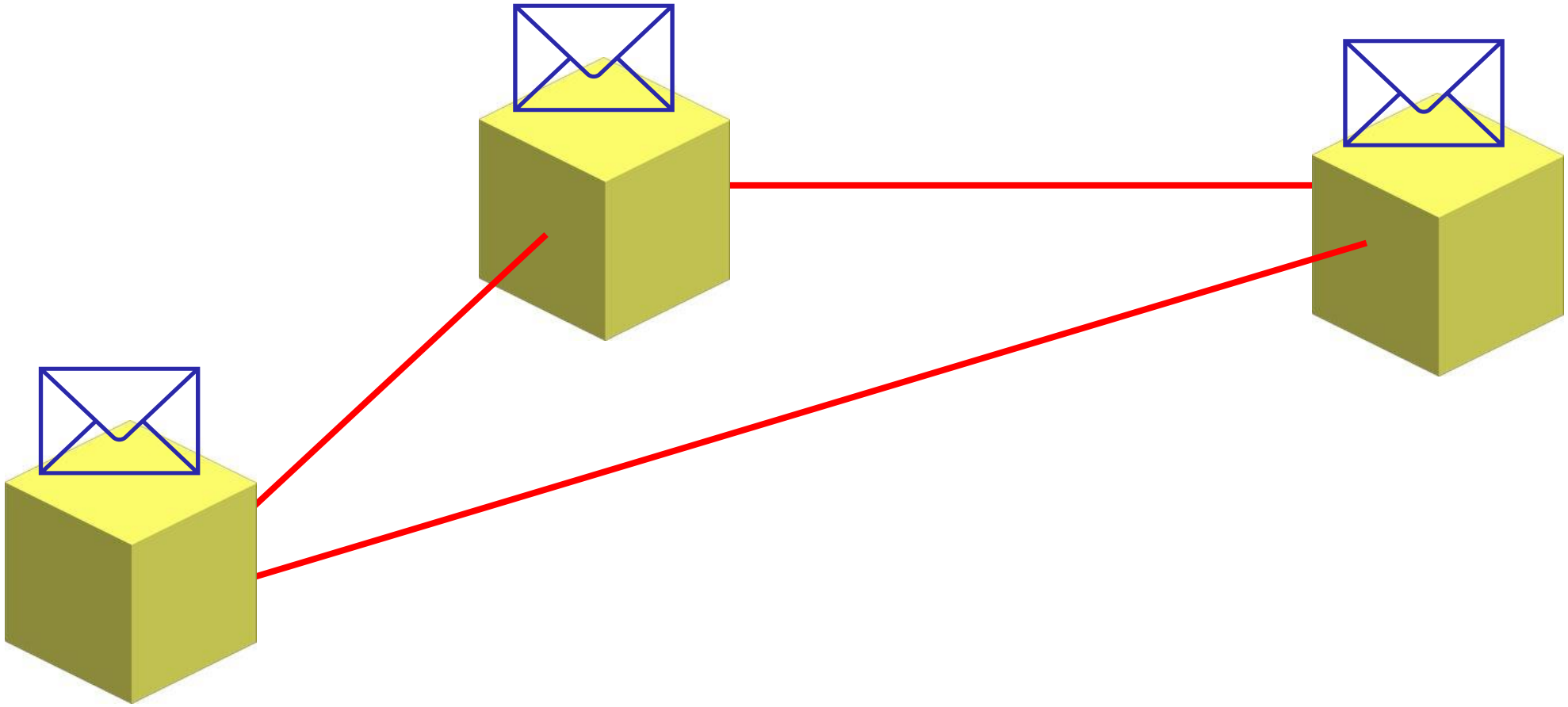
```
sig Node {
    edges: some Node
}

sig Message {
    var loc: Node
}

fact ConnectedGraph{
    all n1, n2: Node |
    n1 in n2.edges iff n2 in n1.edges and
    !(n1 = n2) and n1.*edges = Node}

fact {
    all disj m1,m2:Message |
    always !(m1.loc = m2.loc)
}
```

EXERCISE 1
Concurrent communication in a distributed system



A possible configuration

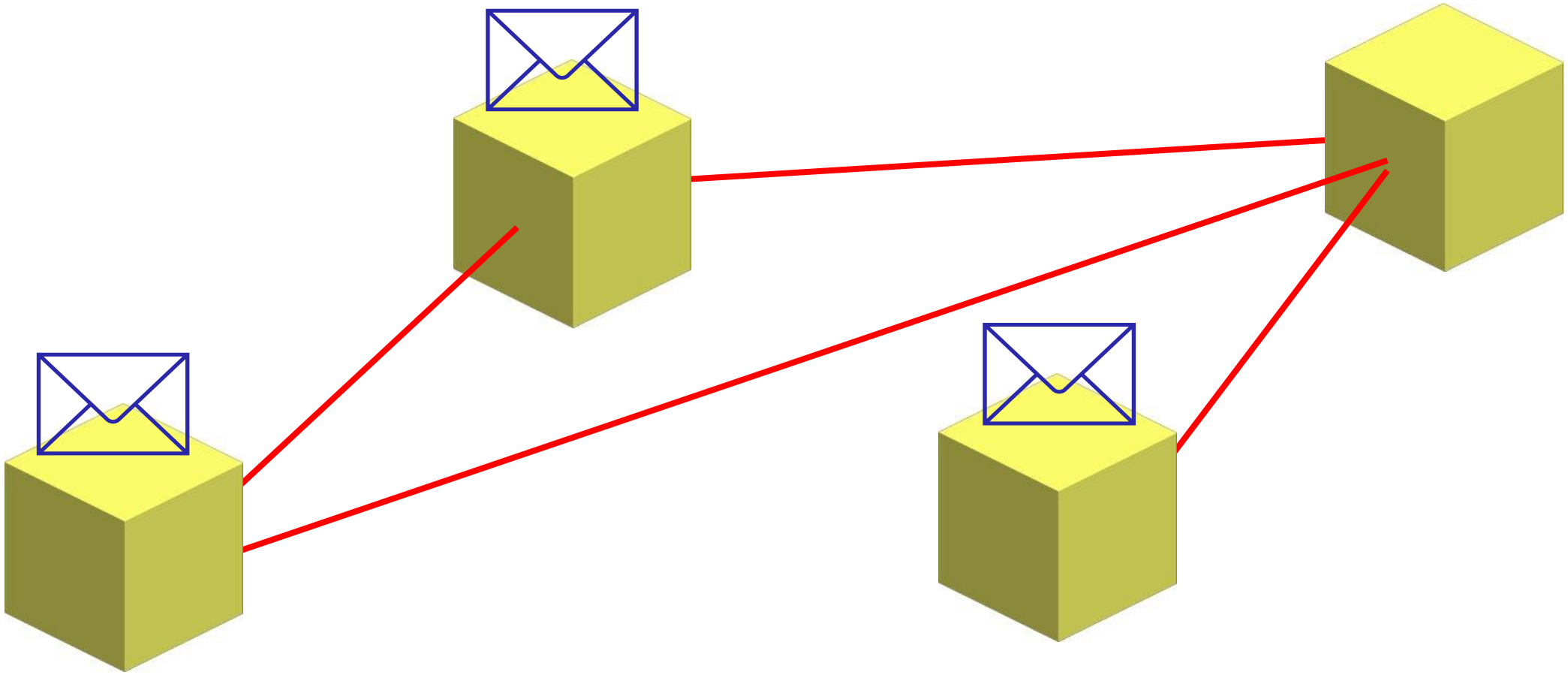
EXERCISE 1
Concurrent communication in a distributed system



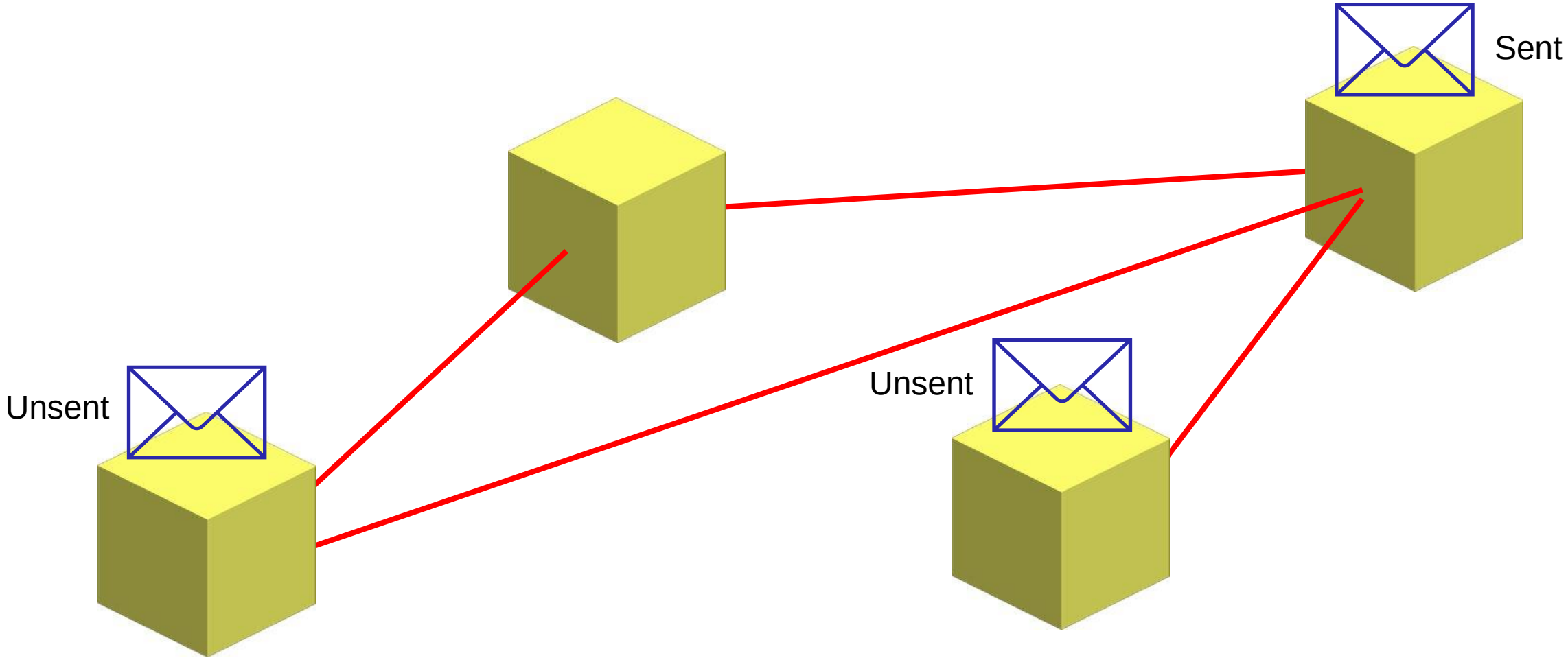
Messages are exchanged between nodes:

```
// a message moves to a node if…
pred send[m: Message, n: Node] {
    // the node is directly connected to the one where the message is currently located
    n in m.loc.edges
    // there are no messages in the destination node
    no m2: Message | m2.loc = n
    // and the location changes to the destination node
    m.loc' = n
}
pred sent[m: Message] {
    some n: Node | send[m, n]
}
pred unsent[m: Message] {
    m.loc = m.loc'
}
```

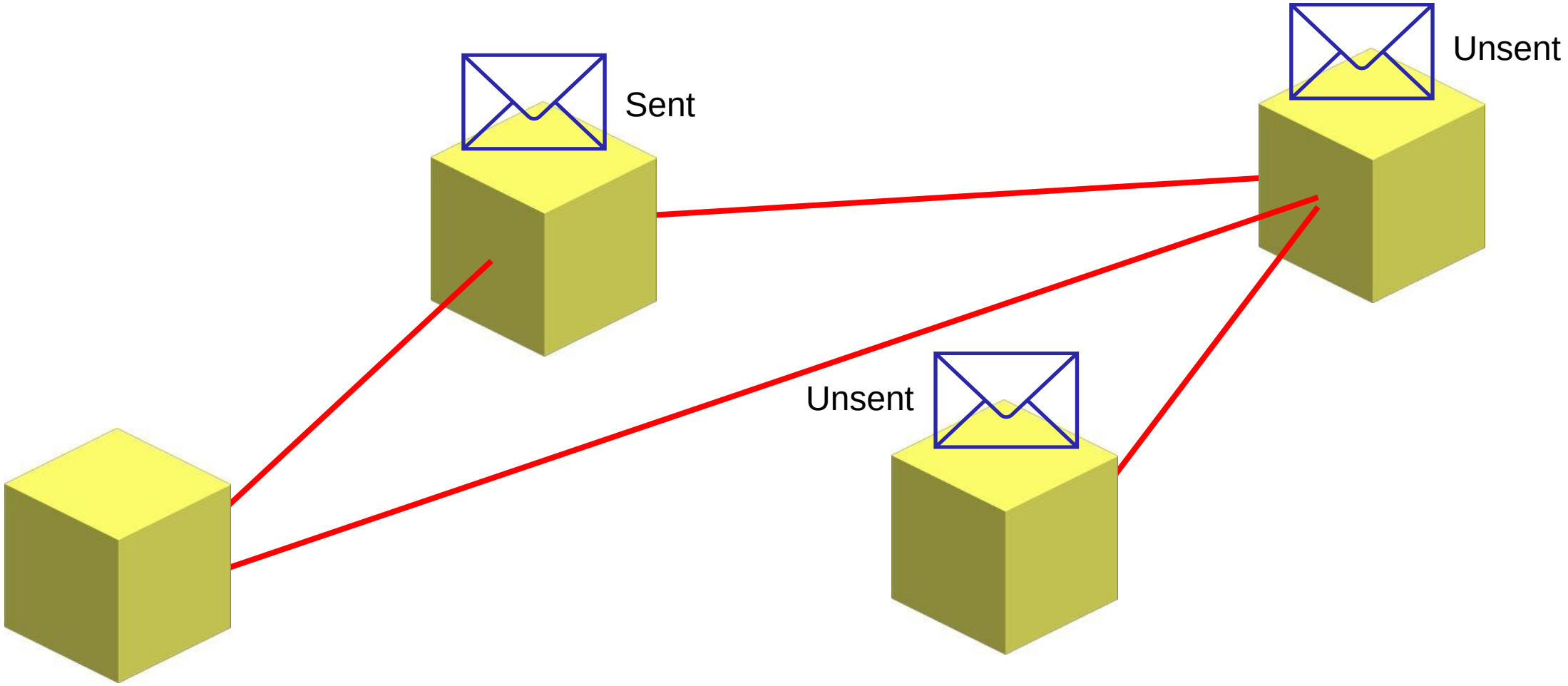
EXERCISE 1
Concurrent communication in a distributed system



EXERCISE 1
Concurrent communication in a distributed system



EXERCISE 1
Concurrent communication in a distributed system



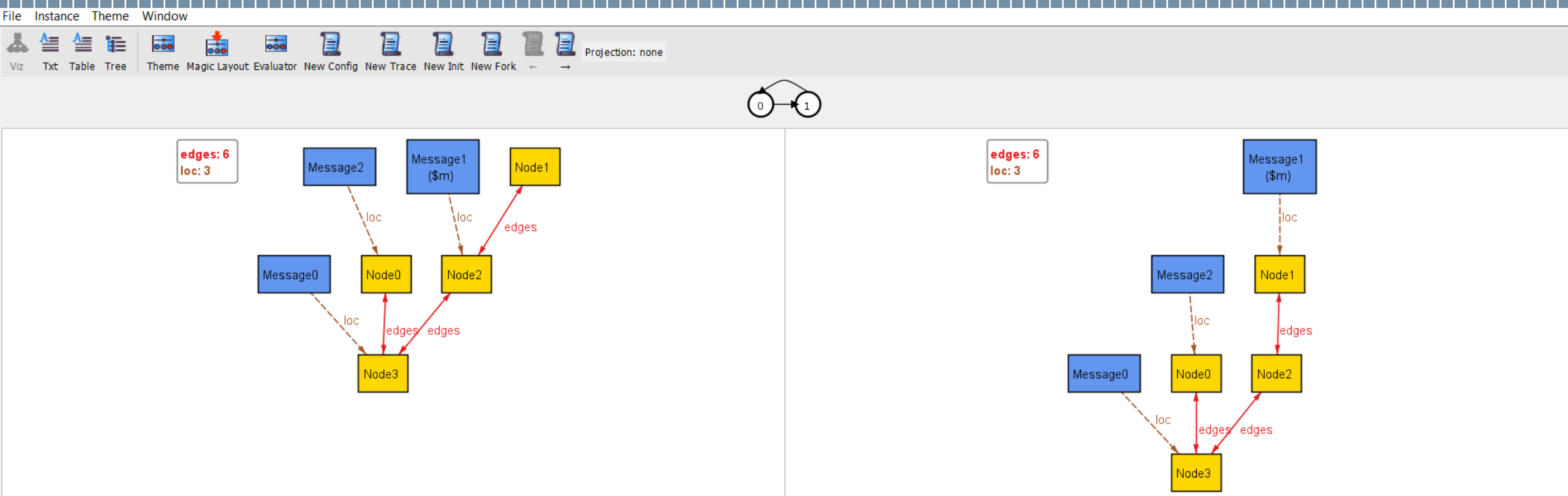
Let's start by moving only **one** message each step:

```
// In every step, a certain message moves, while the others remain unsent
fact MessageSending {
    always (one m: Message| all m2: Message-m| (sent [m] and unsent[m2]))
}
```

FULLY DETERMINING the state:

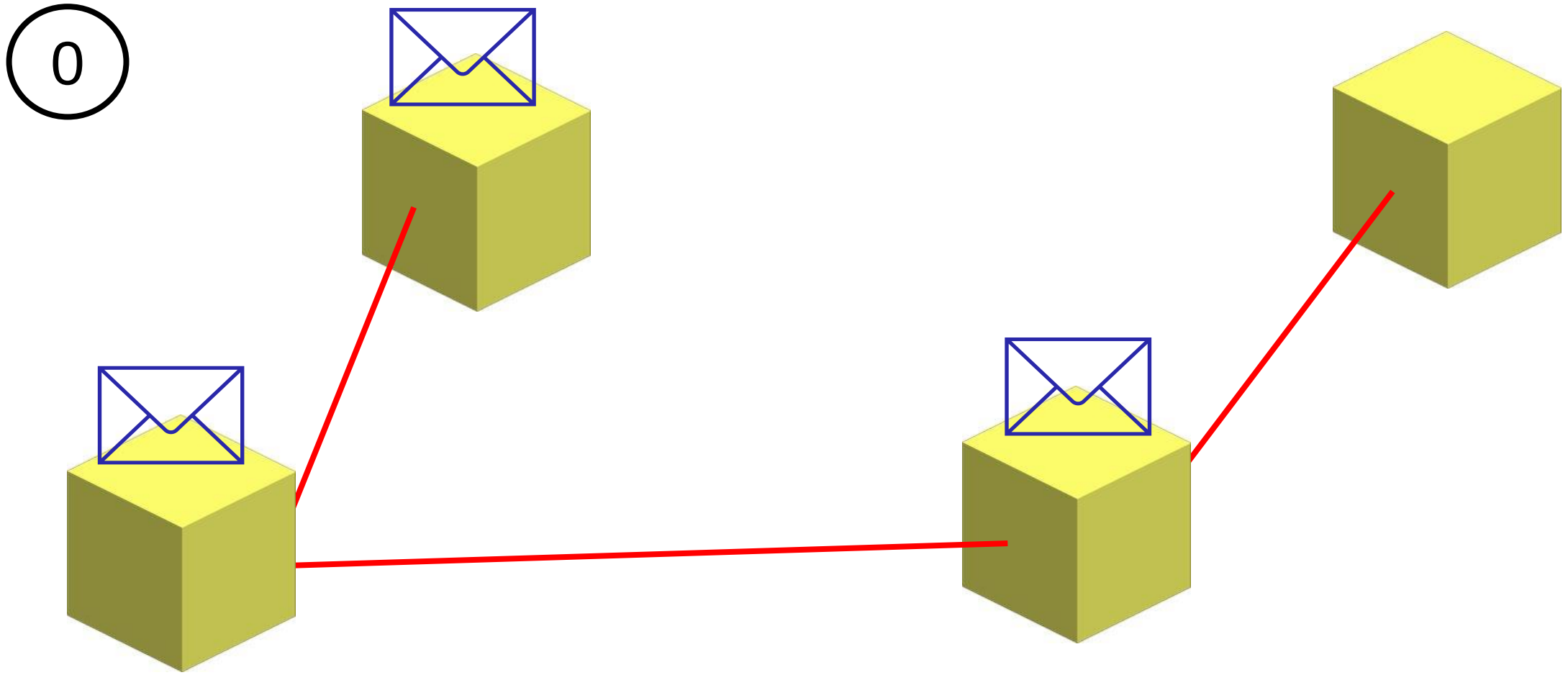
- Make explicit **what changes** during the state transition: sent[m]
- Make explicit also **what does not change:** unsent[m]

EXERCISE 1
Concurrent communication in a distributed system

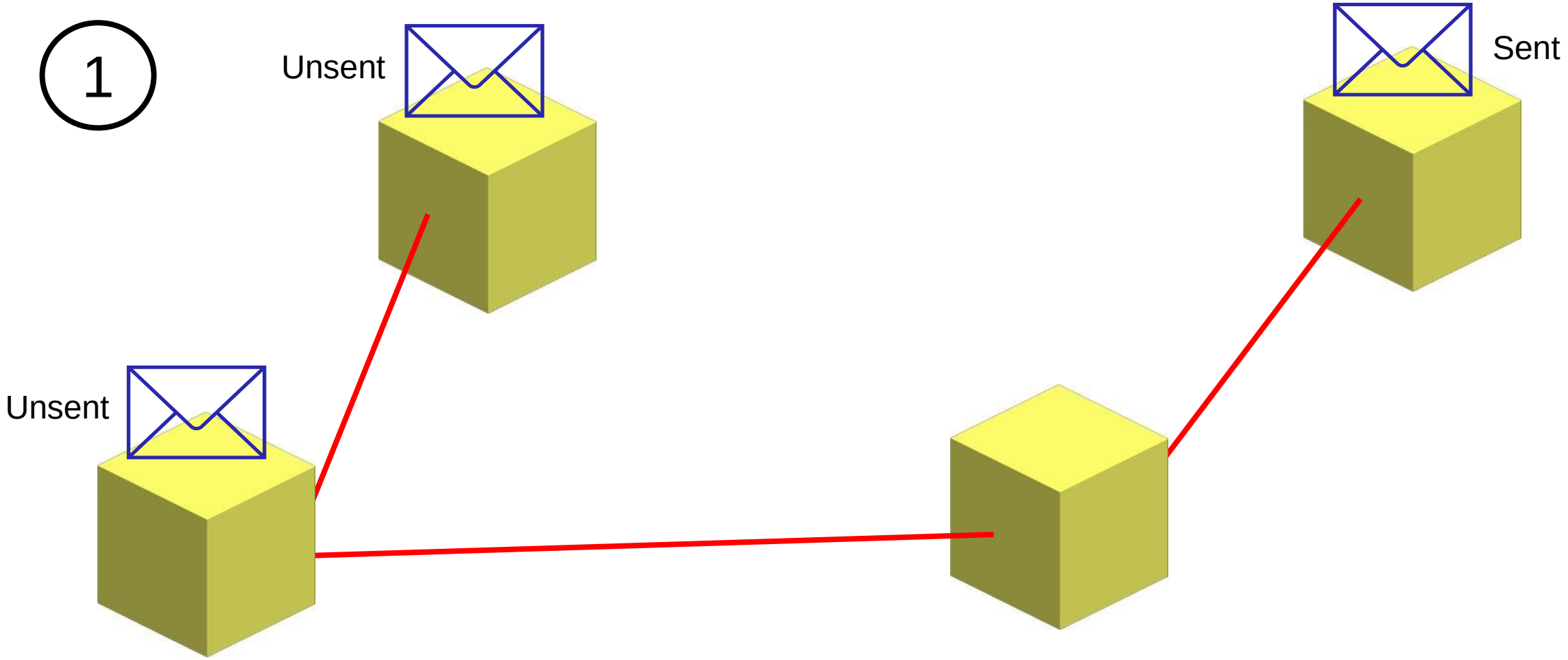



```
run { #Message = 3 } for 6
```

EXERCISE 1
Concurrent communication in a distributed system



EXERCISE 1
Concurrent communication in a distributed system



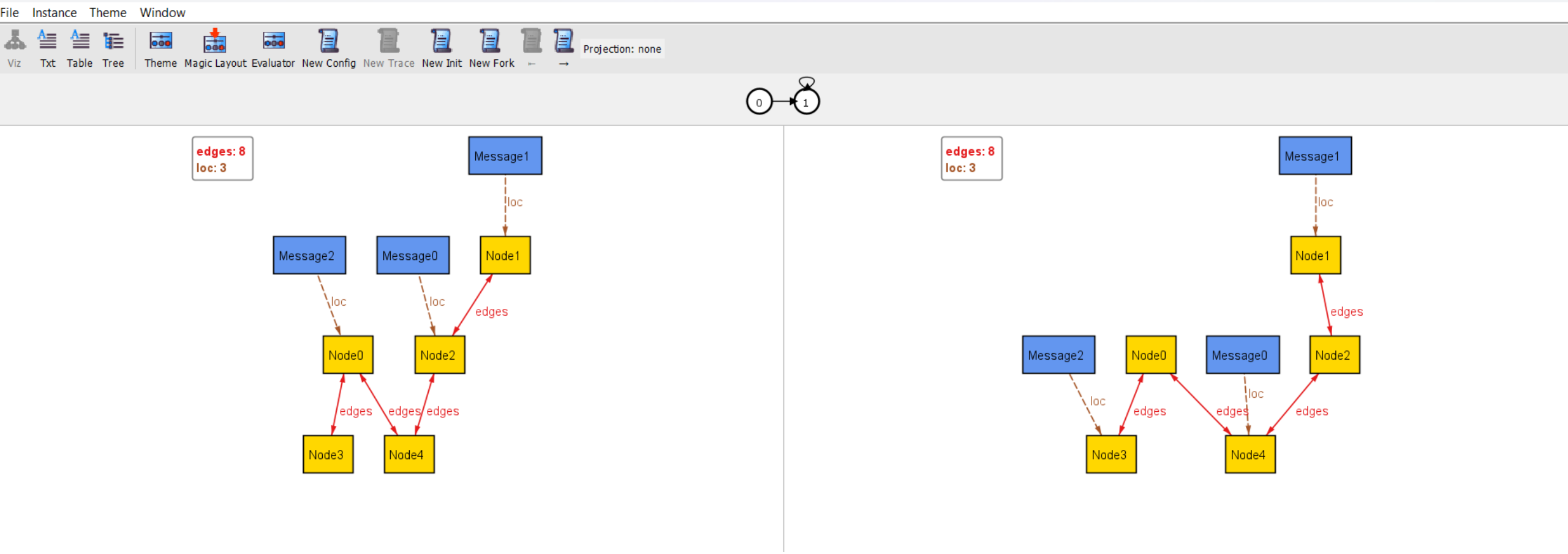
Let's send multiple messages simultaneously:

```
// All the messages are sent
fact SendingOrNot {
    all m: Message | always (sent [m] or unsent [m])
}
```

FULLY DETERMINING the state:

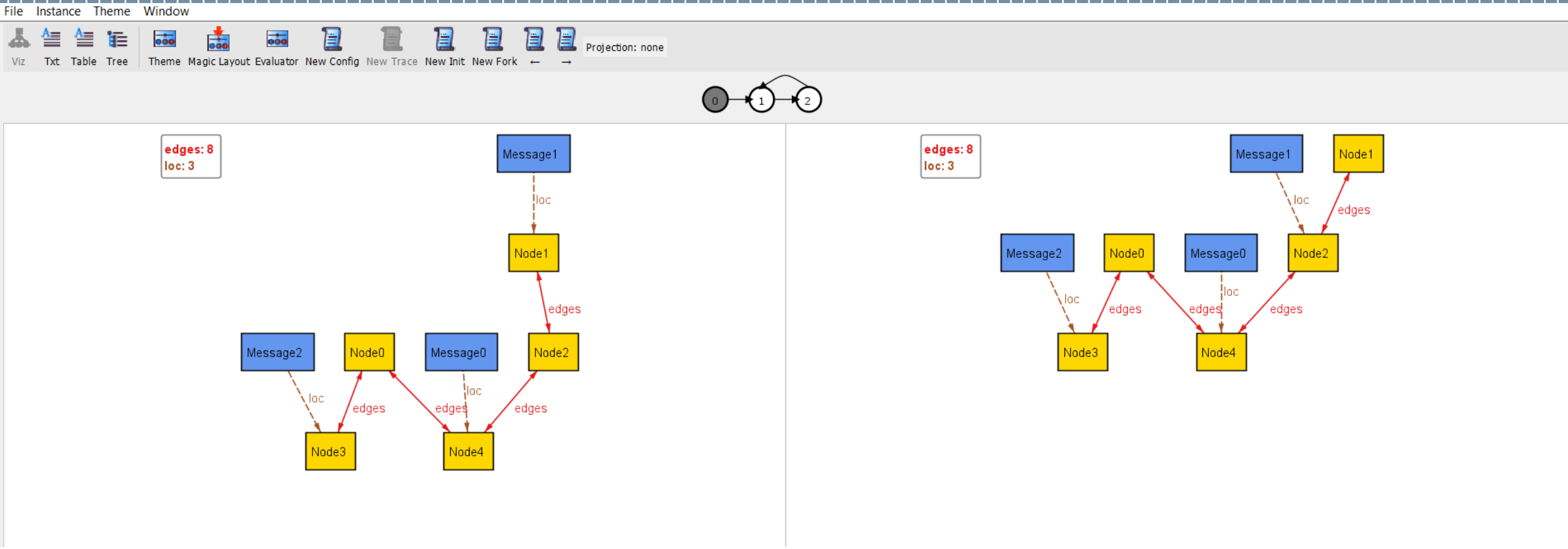
- Make explicit **what changes** during the state transition: sent[m]
- Make explicit also **what does not change:** unsent[m]

EXERCISE 1
Concurrent communication in a distributed system



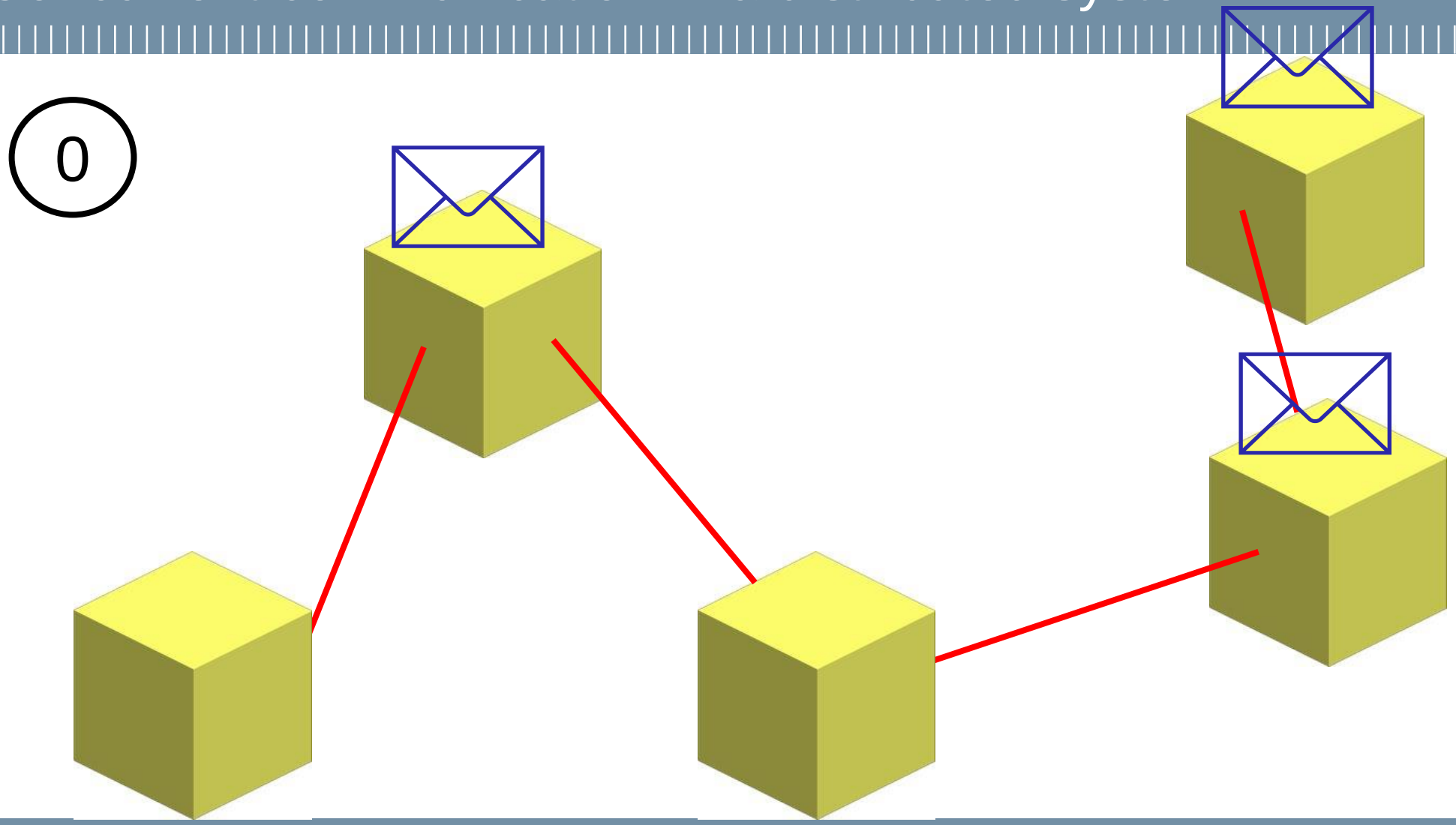

```
run { #Message = 3 } for 6
```

EXERCISE 1
Concurrent communication in a distributed system

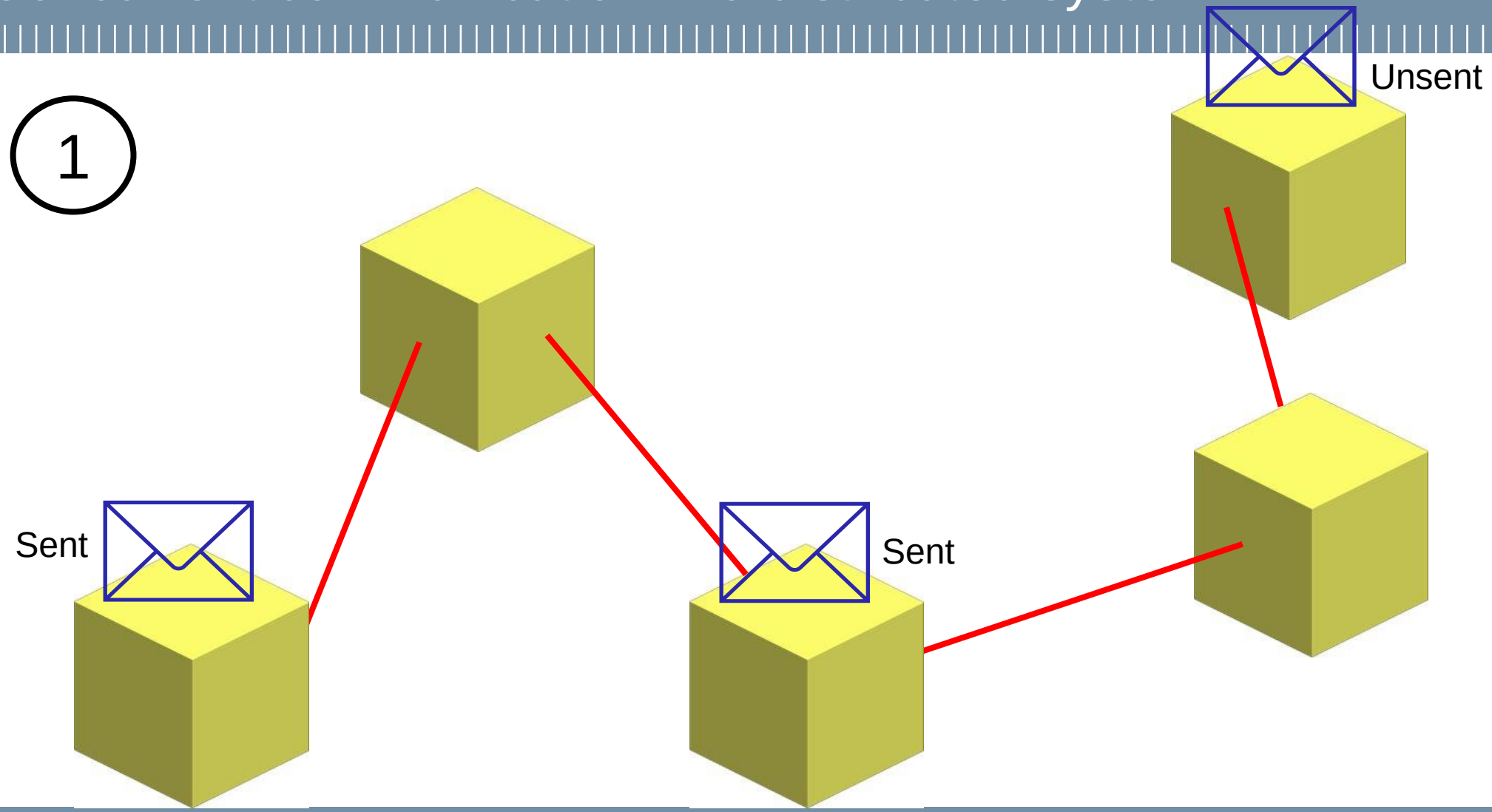



```
run { #Message = 3 } for 6
```

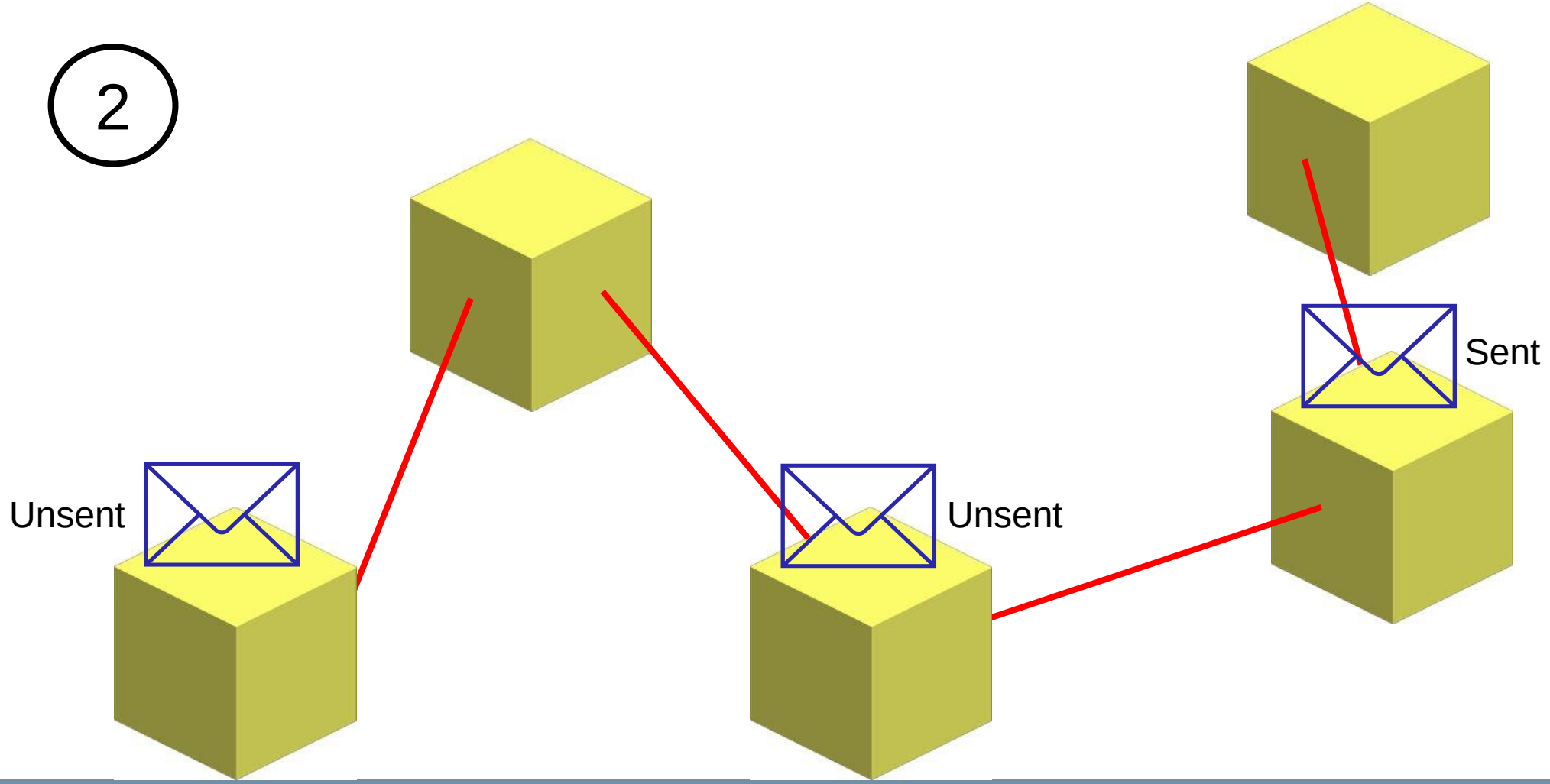
EXERCISE 1
Concurrent communication in a distributed system



EXERCISE 1
Concurrent communication in a distributed system



EXERCISE 1
Concurrent communication in a distributed system



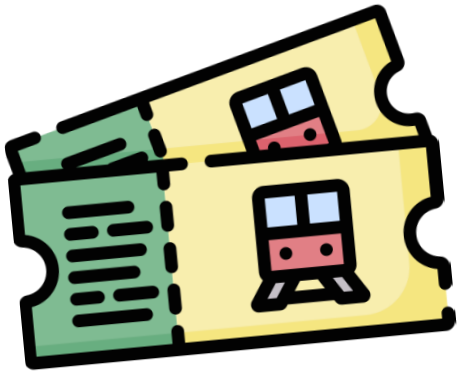
EXERCISE 1
Concurrent communication in a distributed system


```
sig Node {
    edges: some Node
}

fact ConnectedGraph {
    all n1, n2: Node |
    n1 in n2.edges iff n2 in n1.edges and
    !(n1 = n2) and n1.*edges =Node
}

sig Message {
    var loc: Node
}

fact {
    all disj m1,m2: Message |
    always !(m1.loc = m2.loc)
}
```

```
pred send[m: Message, n: Node] {
    n in m.loc.edges
    no m2: Message | m2.loc = n
    m.loc' = n
}

pred sent[m: Message] {
    some n: Node | move[m, n]
}

pred unsent[m: Message] {
    m.loc = m.loc'
}

fact SendingOrNot {
    all m: Message | always moved [m]
}
```

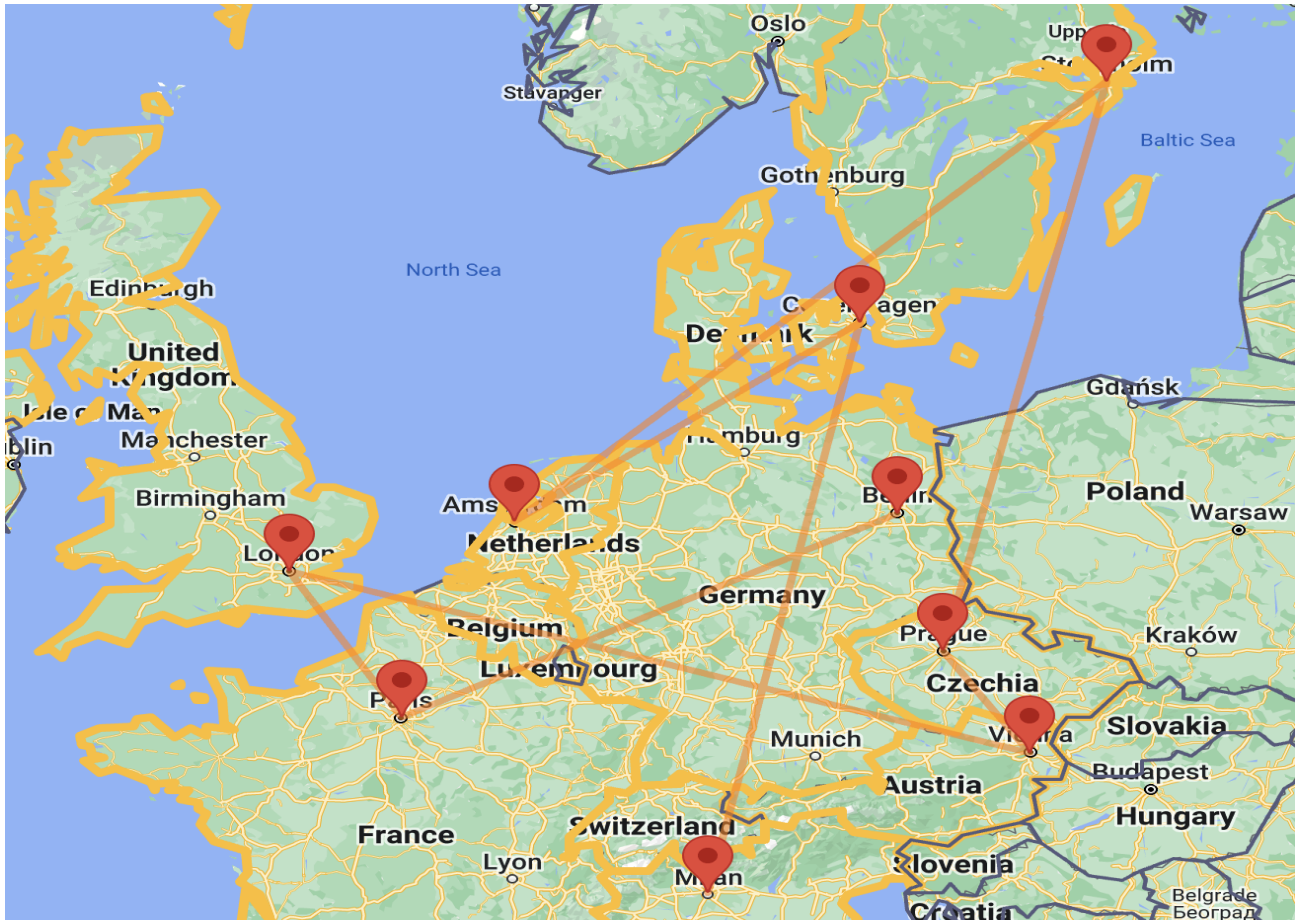
EXERCISE 2
Interrail: general context

- A Person wants to organize an **interrail**.
- An interrail is a type of travel where the **traveler** can **move** to different **cities** by train.
- The person has a **pass** where all the cities he can visit are specified.
- The travel includes **European cities**.



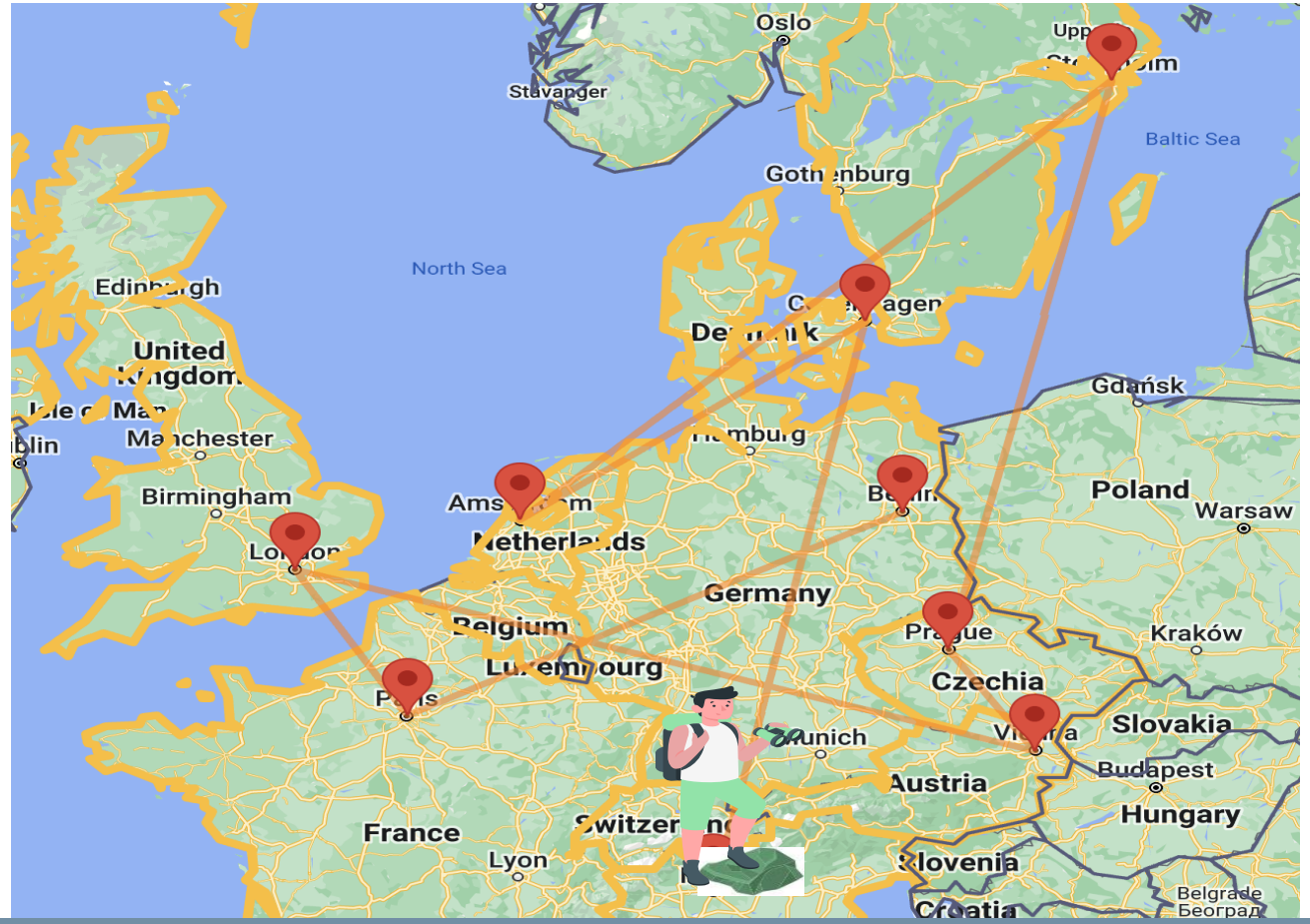
EXERCISE 2
Interrail: general context

- Cities must be **interconnected** through railways in order to create an itinerary.



EXERCISE 2
Interrail: general context

- We assume that the Person can move only to the **cities** that **he has not visited yet**.



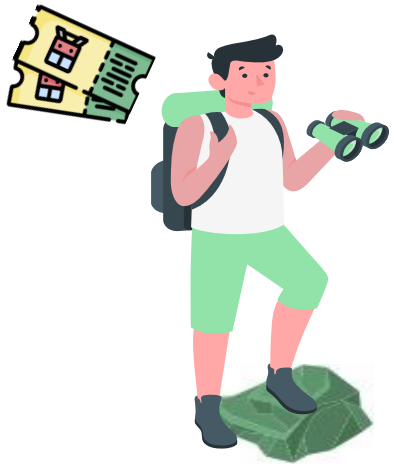
EXERCISE 2
Interrail: general context

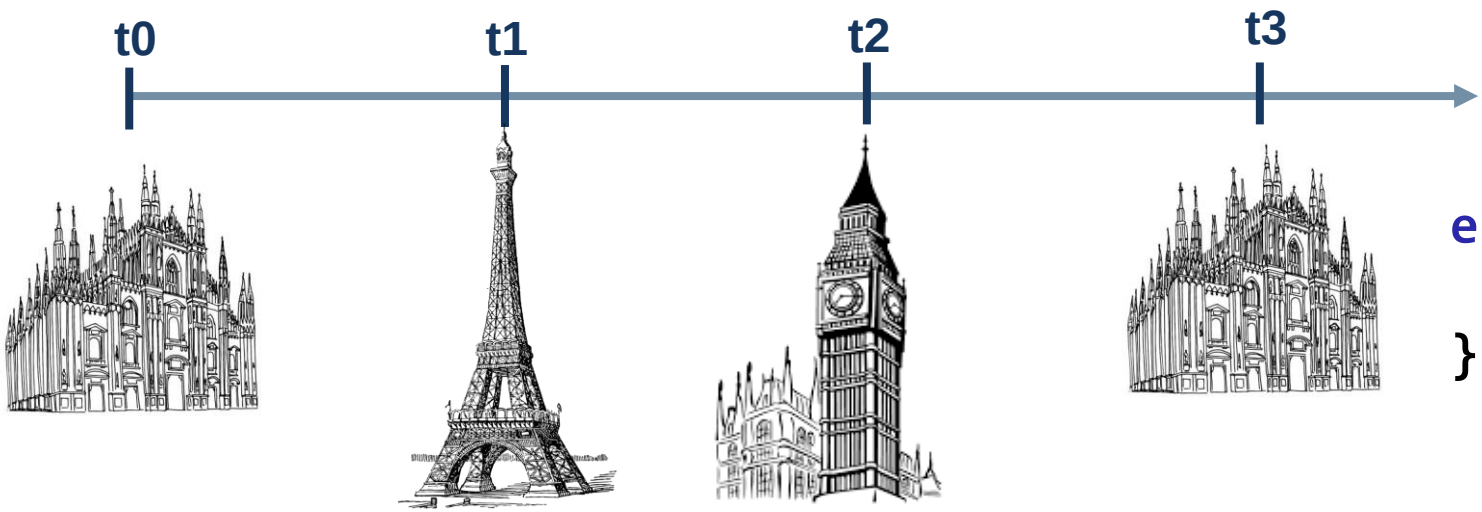
- We want to keep track of whether the person **is traveling** or **has arrived** at the final destination.

EXERCISE 2
Interrail: signatures


```
sig City {
    linkedWith: some City,
    name: disj one EuropeanCities
}

enum EuropeanCities {
    Milan, Berlin, Paris, London,
    Vienna, Prague, Stockholm,
    Amsterdam, Copenaghen
}
```

EXERCISE 2
Interrail: signatures

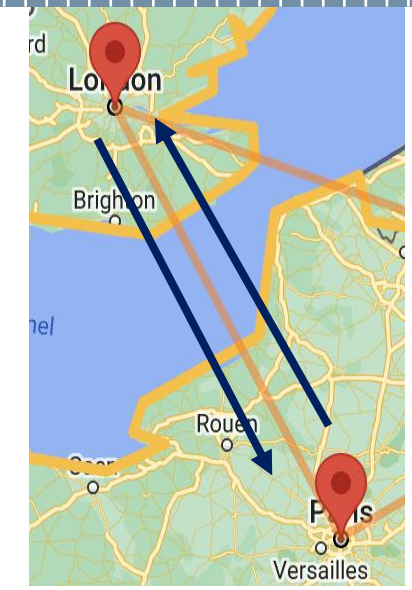





```
sig Person {
    pass: some City,
    var loc: City,
    var visited_cities: set City,
    var status: Status
}

enum Status {
    Travelling, AtDestination
}
```

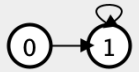
EXERCISE 2
Interrail: contraints

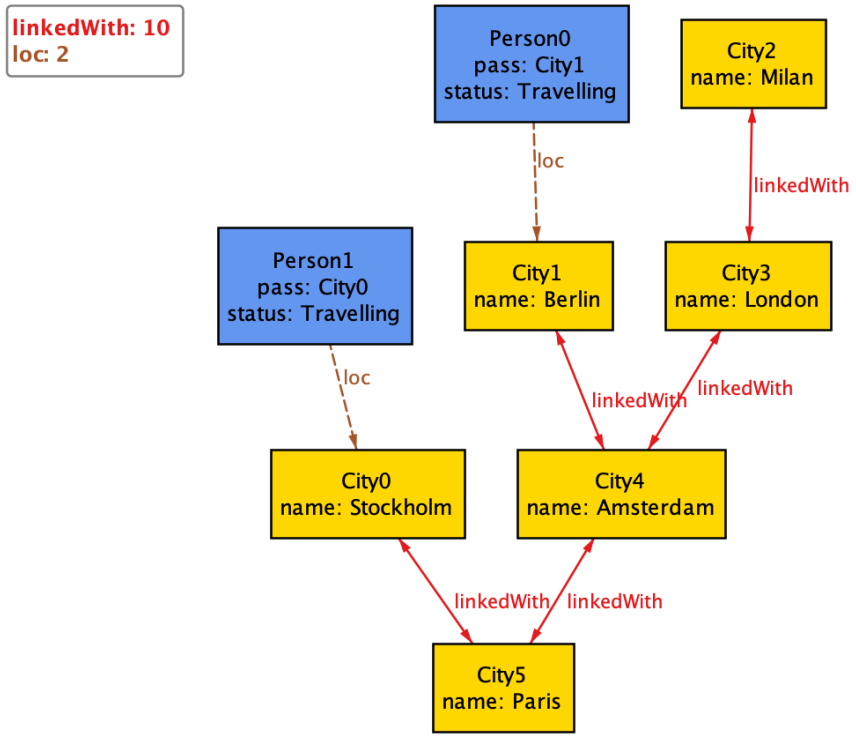
```
// Two cities are connected to each other

fact BidirectionalConnection {
    all c1, c2: City | c1 in c2.linkedWith
    iff c2 in c1.linkedWith and !(c1 = c2)
    and c1.*linkedWith = City
}
```

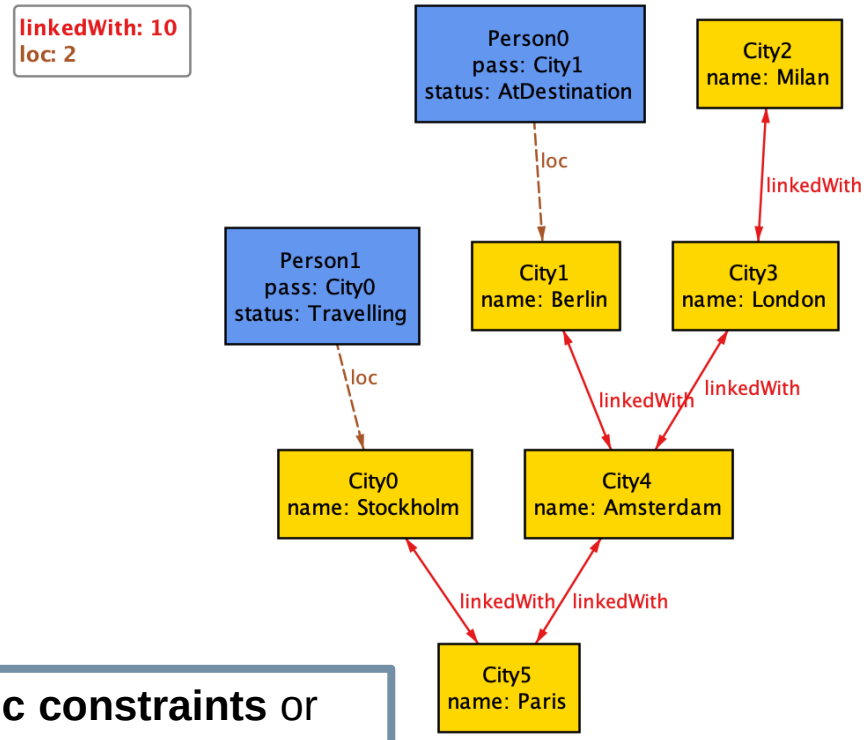



```
// At the beginning of the interrail, the traveler has not visited any cities yet

fact NoVisitedCitiesAtTheBeginning {
    all p:Person | no p.visited_cities
}
```

EXERCISE 2
Interrail: visualizer








```
pred world1{#City = 6 and #Person = 2}
run world1 for 10
```

So far, only **static constraints** or constraints about the **first current state**, **not** about the **behavior** of the system during **state transitions**

EXERCISE 2
Interrail: visualizer

. . .How can we write a **function** that defines the **movement** of a **person** 'p' to a **city** 'c' ?

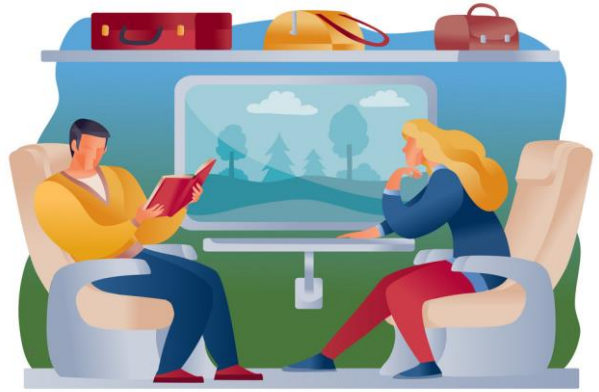
What **conditions** must be applied so that the **logic** of the system does **not break down**?. . .

- The **city** 'c' must be **linked** to the one **where I am** currently in
- I must **have** the **pass** for the city 'c'
- I must **not** have **visited** city 'c' **before**
- The **city** 'c' must be **different** from the **one I am currently** in
- The **place** where **I will be** in the **next state** becomes 'c'
- 'c' **will be** among the **visited cities**

```
// A given person travels to a given city
pred move[p: Person, c: City] {
    c in p.loc.linkedWith and c in p.pass and c not in
    p.visited_cities and c not in p.loc
    p.loc' = c
    c in p.visited_cities'
}
```

EXERCISE 2
Interrail: predicates


```
// A given person has moved
pred moved[p: Person] { some c: City | move[p,c]}
```



```
// A person stays in his location
pred unmoved[p: Person] {  p.loc = p.loc'}
```

EXERCISE 2
Interrail: question

```
// Travelers start their journey from Milan and eventually they will come back
```

```
fact StartingFromMilanAndComingBack {
    all p:Person | p.loc.name = Milan and p.status = Travelling
    and (eventually (p.loc.name = Milan and p.status=AtDestination))
}
```

```
fact StartingFromMilanAndComingBack {
    all p:Person | p.loc.name = Milan and p.status = Travelling
    and after (eventually (always p.loc.name = Milan and p.status=AtDestination))
}
```

```
fact StartingFromMilanAndComingBack {
    all p:Person | p.loc.name = Milan and p.status = Travelling
    and after (eventually (p.loc.name = Milan and p.status=AtDestination))
}
```

EXERCISE 2
Interrail: question

https://app.wooclap.com/LFYKLS?from=instruction-slide

```
// Travelers start their journey from Milan and then they will come back

fact StartingFromMilanAndComingBack {
    all p:Person | p.loc.name = Milan and p.status = Travelling
    and after (eventually (always p.loc.name = Milan and p.status=AtDestination))
}

// Travelers do not come back to Milan if they have not visited all the cities of the pass
before

fact VisitingAllCities {
    all p: Person, c: City | (c in p.pass and !(c.name = Milan)) implies
    (always (after !(p.loc.name = Milan) until (c in p.visited_cities)))
}
```

EXERCISE 2
Interrail: costraints

```
// The people are travelling since their departure

fact Departure {
    all p: Person | always (moved[p] implies
    (p.status = Travelling since p.loc.name = Milan))
}

// An assertion with «triggered»

assert Departure2{
    all p: Person | always (p.status = Travelling implies
    (p.loc.name = Milan triggered p.status = Travelling))
}

check Departure2
```

EXERCISE 2
Interrail: costraints


```
// If a person visits a city, the city remains forever among the visited cities
fact VisitingIsForever {
    all p: Person, c: City | always (c in p.visited_cities
        implies always c in p.visited_cities)
}

// If a city is not one of the visited cities, it has never been visited so far
fact NeverVisitedCities {
    all c: City, p: Person | always( c not in p.visited_cities implies
        historically c not in p.visited_cities)
}

// If a city is one of the visited cities, it has been visited at some point in the past
fact VisitedCities {
    all c: City, p: Person | always (c in p.visited_cities implies once move[p, c])
}
```

EXERCISE 2
Interrail: costraints

```
// In any state, a person either stays where he is or goes to another place

fact MovingOrNot {
    all p: Person | always (moved[p] or unmoved[p])
}
```

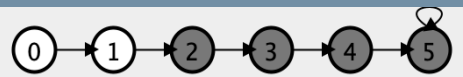
```
// After arriving in Milan, a person is no longer traveling

fact NoLongerTraveling{
    all p: Person |
        after (p.loc.name = Milan releases p.status = Travelling)
}
```

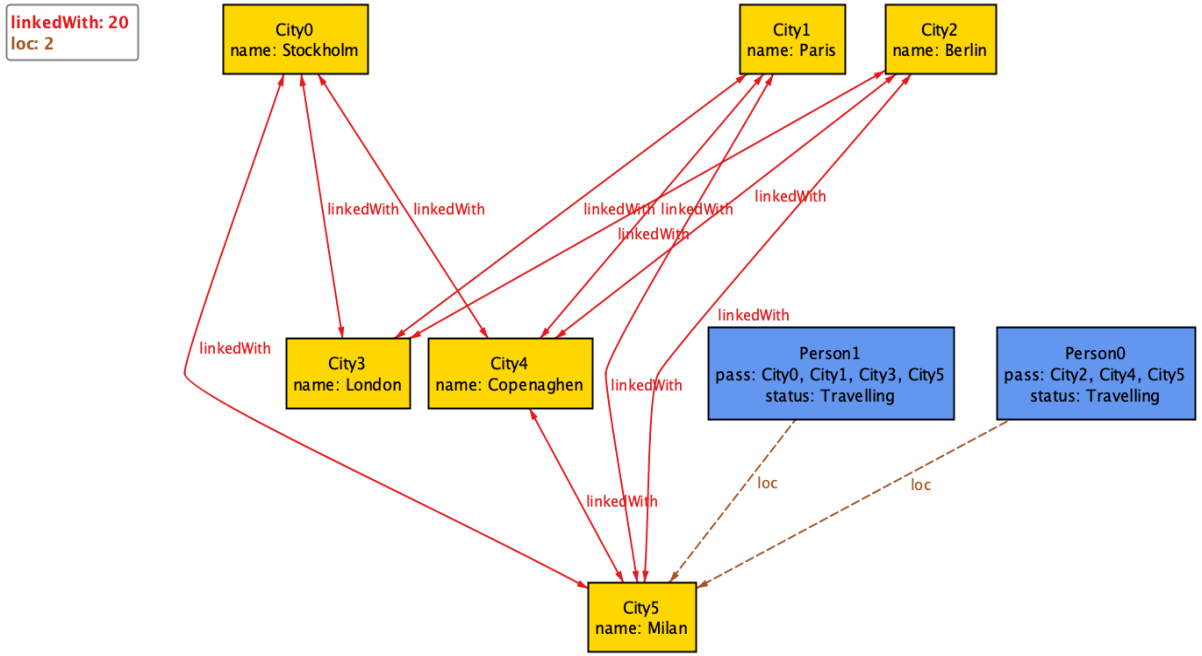
EXERCISE 2
Interrail: predicates

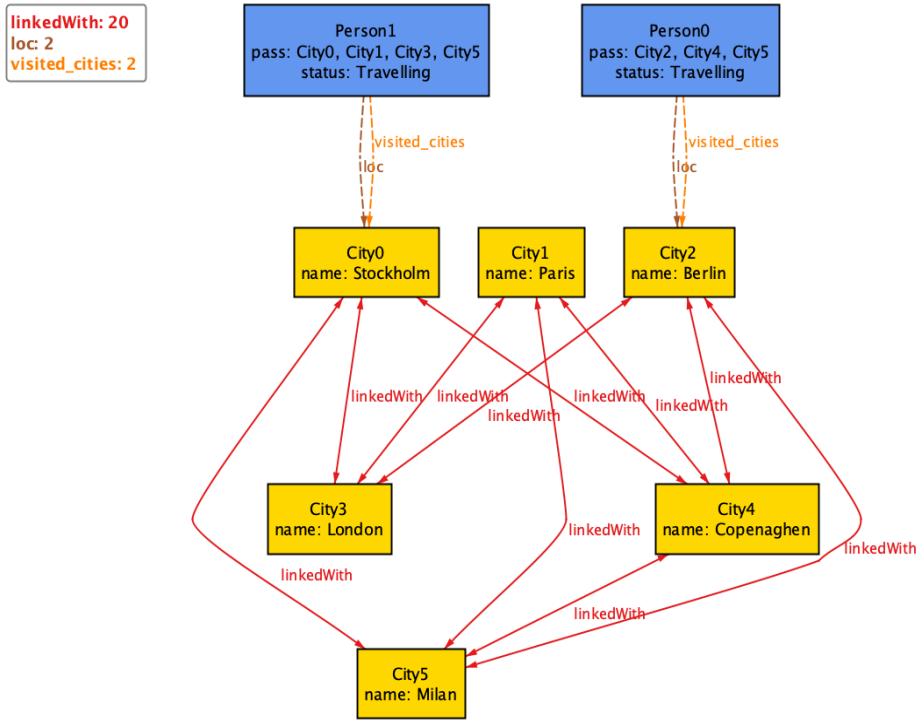
```
pred world1 {
   #City = 6 and #Person = 2
   #Person.visited_cities' = 2 ;
   #Person.visited_cities' = 4 ;
   #Person.visited_cities' = 6
}


run world1 for 10
```

and after

EXERCISE 2
Interrail: visualizer

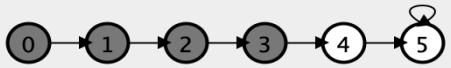


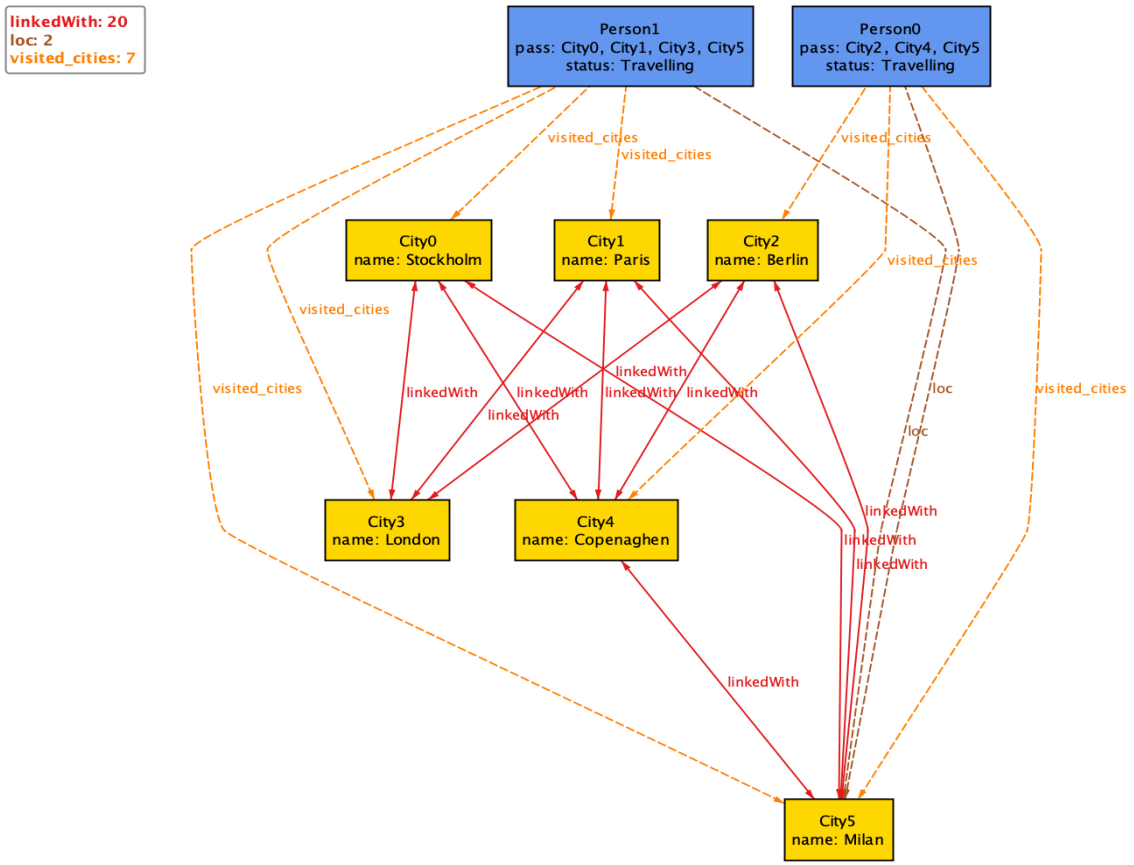


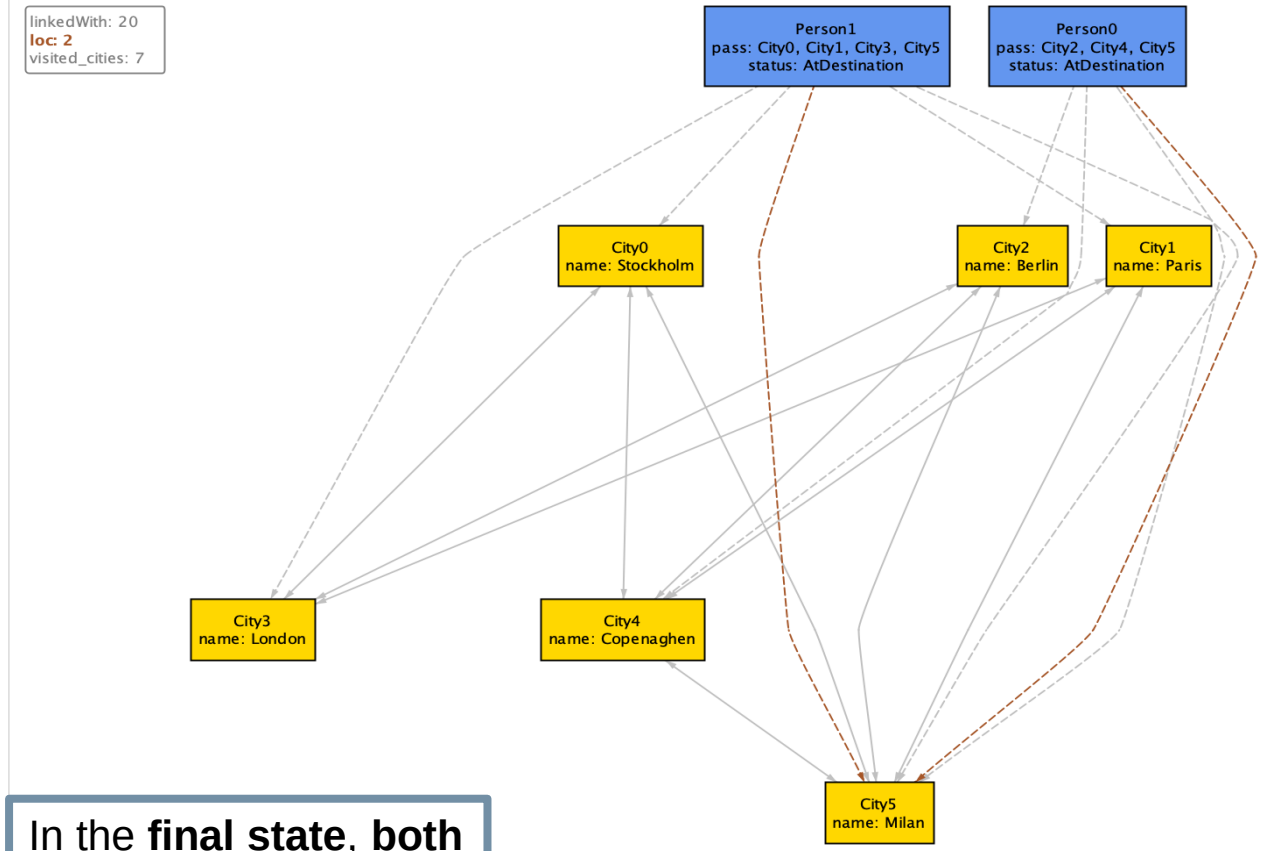


At the **beginning** of the trip, **both people** have **not yet visited any cities**

EXERCISE 2
Interrail: visualizer

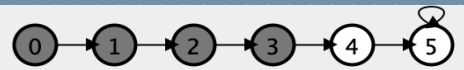


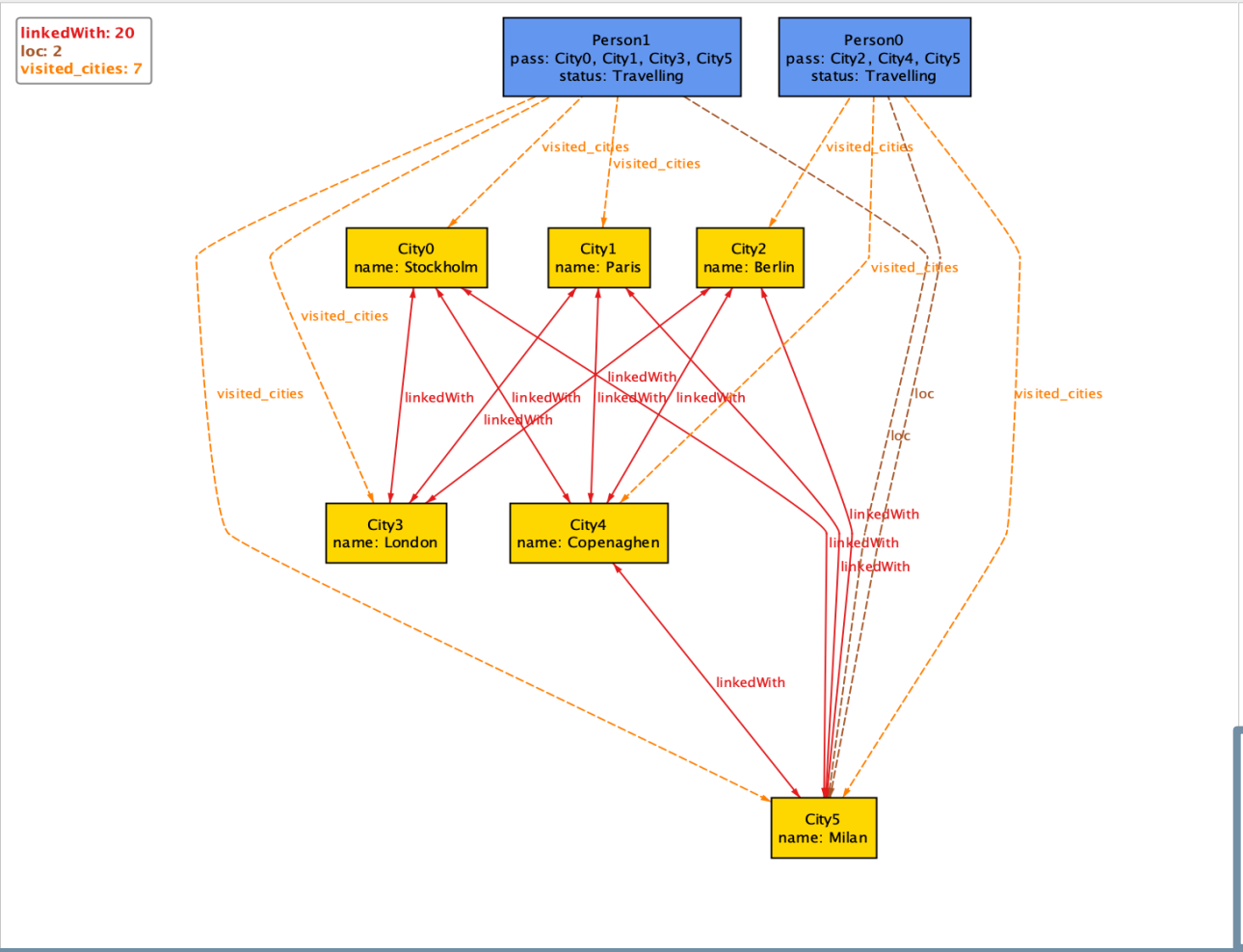


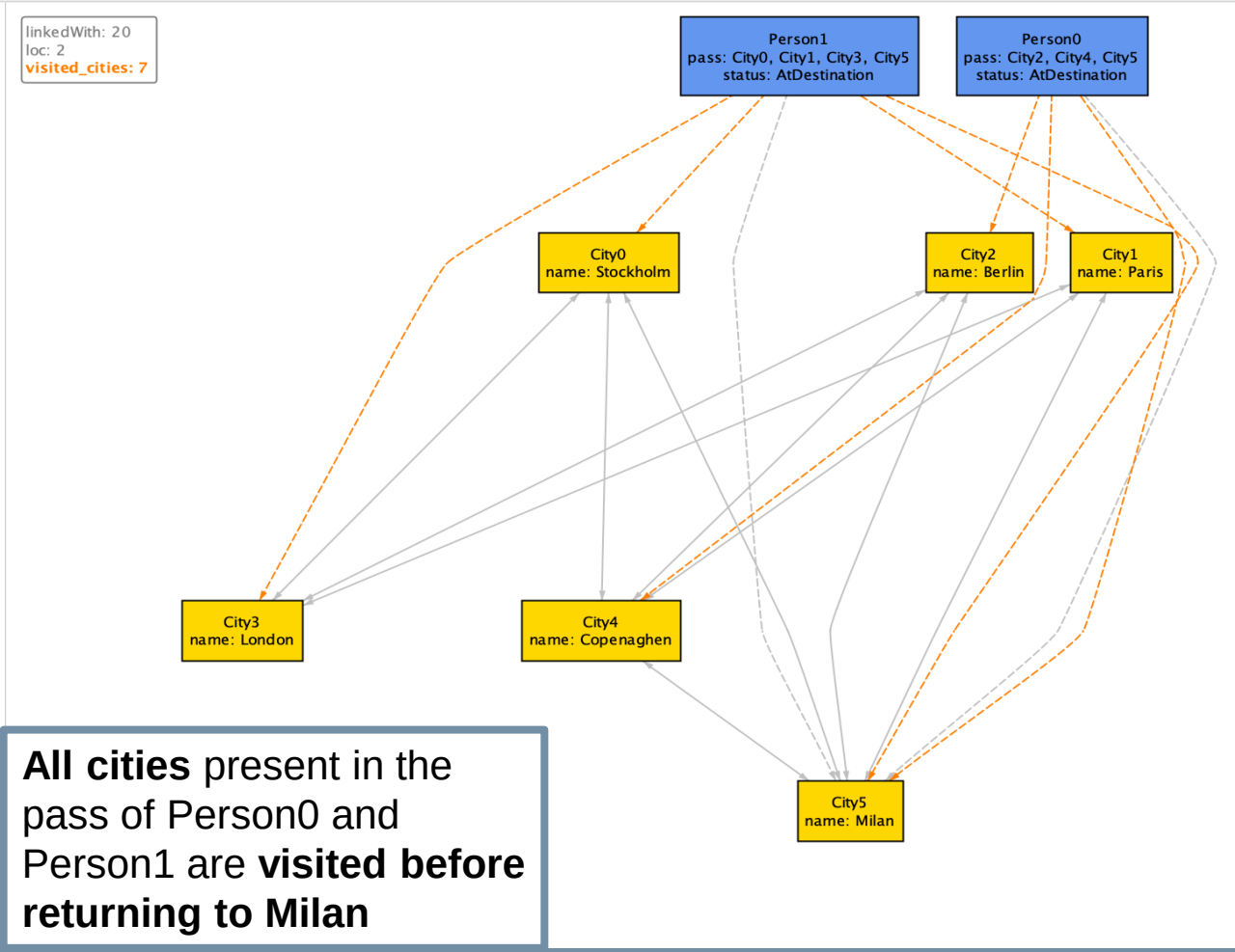


In the **final state**, **both** Person0 and Person1 **return to Milan**

EXERCISE 2
Interrail: visualizer

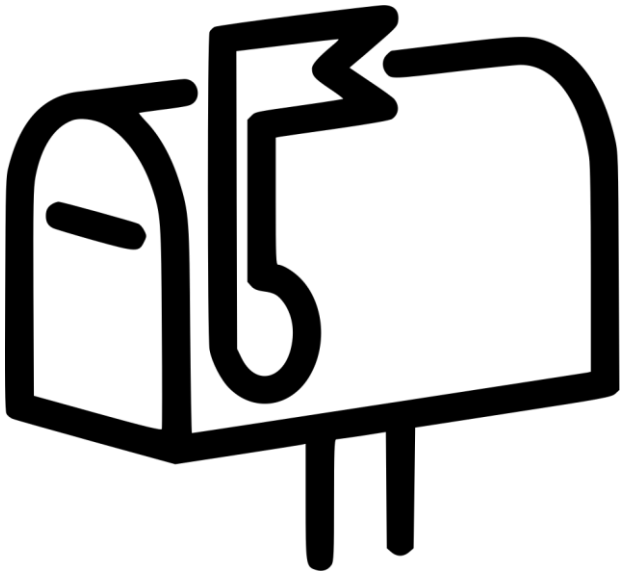






All cities present in the pass of Person0 and Person1 are **visited before returning to Milan**

EXERCISE 3
Message Deletion from Mailbox



Model a scenario in which **messages** can be **deleted** from a **mailbox** and later **restored**.

Hint: The **Message** ends up in a **Trash** and can be **restored** from there.

EXERCISE 3
Message Deletion from Mailbox


```
var sig Message {}
var sig Trash in Message {}

pred delete[m: Message] {
    m not in Trash
    Trash' = Trash + m
    Message' = Message
}

pred restore[m: Message] {
    m in Trash
    Trash' = Trash - m
    Message' = Message
}
```

```
pred empty {
    #Trash > 0
    after #Trash = 0
}

pred doNothing {
    Message' = Message
    Trash' = Trash
}

pred restoreEnabled[m:Message] {
    m in Trash
}
```

EXERCISE 3
Message Deletion from Mailbox


```
fact Behaviour {
    no Trash
    always {
        (some m: Message | delete[m] or restore[m]) or empty or doNothing
    }
}

run {}
```

EXERCISE 3
Message Deletion from Mailbox


```
assert restoreAfterDelete {
    always (all m : Message | restore[m] implies once delete[m])
}

check restoreAfterDelete for 10 steps
check restoreAfterDelete for 1.. steps

assert deleteAll {
    always ((Message in Trash and empty) implies after always no Message)
}

check deleteAll
```

EXERCISE 3
Message Deletion from Mailbox


```
check MessagesNeverIncreases {
    always (Message' in Message and #Message' = #Message)
}

check IfNotDeletedMessagesNotEmpty {
    (always all m : Message | not delete[m]) implies always not empty
}

// A deleted message can still be restored if the trash is not empty
assert restoreIsPossibleBeforeEmpty {
    always (all m:Message | delete[m]
            implies after ((empty or restore[m])
            releases restoreEnabled[m]))
}

check restoreIsPossibleBeforeEmpty for 3 but 1.. steps
```