



NTNU – Trondheim
Norwegian University of
Science and Technology

Scola: a Scenario-Oriented Language

Antoine Rauzy

Department of Mechanical and Industrial Engineering

S. P. Andersens veg 5, Valgrinda*3.306

Antoine.Rauzy@ntnu.no

Licenses & Versions



The present document is distributed under Creative Common License CC-BY-ND.

ScolaWizard is free software distributed by the AltaRica Association under GNU GPLv3 license.

Version	1.1.0
Date	02/09/2018

Agenda

- Introduction
- Getting Started
- Scenarios
- Base Types and Expressions
- Instructions
- Assertions
- Structuring Constructs
- References

Appendix

- Grammar
- Semantics
- Solutions to the Exercises

INTRODUCTION

Rational

This presentation specifies a **domain specific modeling language Scola**.

Scola stands for scenario-oriented language. It is a **textual** language.

Scola aims at supporting **systems architecture** studies by giving the system architect a mean to **describe** and to **play scenarios**.

Scola is relatively close to the BPMN formalism (White & Miers 2008), although simpler and more formal.

The name Scola and the idea of an scenario-based approach to systems engineering is borrowed from the work by Méli^{ss}a Issad's during her PhD thesis (Issad & al. 2018) .
The DSL presented here is however completely different from the one proposed by Méli^{ss}a, although both are remotely inspired from Milner's π -calculus (Milner 1999) .

Scola's Trinity

Scola involves three fundamental concepts:

- **System architecture**, i.e. the decomposition of the system under study into a hierarchy of nested components.
- **Scenarios**, i.e. sequences of actions that can be performed on the system and that may transform the system architecture.
- **Processes** that execute scenarios.

Scola provides constructs to structure models that are stemmed from object-oriented and prototype-oriented programming. To be more specific, Scola is based on the set of structuring constructs of S2ML (Batteux & al. 2015).

Scola Models

A Scola model is made of two parts:

- A description of the functional or physical **decomposition** of the system under study.
- A description of **scenarios** applying on this system.

The description of the system consists eventually in a hierarchy of nested blocks. Each **block** can compose any number of sub-blocks, ports and assertions. The system is represented by the top-most component, which is implicit.

A **port** is a holder for an atomic value (Boolean, integer, real, symbol, string...).

An **assertion** is an instruction that is applied to update the values of ports.

Blocks, ports and assertions can be dynamically created, destroyed and moved.

Each **scenario** can compose any number of sub-scenarios.

Scenarios are made of **states**, **tasks** and **gateways**. Tasks contain lists of **instructions** that create, destroy and modify the system description. Gateways make choices about scenarios.

It is possible to attach a scenario to a particular block (like lane in BPMN).

Scola models describe the evolution of the system via the execution of **processes**.

Organization of this Document

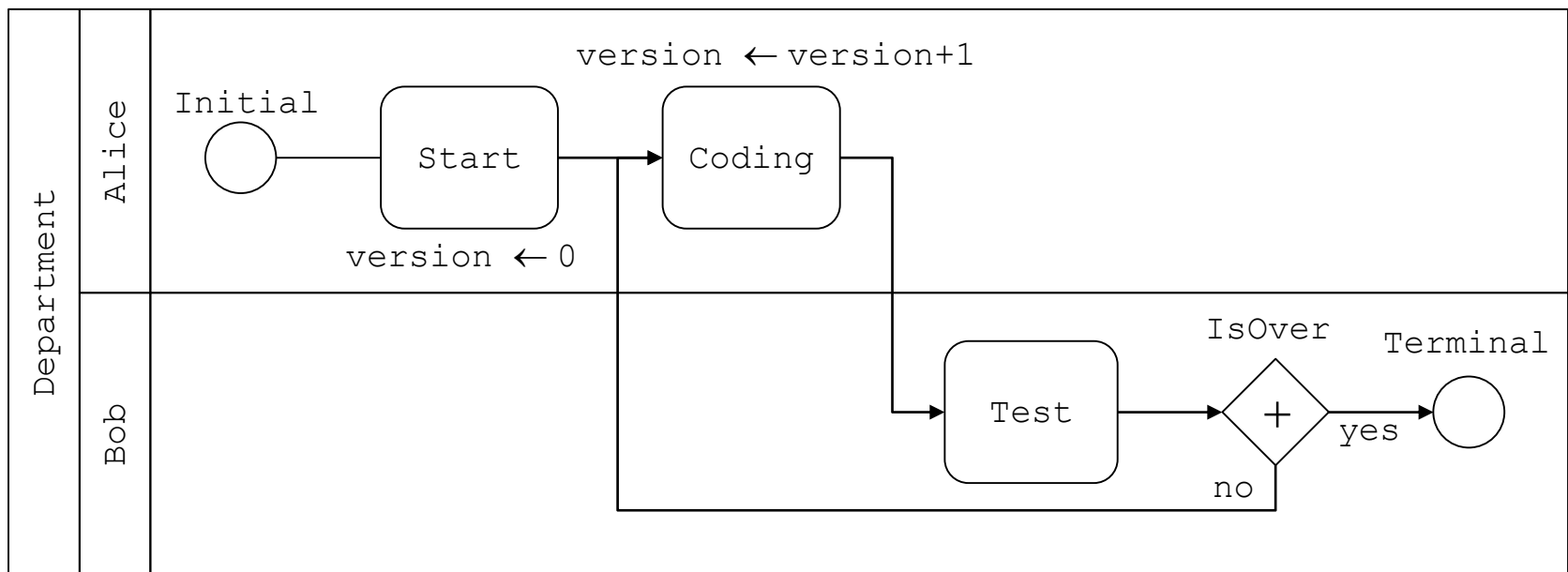
The remainder of this document is organized as follows.

- Section Getting Started introduces basics about Scola.
- Section Scenarios describes constructs to design scenarios, i.e. states, tasks and gateways.
- Section Base Types and Expressions presents types of ports and expressions.
- Section Instructions presents instructions that are used both in tasks and in assertions.
- Section Assertions presents assertions
- Section Structuring Constructs presents constructs to structure Scola models.
- Appendix Grammar gives the Backus-Naur Form of Scola's Grammar.
- Appendix Solutions to the Exercises gives the solution to the exercises proposed along the document.

GETTING STARTED

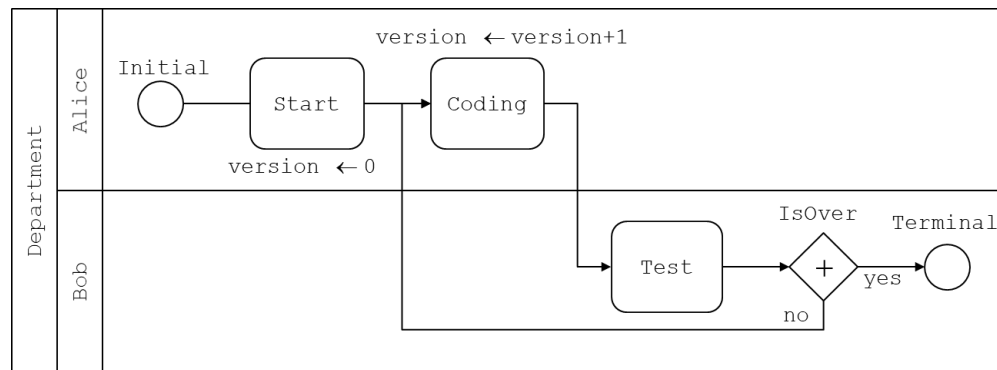
Scenarios & their graphical representation

Assume we want to represent the process of a small software development project in the R&D department of a company. Assume moreover that this project involves Alice and Bob. Alice and Bob works in turn: Alice codes the software, then Bob tests it. The project is achieved after a certain number of iterations. This progress of the project can be represented graphically as follows.



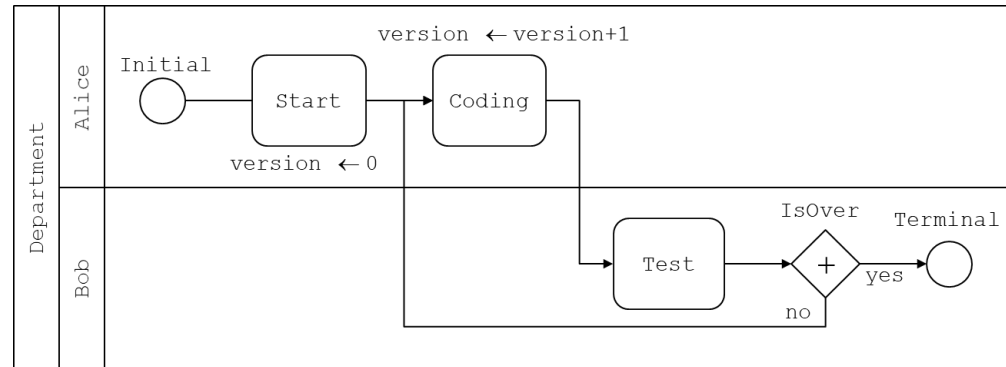
Scenarios, States, Tasks and Gateways

We have a system made of four components: The department, Alice, Bob and the software. Alice, Bob and the software "belong" to the department. Moreover, the software has a version number that evolves throughout the development process. The development process is represented as a scenario involving two sub-scenarios represented by lanes: one lane for Alice and one lane for Bob.



1. The scenario starts in the state `Initial`.
2. Alice performs the task `Start` in which a port (variable) `version` is reset to 0.
3. Alice performs the task `Code`, in which `version` is incremented.
4. Bob performs the task `Test`.
5. There is a choice, the choice gateway `IsOver`. If the branch `yes` is chosen, the scenario continues with the state `Terminal`, otherwise it goes back to task `Code` (of Alice).
6. The scenarios ends on task `Terminal`,

Code: System



```

block Department
  block Alice
  end
  block Bob
  end
  block Software
    integer version 0
  end
end

```

- Systems are represented by hierarchies of nested *blocks*. Each block represents thus a component or a function of the system.
- Blocks may also contain ports. *Ports* hold constant values (Booleans, integers, reals, symbols or strings).
- Like blocks, ports have a name. In addition, they are declared with a default value.
- Blocks are thus *containers* for blocks and ports. One says that they *compose* blocks and ports. Within a block, all objects should have different names.
- In our example, the system (which is an implicit block) composes on block `Department`, which itself composes three blocks: `Alice`, `Bob` and `Software`. The block `Software` composes the integer port `version` whose default value is 0.

Code: Scenario

scenario Development

scenario AliceLane **as** Department.Alice

state Initial

task Start

set owner.Software.version 0

end

task Code

set owner.Software.version (**add owner**.Software.version 1)

end

next Initial Start

next Start Code

end

scenario BobLane **as** Department.Bob

task Test **end**

choice IsOver

case yes

case no

end

state Terminal

next Test IsOver

end

next AliceLane.Code BobLane.Test

next BobLane.IsOver.no AliceLane.Code

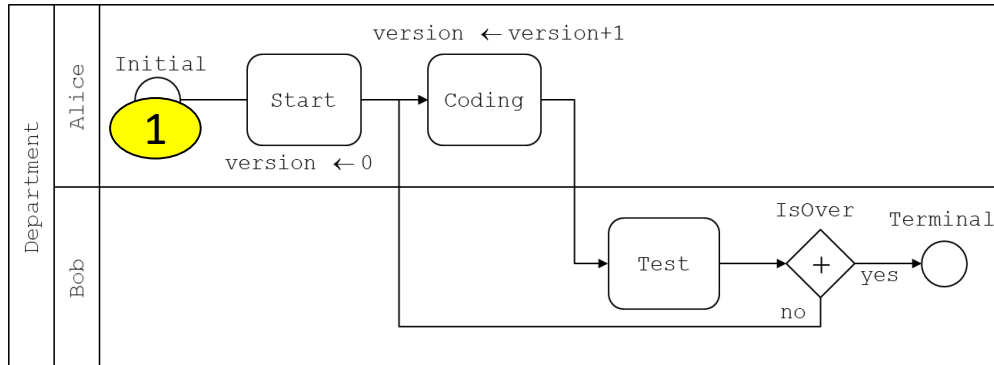
next BobLane.IsOver.yes BobLane.Terminal

end

- *Scenarios* are containers for states, tasks, gateways and other scenarios.
- Tasks are containers for *instructions*.
- *Next directives* chain states, tasks and gateways. They are represented with arrows.
- The dot notation is used to refer elements inside containers. In the container Dialog, AliceLane.Code refers to the task Code of the sub-container Alice. The keyword owner refers to the parent block.
- The order of declarations is irrelevant.

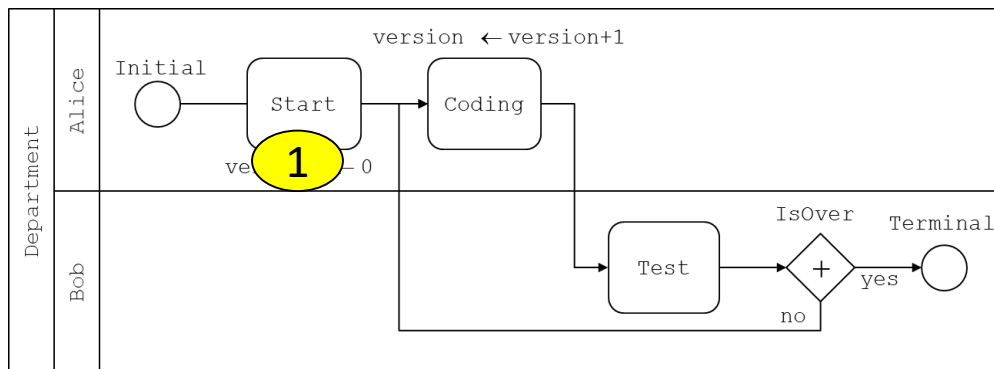
Executions & Processes (1)

1



Scenarios are executed by processes.
Here a process number 1 is created on the state Initial.

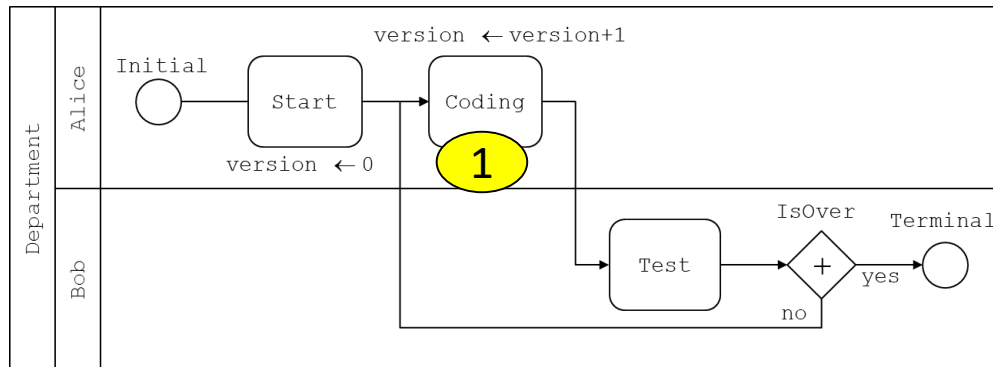
2



The process 1 then moves to task Start.
A process can perform a task if it can perform all instructions of the task.
Tasks are atomic: instructions are performed without interruption.

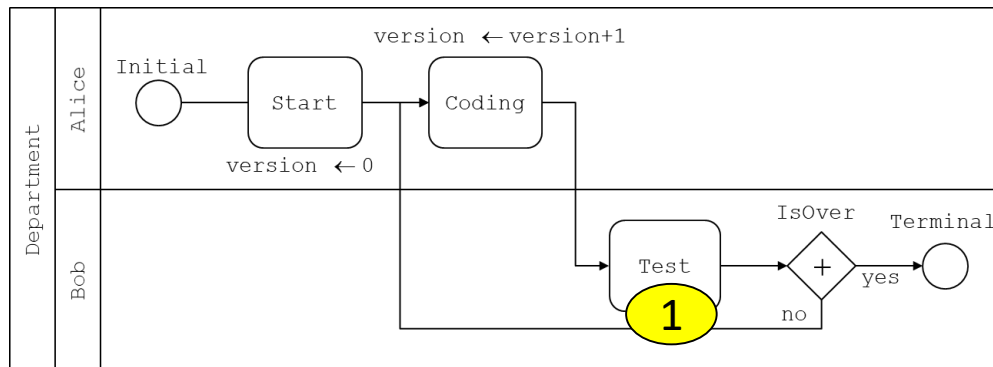
Executions & Processes (2)

3



`Software.version = 0`

4



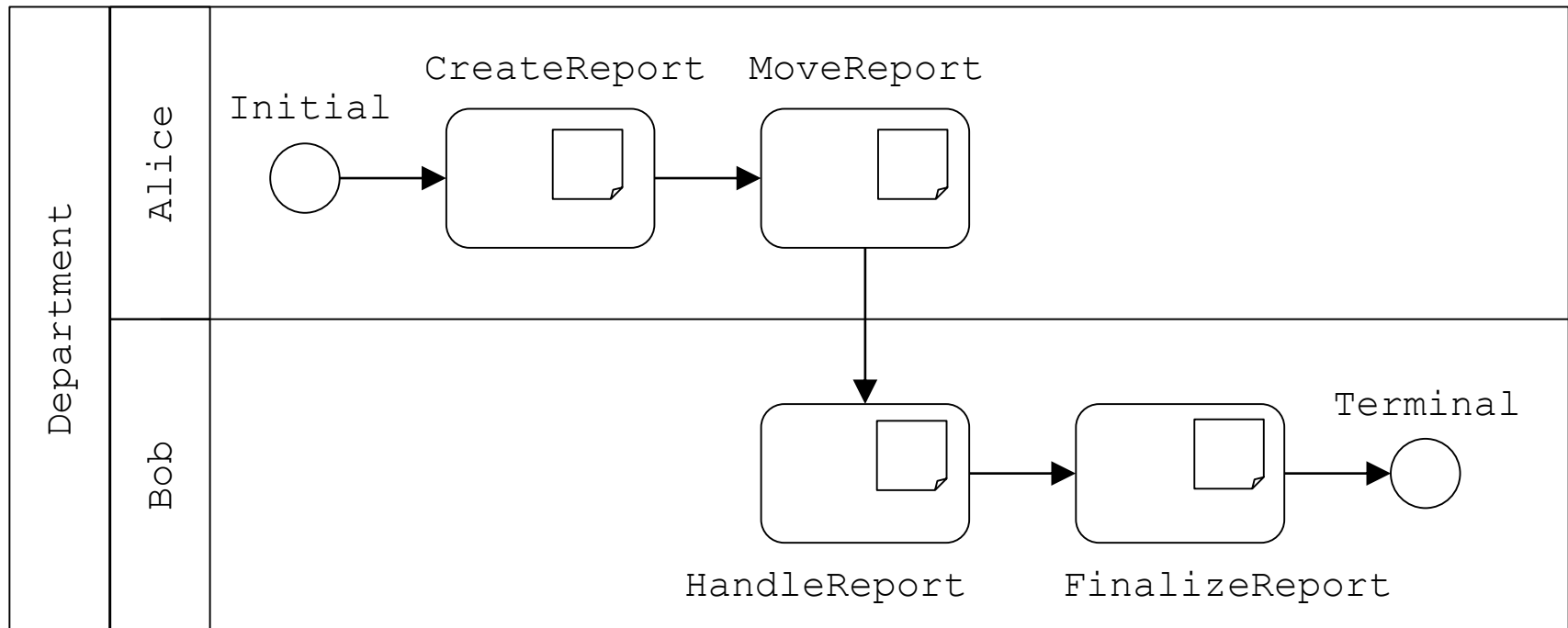
`Software.version = 1`

And so on...

Mobile Components

Scola makes it possible to describe mobile components, i.e. components that are possibly dynamically created, destroyed and moved from place to place in the system. As an illustration, assume that Alice wants now to write the activity report of the project that Bob will be in charge of finalizing. This report is a document that will be created by Alice, then moved to Bob.

This scenario can be represented graphically:



Example: Activity Report (1)

```
block Department
  block Alice end
  block Bob end
  block Software
    integer version 0
  end
end
```

Same system as before!

A domain, i.e. a finite set of symbolic constants is created to encode the status of the report.

```
domain ReportStatus {CREATED, MODIFIED, FINALIZED} end
```

```
scenario ActivityReport
  scenario AliceLane as Department.Alice
    ...
  end
  scenario BobLane as Department.Bob
    ...
  end
  next AliceLane.MoveReport BobLane.HandleReport
end
```

The next directive applies to tasks belonging to different storylines. It is here declared at the parent level.

Example: Activity Report (2)

```
scenario AliceLane as Department.Alice
  state Initial
  task CreateReport
    new block report
    new ReportStatus report.status CREATED
    new string report.title "Activity Report"
    new string report.content "Alice's contribution"
  end
  task MoveReport
    move report main.Department.Bob.report
  end
  next Initial CreateReport
  next CreateReport MoveReport
end
```

- New blocks and ports are dynamically created.
- When a block or a port is created, it is inserted in the block the current storyline refers to (here Department.Alice).
- The keyword "main" refers to the model.
- Moving a block or a port requires that no item with the same name belongs to the target block. The process cannot perform the task until this condition is realized.

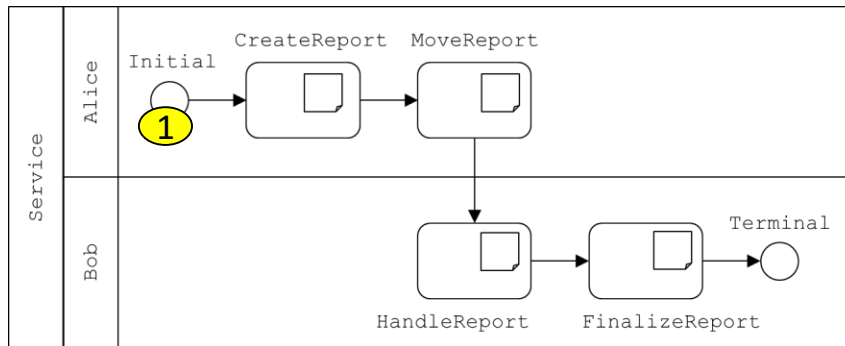
Example: Activity Report (3)

```
scenario BobLane as Department.Bob
  task HandleReport
    set report.content (append report.content " & Bob's contribution")
    set report.status MODIFIED
  end
  task FinalizeReport
    set report.status FINALIZED
  end
  state Terminal
  next HandleReport FinalizeReport
  next FinalizeReport Terminal
end
```

- The keyword "owner" refers to the parent container.
- Receptions of items are blocking: the process waiting the item stops until it receives the item.

Example: Activity Report (4)

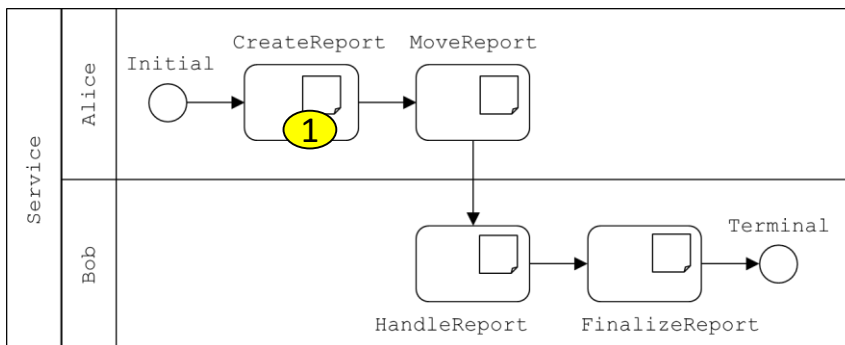
1



```

block Department
block Alice
block Bob
  
```

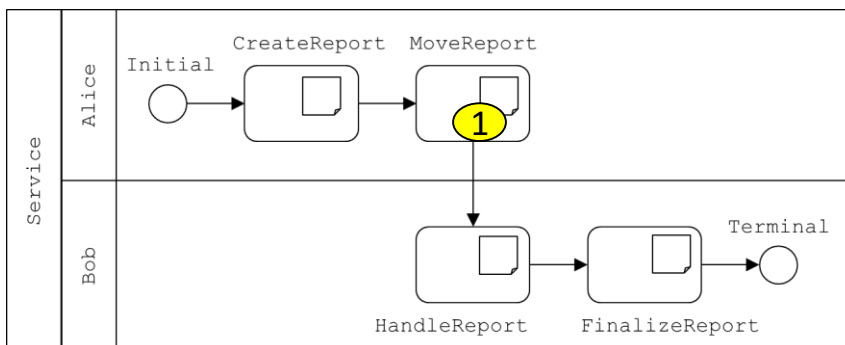
2



```

block Department
block Alice
block Bob
  
```

3

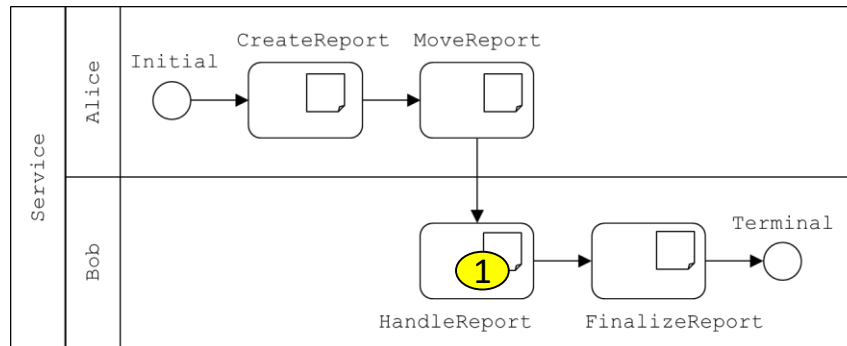


```

block Department
block Alice
  block report
    port status CREATED
    port title "Activity Report"
    port content "Alice's contribution"
  block Bob
  
```

Example: Activity Report (5)

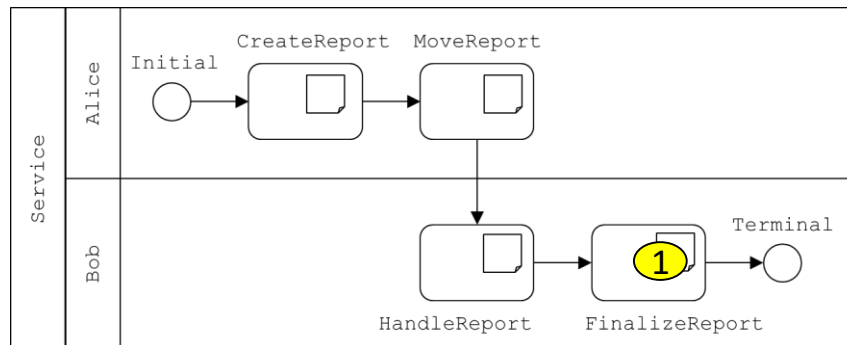
4



```

block Department
block Alice
block Bob
  block report
    port status CREATED
    port title "Activity Report"
    port content "Alice's contribution"
  
```

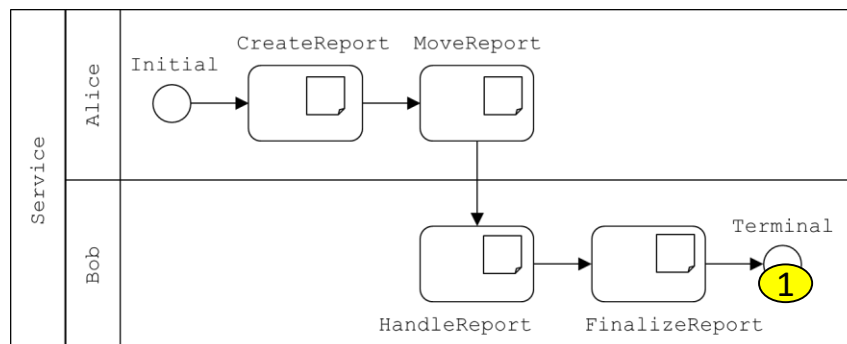
5



```

block Department
block Alice
block Bob
  block report
    port status MODIFIED
    port title "Activity Report"
    port content "Alice's contribution &
      Bob's contribution"
  
```

6



```

block Department
block Alice
block Bob
  block report
    port status FINALIZED
    port title "Activity Report"
    port content "Alice's contribution &
      Bob's contribution"
  
```

Wrap-Up

- Scola models are made of systems (represented as hierarchies of blocks) and scenarios applying on these systems.
- Each scenario describes a particular facet or function of the system. There may be many scenarios applied to the same system.
- **Blocks** can **compose** other blocks and ports.
- **Ports** hold constant values (Boolean, integers, reals, symbols or strings).
- Blocks and ports can be dynamically created, destroyed and moved.
- **Scenarios** can compose other scenarios and be applied to a particular sub-system (which is graphically represented by a lane).
- Scenarios are made of **states** (represented by circles), **tasks** (represented by rounded rectangles) and **gateways** (represented by diamonds) which are linked together by means of **next directives** (represented by arrows).
- Tasks can compose **instructions** that modify the state of the system.
- Scenarios are executed by **processes**. The semantics of a Scola model is the set of all possible executions starting with a process located on each **initial state** (i.e. states without predecessors) and normally ending when all active processes have reached a **terminal state** (i.e. a state without successor).

Exercises

Exercise: Hello World!

Consider a system without subsystem and that performs a single actions: saying "Hello World". Give the code for this scenario and represent it graphically. Execute it.

Exercise: Greatest Common Divisor

Design a Scola model that calculates the greatest common divisor (GCD) of two integers. Execute it with $a=96$ and $b=81$.

Hint: recall that $\text{GCD}(a, a) = a$ and that $\text{GCD}(a, b) = \text{GCD}(a, b-a)$ if $a < b$.

Exercise: Syracuse Problem (Collatz conjecture)

Design a Scola model that takes any integer n and performs the following operations:

- If n is equal to 1, the execution stops.
- If n is even ($n \bmod 2 = 0$), then the execution goes on with $n/2$.
- If n is odd ($n \bmod 2 = 1$), then the execution goes on with $3n+1$.

Execute this model for $n=19$.

Scola operations for multiplication and the modulo are respectively `mul` and `mod`.

Exercises

Exercise: At the restaurant.

At the restaurant, the client orders a pizza to the waiter. The waiter transmit the order to the cook, who bakes the pizza. Once the pizza is baked, the cook gives it to the waiter, who brings it to the client. Eventually, the client eats the pizza.

Represent and execute this scenario.

Exercise: Car assembly

In a car assembly line, the first station paints the car's body, the second assemble the engine and the third the wheels.

Represent and execute this scenario.

SCENARIOS

States and Tasks

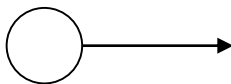
In Scola, there is a unique type of **state** and a unique type of **task**.

States can be however sorted into three categories:

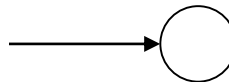
- **Initial states**, i.e. states that do not occur as the right member of a next directive.
- **Terminal states**, i.e. states that do not occur as the left member of a next directive.
- **Intermediate states**, the other.

Initial and terminal states play a very important role in the definition of scenarios. Intermediate states are accepted for the sake of the completeness, although it is always possible to remove them from scenarios without changing the semantics. States are graphically represented as circle.

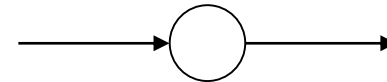
initial state



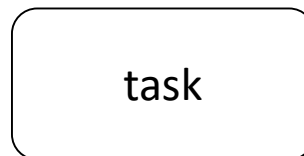
terminal state



intermediate state



Tasks are containers for instructions. They are represented by rounded rectangles.

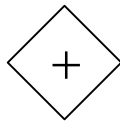


Gateways

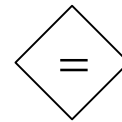
Scola provides 7 types of **gateways**.



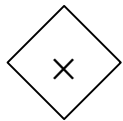
test



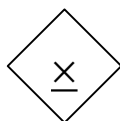
choice



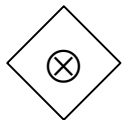
meet



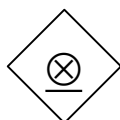
fork



join

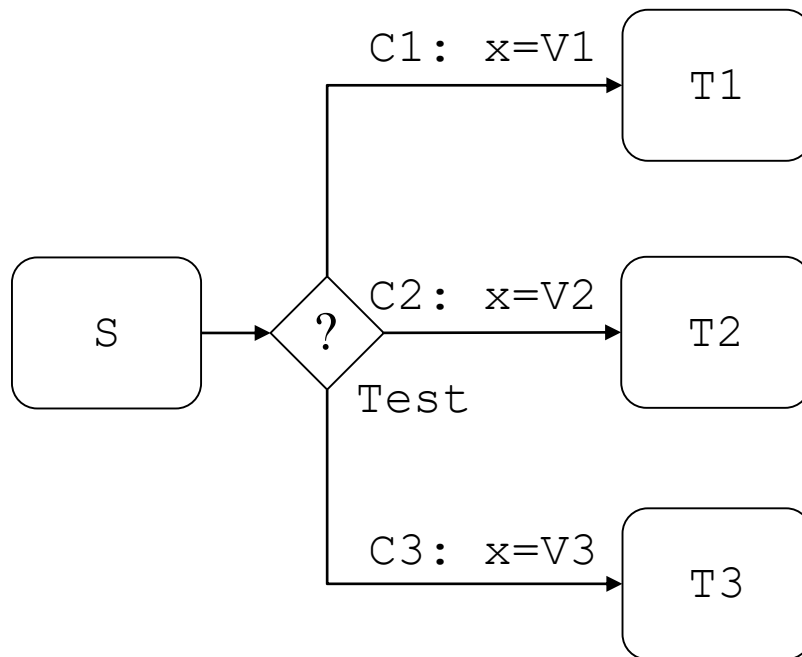


split



merge

Test Gateways

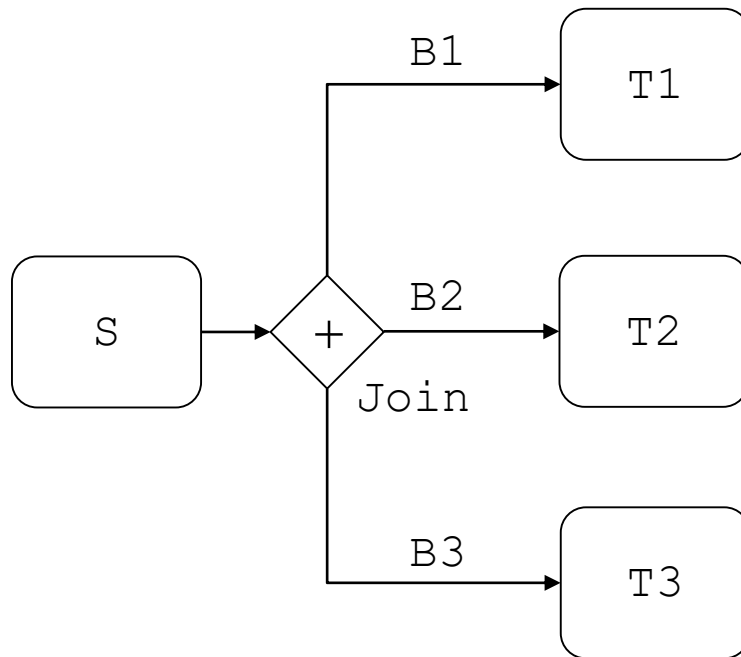


```
task S ... end
task T1 ... end
task T2 ... end
task T3 ... end
test Test
    case C1 (eq x V1)
    case C2 (eq x V2)
    case C3 (eq x V3)
end
next S Test
next Test.C1 T1
next Test.C2 T2
next Test.C3 T3
```

A **test gateway** can have any number of (output) case branches.

A process (coming from the task *S*) located on the test gateway *Test* can move forward if one and only one of the conditions labelling the case branches is verified.

Choice Gateways

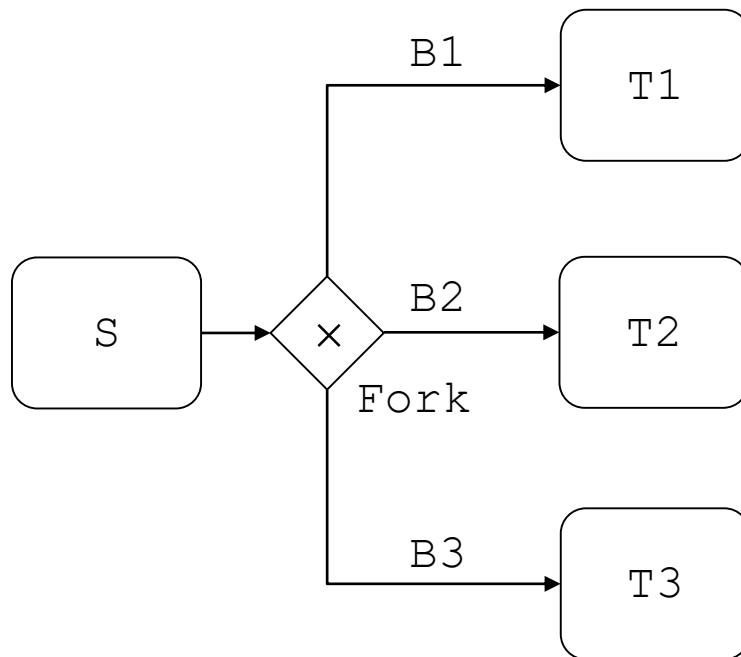


```
task S ... end
task T1 ... end
task T2 ... end
task T3 ... end
choice Choice
  branch B1
  branch B2
  branch B3
end
next S Test
next Choice.B1 T1
next Choice.B2 T2
next Choice.B3 T3
```

A **choice gateway** can have any number of (output) branches.

A process (coming from the task S) located on the choice gateway `Choice` can move forward on any of the (output) branches.

Fork Gateways

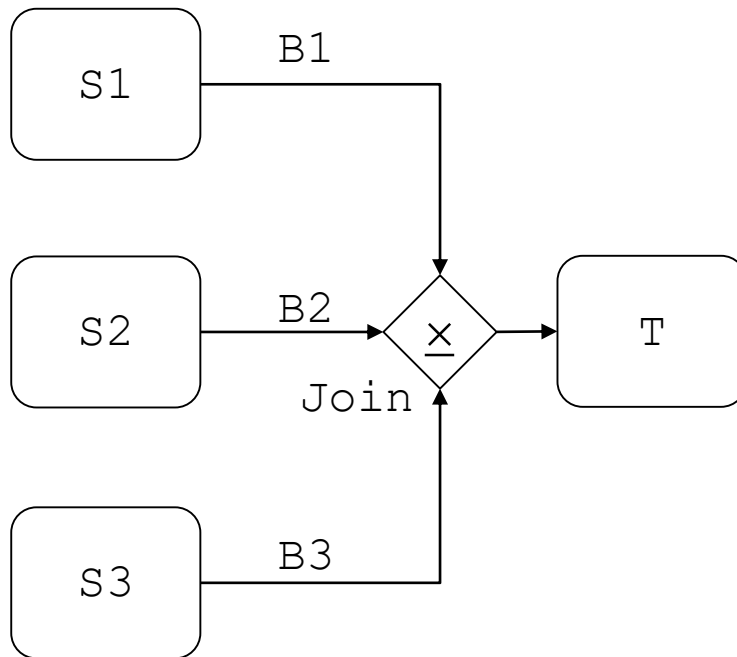


```
task S ... end
task T1 ... end
task T2 ... end
task T3 ... end
fork Fork
    branch B1
    branch B2
    branch B3
end
next S Test
next Fork.B1 T1
next Fork.B2 T2
next Fork.B3 T3
```

A **fork gateway** can have any number of (output) branches.

A process (coming from the task S) located on the fork gateway `Fork` can move forward. It is then deactivated (killed) and a new process is created on each branch of `Fork`. These new processes are not related to the process that created them.

Join Gateways



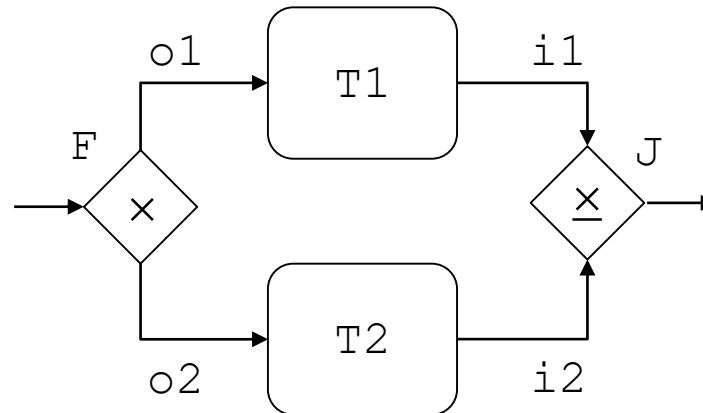
```
task S1 ... end
task S2 ... end
task S3 ... end
task T ... end
join Join
    branch B1
    branch B2
    branch B3
end
next S Test
next S1 Join.B1
next S2 Join.B2
next S3 Join.B3
next Join T
```

A **join gateway** can have any number of (input) branches.

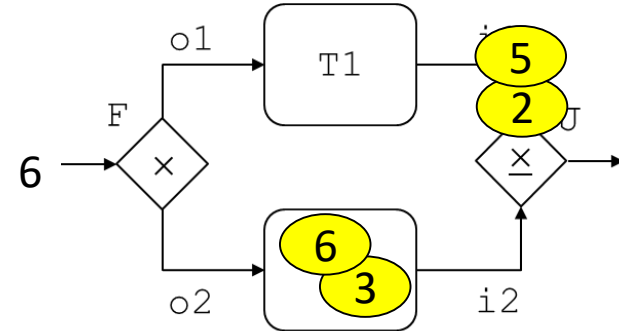
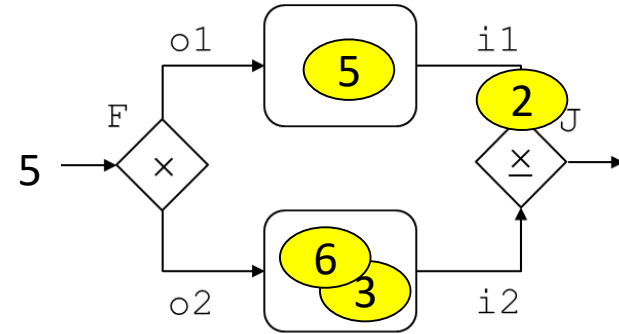
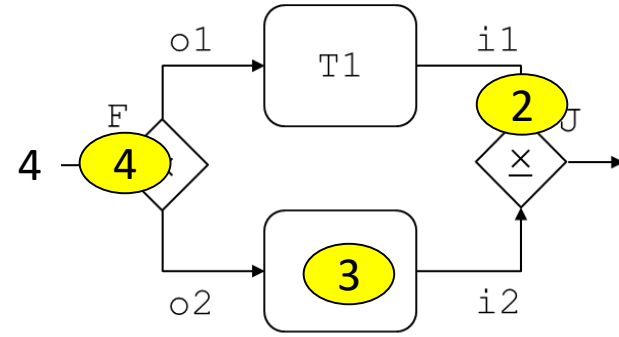
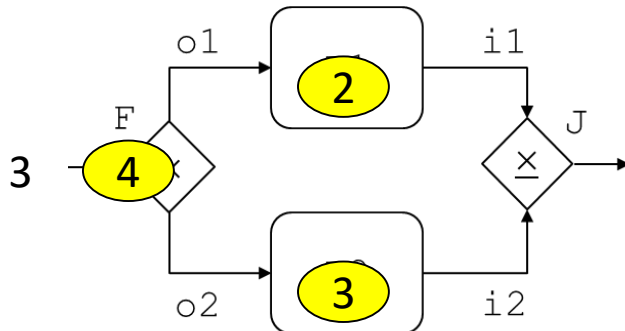
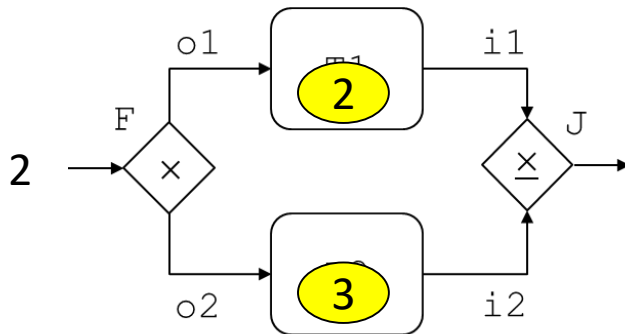
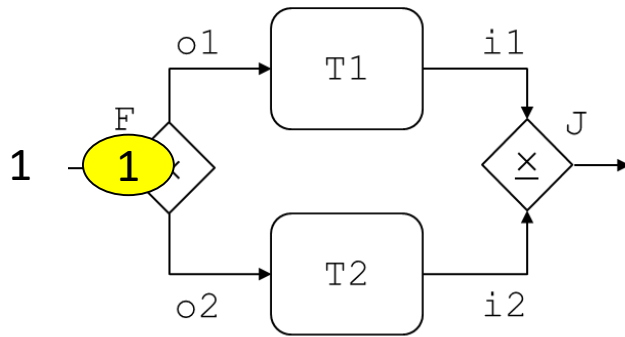
It does the opposite operation of a fork gateway. Processes arriving on input branches are stored into queues (first in, first out). When there is a process in the queue of each input branch (B1, B2, B3), they can move forward, which means that they are deactivated (killed) and that a new process is created on task T.

Example: Production Line (1)

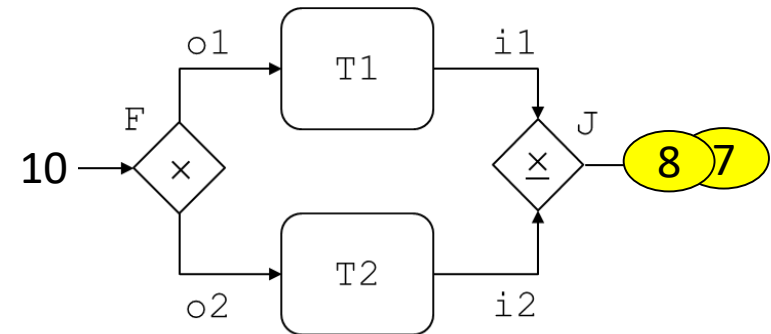
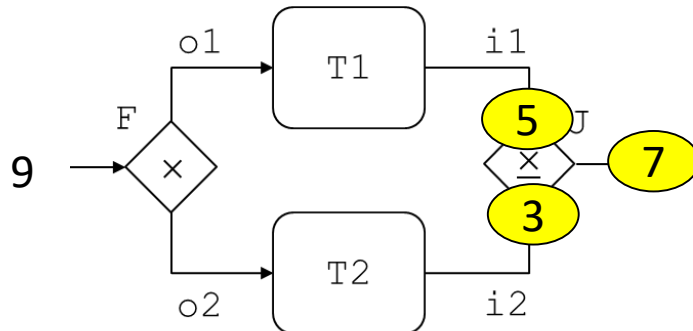
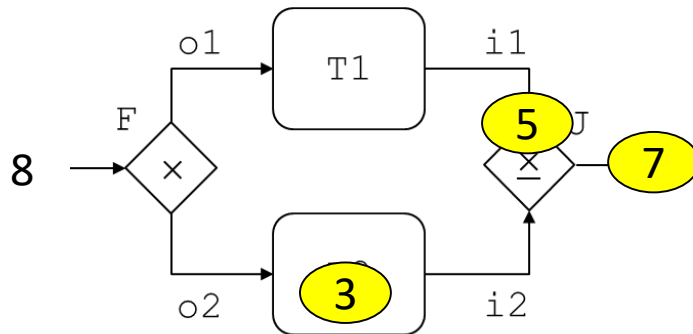
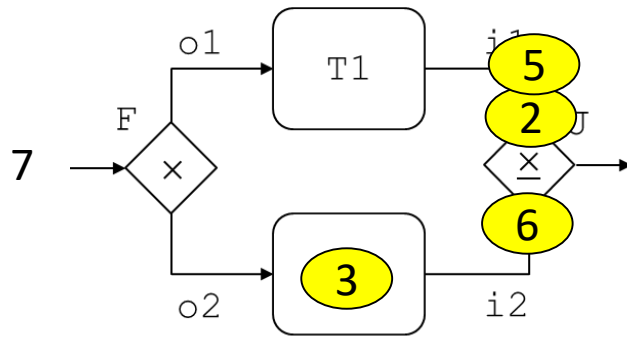
Consider (part of) a production line in which parts made of two components arrive on a conveyor belt to a first treatment unit F (represented by a fork gateway) where they are separated. Once separated, components are sent respectively units of type $T1$ and $T2$. When treatments performed by units $T1$ and $T2$ are done, components are joined together in a unit J (represented by a join gateway). The important point here is that it does not matter to assemble components coming from different parts, as all the parts are the same.



Example: Production Line (2)

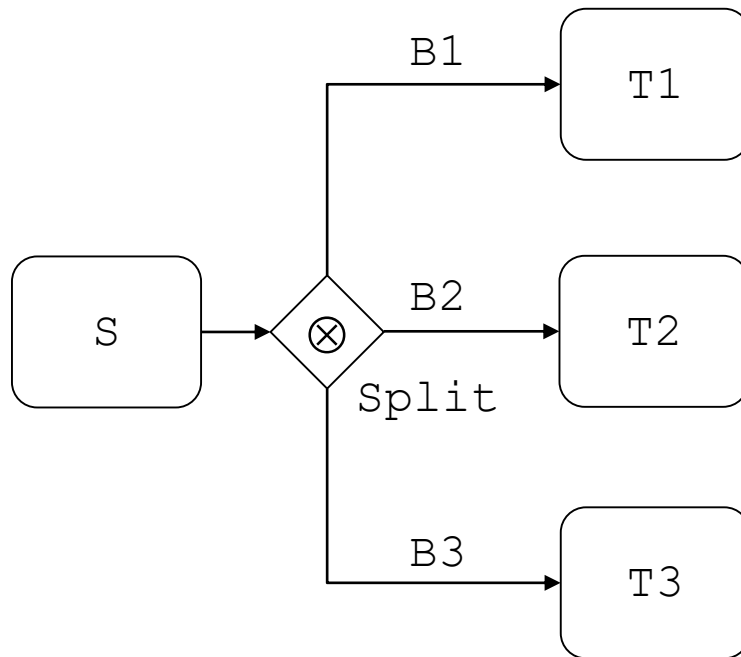


Example: Production Line (3)



And so on...

Split Gateways

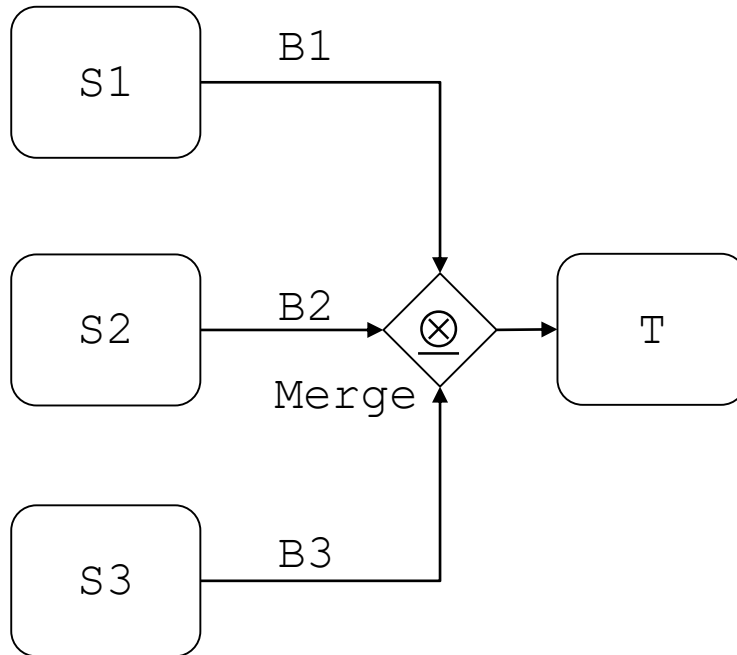


```
task S ... end
task T1 ... end
task T2 ... end
task T3 ... end
split Split
    branch B1
    branch B2
    branch B3
end
next S Test
next Split.B1 T1
next Split.B2 T2
next Split.B3 T3
```

A **split gateway** can have any number of (output) branches.

Split gateways are similar to fork gateways except that they link the deactivated process (parent process) with the created processes (children processes). A process (coming from the task `S`) located on the split gateway `Split` can move forward. It is then deactivated (killed) and a new child process is created on each branch of `Split`. These new processes are children of the killed process.

Merge Gateways



```
task S1 ... end
task S2 ... end
task S3 ... end
task T ... end
merge Merge
    branch B1
    branch B2
    branch B3
end
next S Test
next S1 Merge.B1
next S2 Merge.B2
next S3 Merge.B3
next Merge T
```

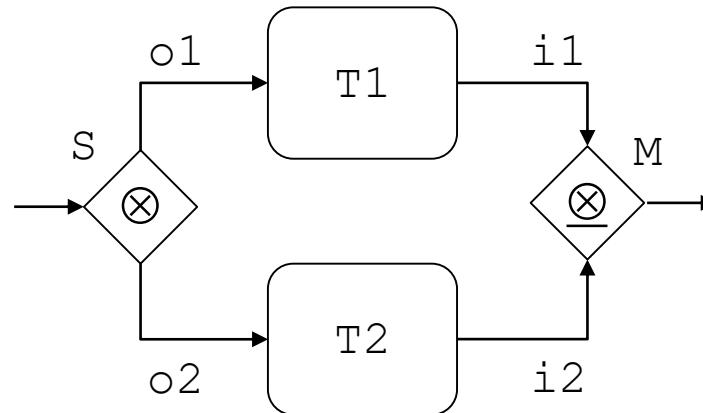
A **merge gateway** can have any number of (input) branches.

It does the opposite operation of a split gateway. Processes arriving on input branches are stored. When all the children processes of a parent process are in the sets associated with input branches (B1, B2, B3) of Merge, they can move forward, which means that they are deactivated (killed) and that the parent process is reactivated on task T.

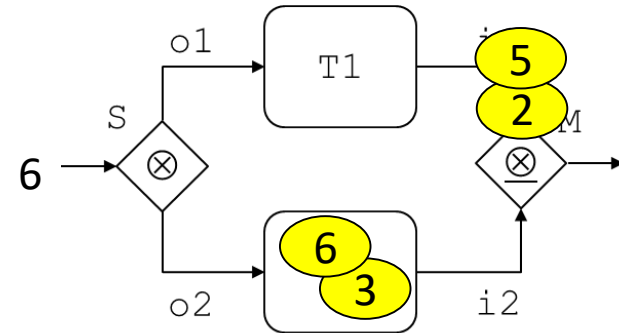
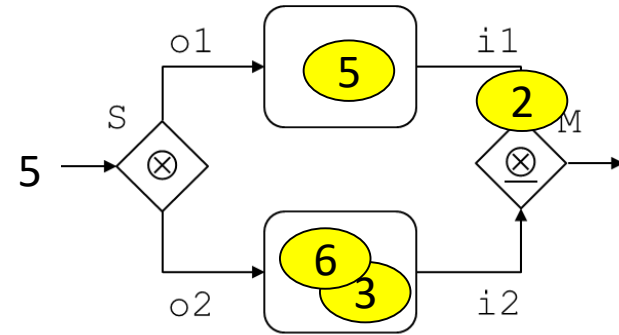
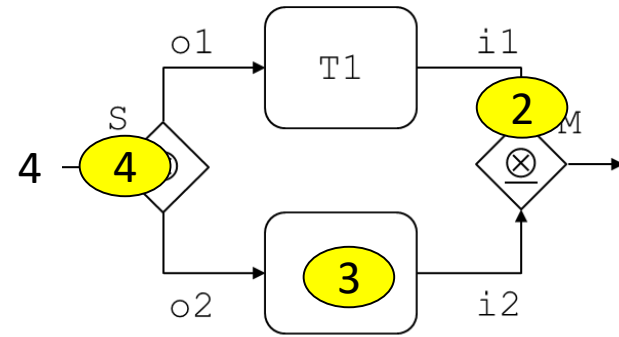
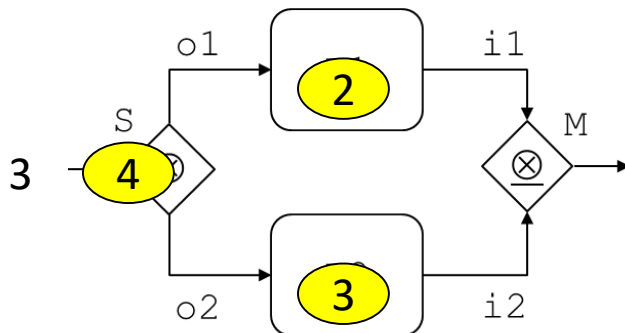
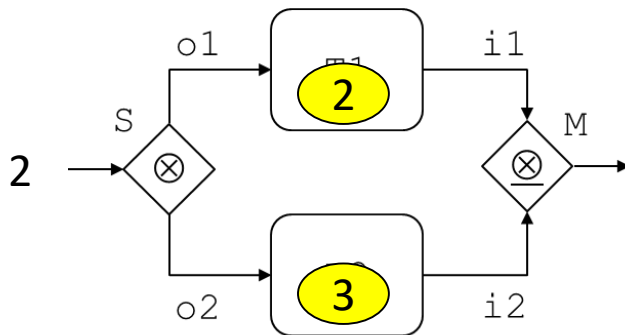
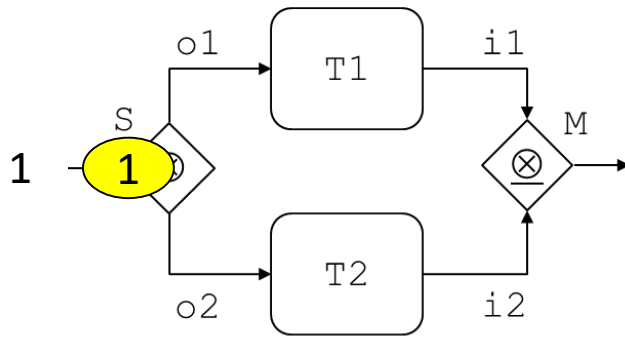
Example: Production Line revisited (1)

Consider (part of) a production line in which parts made of two components arrive on a conveyor belt to a first treatment unit S (represented by a split gateway) where they are separated. Once separated, components are sent respectively units of type $T1$ and $T2$. When treatments performed by units $T1$ and $T2$ are done, components are reassembled together in a unit M (represented by a merge gateway).

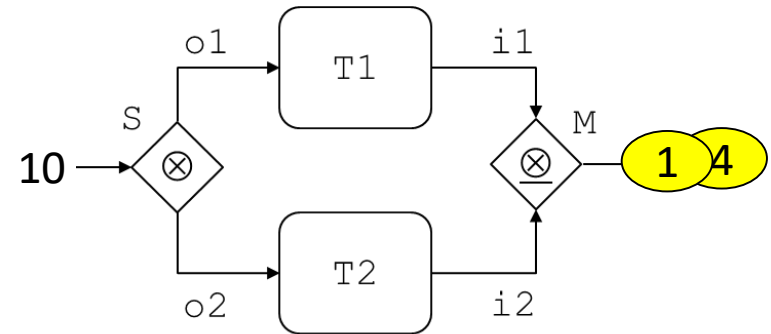
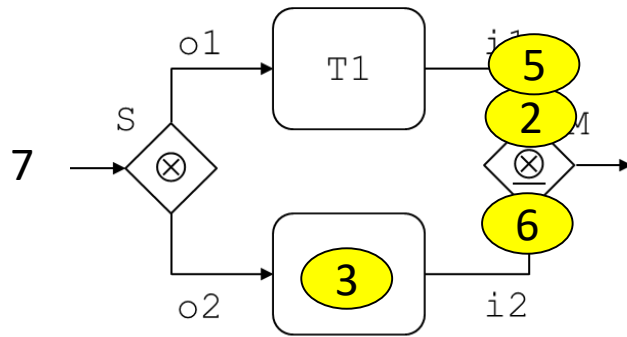
The important point here is that components of the same part must be re-assembled together.



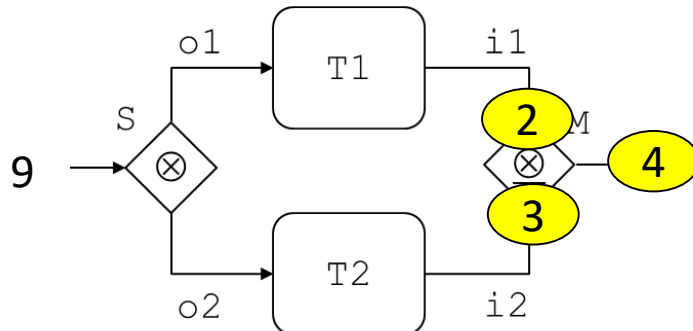
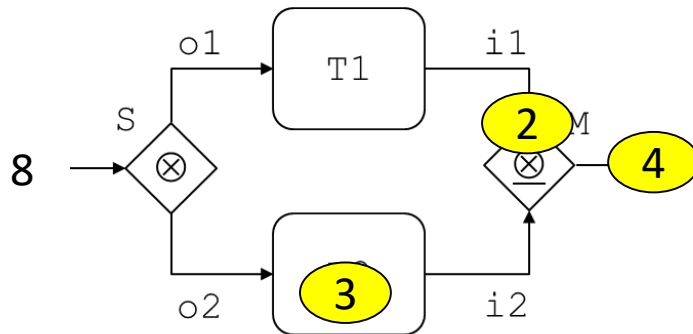
Example: Production Line revisited (2)



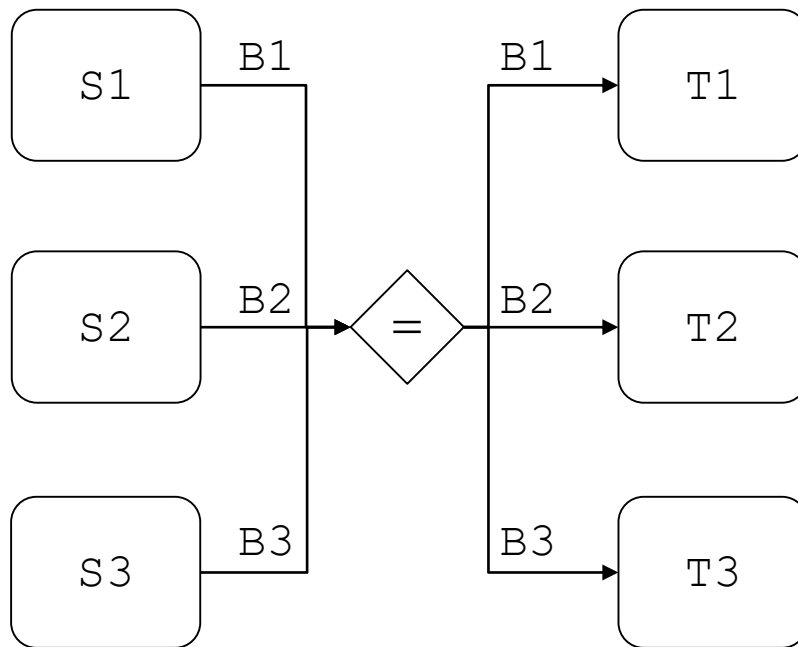
Example: Production Line revisited (3)



And so on...



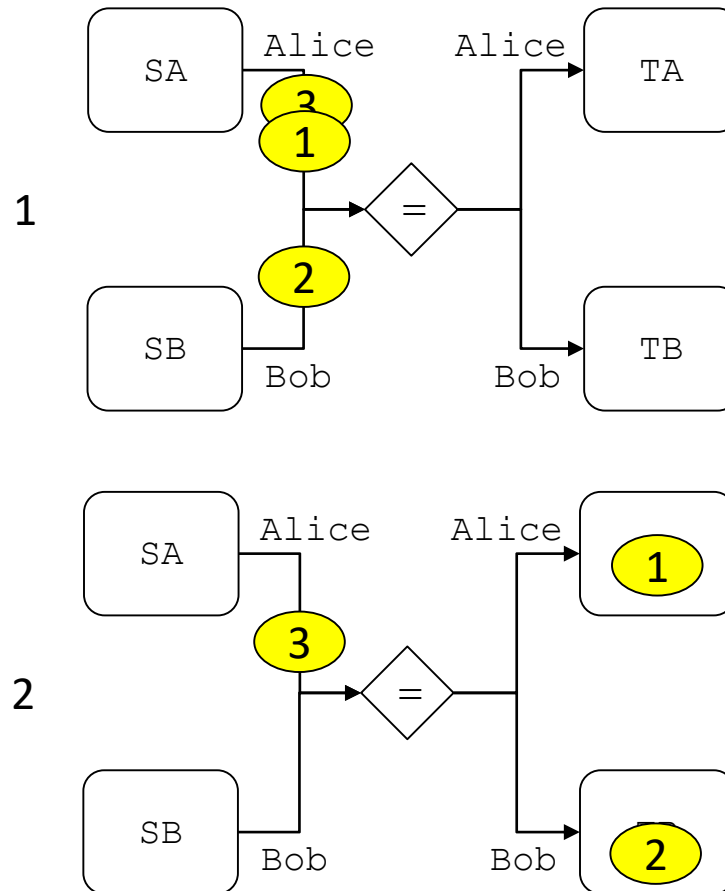
Meet Gateways



```
meet Meet
  branch B1
  branch B2
  branch B3
end
next S1 Meet.B1
next S2 Meet.B2
next S3 Meet.B3
next Meet.B1 T1
next Meet.B2 T2
next Meet.B3 T3
```

A **meet gateway** can have any number of branches. Branches are both input and output branches. Branches manages in-coming processes in queues (first in, first out). When there is a process in each queue, all first processes of each queue can move forward. They are just moved to the next locations of branches (here tasks T1, T2, T3).

Example: Rendez-Vous



Exercises

Exercise: Life-Cycle.

The life-cycle of a product is usually made of three phases: design, operation and decommissioning. The operation phase is itself decomposed into two sub-phases: production and maintenance.

Give the code that represent such a life-cycle and represent it graphically. Execute it.

Exercise: Ternary Meter

Design a Scola model to represent a meter with three wheels (like a kilometric meter) that counts in base 3.

Exercise: Tapes and Siphons

Design a Scola model that, at the one end, creates as many processes as the analyst wishes (a tape) and, at the other end, kills these processes (a siphon).

Exercise: Travel Reservation

Design a Scola model to represent a travel reservation (flight + hotel)

Exercises

Exercise: Dynamic Car Assembly

Consider a car assembly line. The process is as follows:

- A new car enters into the assembly line.
- It is then moved to a first station where is painted.
- It is then moved to the second station where the engine is assembled.
- It is then move to the third station where the wheels are assembled in two steps: first the front train, then the rear train.
- The car is then delivered (taken out the production line).

Each car must have its own series number.

There can be at most one car at each place of the assembly line, i.e. at the beginning of the line and in each station.

Hint: Use test gateway to prevent a car to be moved to a place where there is already another car. The Boolean expression (`is_block path`) can be used to check the presence of a block at the give place.

BASE TYPES & EXPRESSIONS

Base Types

Base types for ports are: Boolean (true and false), integers, reals, symbols and string.

A port is a holder for a base type.

Once declared with the directive `port` (or the instruction `new`) the value of a port can be changed arbitrarily.

This behavior may be too loose (models may be hard to debug). It is thus possible to declare a port together with its type, which forces it to take only values of this type, e.g.

```
block
  port anything false
  Boolean working false
  integer count 0
  real distance 1.0e-4
  symbol _state WORKING
  string title "Activity Report"
end
```

Warning: even if a port is declared as a (generic) symbol, its value must be always belong to a defined domains.

Domains

It is possible to restrict further the possible values of symbolic ports by declaring domains, i.e. finite sets of symbolic constants, and declaring ports with these domains. E.g.

```
domain UnitState {WORKING, DEGRADED, FAILED} end

block Unit
  UnitState _state WORKING
end
```

We shall see a specific application of domains with assertions.

Expressions (1): Boolean operators

The current version of Scola implements a number of operators applying on Boolean, numbers, symbols and string.

Boolean operators

Operator	#arguments	Description
and	≥ 1	Boolean and
or	≥ 1	Boolean or
not	1	Boolean not

Expressions (2): Inequalities

Operator	#arguments	Description
eq	2	$\text{arg1} = \text{arg2}$
df	2	$\text{arg1} \neq \text{arg2}$
lt	2	$\text{arg1} < \text{arg2}$
gt	2	$\text{arg1} > \text{arg2}$
leq	2	$\text{arg1} \leq \text{arg2}$
geq	2	$\text{arg1} \geq \text{arg2}$

Operators eq and df are polymorphic: they apply on Boolean, numbers, symbols and strings.

The other operators compare only numbers.

Expressions (3): Associative Arithmetic Operators

Operator	#arguments	Description
add	≥ 1	addition
sub	≥ 1	subtraction
mul	≥ 1	multiplication
div	≥ 1	division (for integers, integral division)
min	≥ 1	minimum value
max	≥ 1	maximum value
count	≥ 1	counts the number of (Boolean) arguments that are satisfied

Expressions (4): Other Arithmetic Operators

Operator	#arguments	Description
opp	1	$-x$
inv	1	$1/x$
abs	1	absolute value
exp	1	exponential
log	1	logarithm
sqrt	1	square root
ceil	1	smallest integer greater than the argument
floor	1	biggest integer smaller than the argument
pow	2	x^y
mod	2	modulo
integer	1	casts the argument to the closest integer
real	1	casts the argument to real (e.g. to avoid integral division)

Expressions (5): String and Conditional Operations

Operations on strings

Operator	#arguments	Description
append	≥ 1	concatenation
string	1	casts the argument to a string

Conditional expressions

Operator	#arguments	Description
if	3	if-then-else

Expressions (5): Path Operations

Path expressions

Operator	#arguments	Description
symbol	0	returns an empty path
symbol	1	casts the string argument into a path
identifier	1	returns the last identifier of a path
owner	1	returns the path minus its last identifier
append	≥ 1	concatenate the paths given as arguments
is_block	1	checks whether the argument is a path to a block
is_port	1	checks whether the argument is a path to a port
is_assertion	1	checks whether the argument is a path to an assertion
size	1	returns the number of elements of a block
element	2	returns the n-th element of a block.

Reference versus Value

Paths to elements of systems are used in two ways, as illustrated by in following assignment.

```
set Store.count (add Store.count 1)
```

The first occurrence of `Store.count` is a reference to the port `Store.count`, while the second one denotes the value of this port.

Now, we may want to give the value `Store.count` to a port `path`, and then to use the value of the port `path` as the first argument of an assignment. The following instructions do not work.

```
set path Store.count  
set path (add path 1)
```

path is assigned the value of `Store.count` and not to the path `Store.count`.

Even if the value of `path` is `Store.count`, it is `path` which is assigned and not `Store.count`. Moreover it is assigned to the value of `path` (plus one) and not to the value of `Store.count` (plus one).

Quote & Eval (1)

To prevent a port to be evaluated, it is possible to **quote** it.

```
set Store.count 1  
set path 'Store.count  
set path (quote Store.count)
```

← equivalent

The value of `path` is the symbol `Store.count` and not the value of the port `Store.count`.

Symmetrically, to evaluate a symbol, i.e. to take the value of the port reachable with this path, it is possible to **eval** it.

```
set (eval path) (add (eval path) 1)
```

Store.count

The above assignment increments by 1 the value of `Store.count`.

Quote & Eval (2)

Functions quote and eval apply recursively to arguments of other functions:

```
(quote (append a b)) → (append (quote a) (quote b))  
(eval (add a b)) → (add (eval a) (eval b))
```

Functions quote and eval cancel one another when applied to references:

```
(quote (eval a)) → a  
(eval (quote a)) → a
```

Instructions such as assignment quote implicitly their arguments referring to a port before evaluating them:

```
set (eval path) (add (eval path) 1)  
    ①           ②
```

① (eval (quote (eval path))) → Store.count

② (eval (eval path)) → 33

Exercises

Exercise: Largest port

A block `Store` contains an arbitrary number of integer ports. Design a scenario to get the name of the port with the largest value.

Hint: use instruction `if condition then instruction` and instruction block `begin instructions end`

INSTRUCTIONS

Instructions

Instructions are used in tasks and in assertions (see next section). Instructions can be divided into two groups:

- Assignment, conditional instructions, blocks of instructions that can be used both in tasks and assertions.
- Instructions to create, destroy and move components that can be used only in tasks. The special instruction `fail` enters also in this category.

The semantics of instructions of the first category is straightforward.

Assignment:

```
set path-expression expression
```

Conditional instruction

```
if Boolean-expression then instruction [else instruction]  
# the else part is optional
```

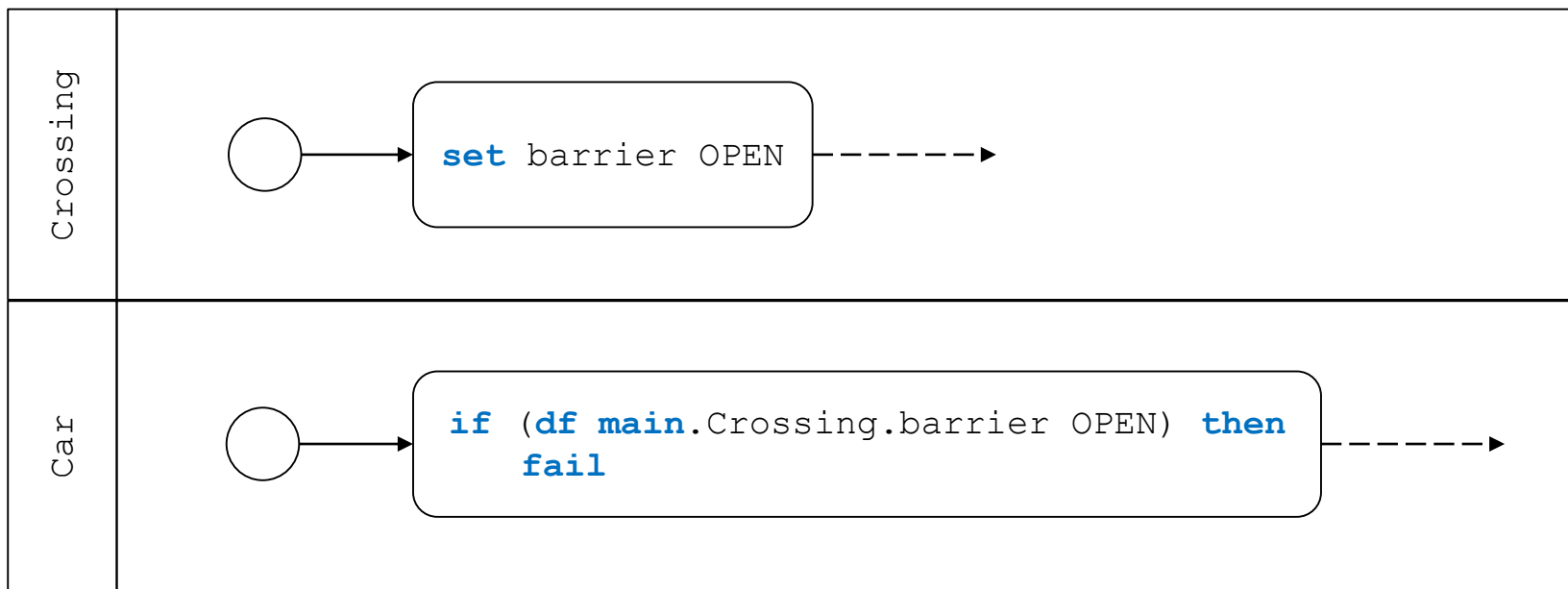
Blocks of instructions

```
begin instructions* end
```

Fail Instruction

The fail instruction always fails. It is used in combination with the conditional instruction to postpone the execution of a task until a certain condition is verified.

Consider for instance a car waiting at a railway crossing. The driver waits for the barrier to open before to go. In its simplest form, it could be as follows.



See exercise Dynamic Car Assembly Revisited for an illustration

Instructions to Create and to Destroy Components

- Instructions to create blocks and ports are as follows.

new port *path-expression expression*

new block *path-expression*

Required: there must be no component with the same name at the same place.

- The instruction to delete a block, a port or an assertion is as follows.

delete *path-expression*

Required: the referred component must exist.

- The instruction to clone a component is as follows (this instruction should not to confuse with the `clones` directive)

clone *path-expression path-expression*

Required: the cloned component (first argument) must exist and there must be no component with the same name at the same place (second argument).

See example `ActivityReport.scola` for an illustration.

Instructions to Move Components

- The instruction to move a block or a port to another location is as follows.

move *path-expression path-expression*

Required: the moved component (first argument) must exist and there must be no component with the same name at the same place (second argument).

See example ActivityReport.scola for an illustration.

- Instructions to move a block or a port in an asynchronous way are as follows.

send *path-expression path-expression*

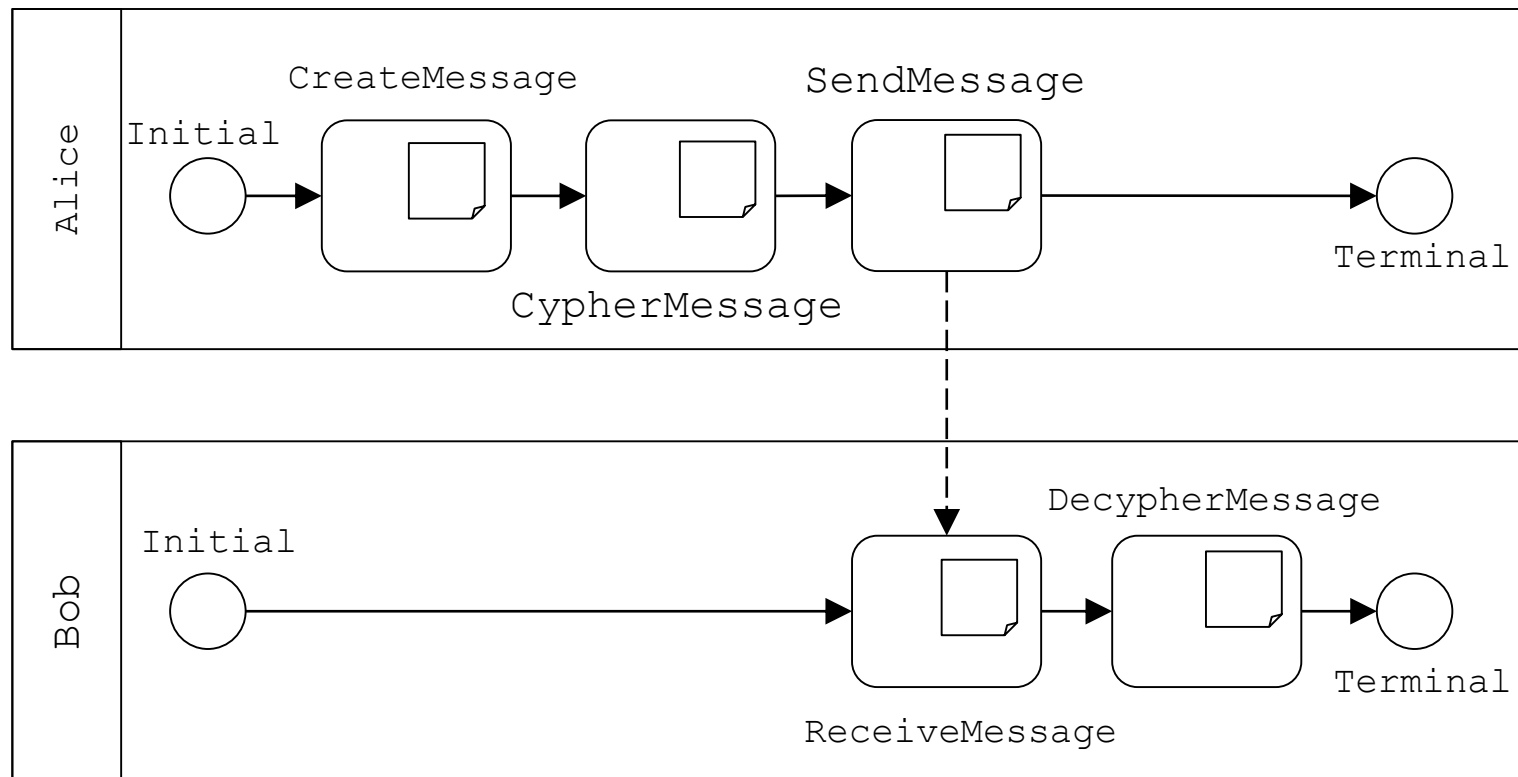
receive *path-expression path-expression identifier*

The first argument of the `send` instruction is the path to the sent component. The second argument is the path to the block in which it is sent.

The first argument of `receive` instruction is the path to the send component. The second argument is the path to the block that sends the component. The third argument is the identifier of the component once received.

Required: the sent component, the sending block and the receiving block must exist and there must be no component with the same name in the receiving block.

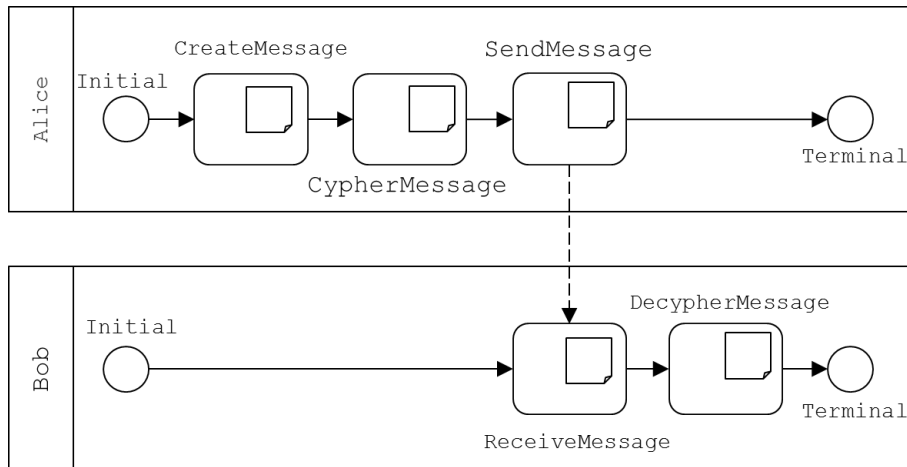
Example: Cryptography (1)



The two processes (one for Alice, one for Bob) are running in parallel.

Sending and reception of the message are asynchronous: once the `send` instruction executed, the sent component is removed from the sending block. It is inserted in the receiving block only once the `receive` instruction has been executed.. Reception is blocking.

Example: Cryptography (2)



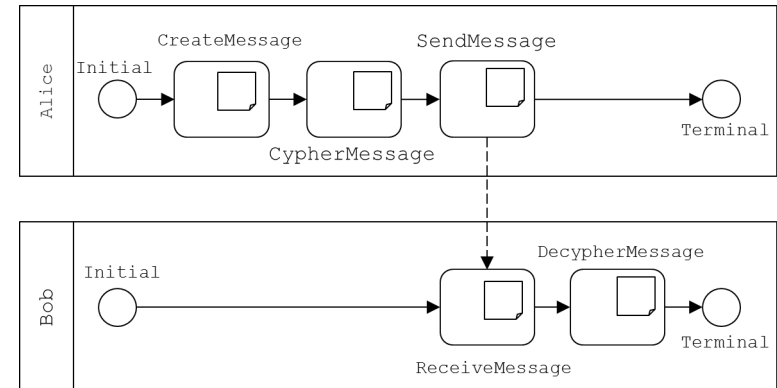
```
block Alice end  
block Bob end
```

```
scenario CypheredMessage  
  scenario AlicePool as Alice  
    ...  
  end  
  scenario BobPool as Bob  
    ...  
  end  
end
```

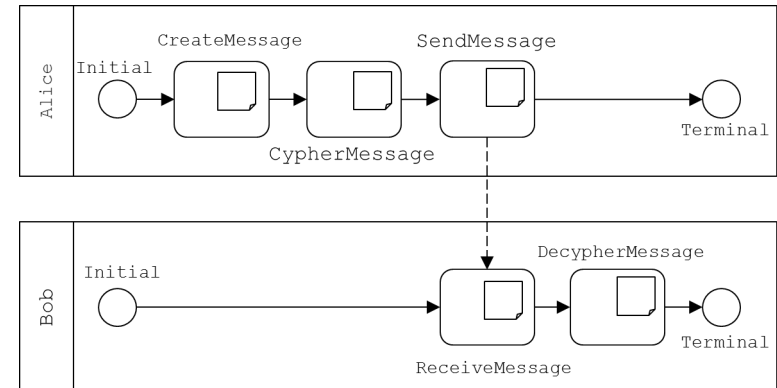
Example: Cryptography (3)

```

scenario AlicePool as Alice
  state Initial
  task CreateMessage
    new block message
    new port message.status CREATED
  end
  task CypherMessage
    set message.status CYPHERED
  end
  task SendMessage
    send message main.Bob
  end
  state Terminal
  next Initial CreateMessage
  next CreateMessage CypherMessage
  next CypherMessage SendMessage
  next SendMessage Terminal
end
  
```



Example: Cryptography (4)

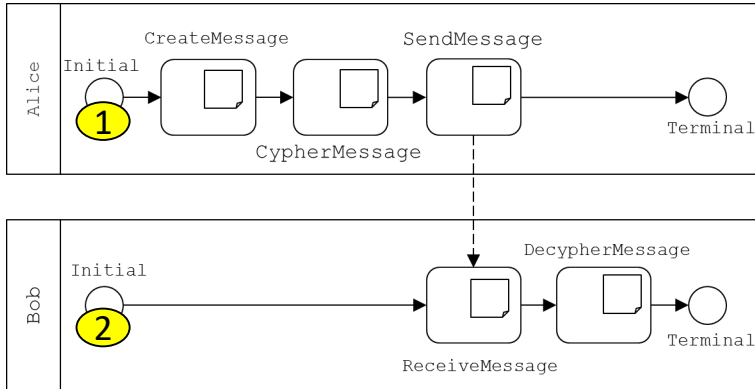


```

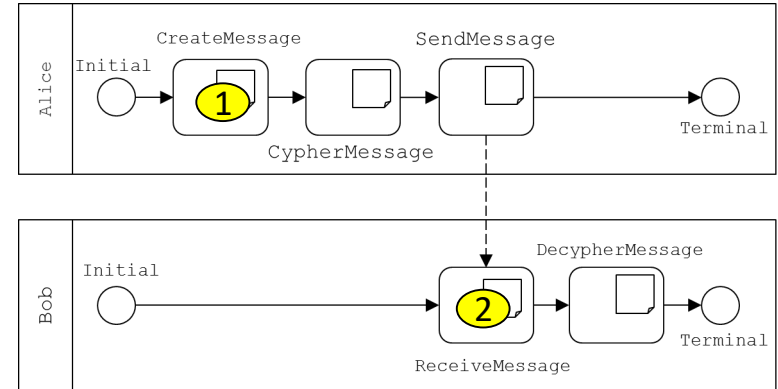
scenario BobPool as Bob
  state Initial
  task ReceiveMessage
    receive message owner.Alice receivedMessage
  end
  task DecypherMessage
    set receivedMessage.status DECYPHERED
  end
  state Terminal
  next Initial ReceiveMessage
  next ReceiveMessage DecypherMessage
  next DecypherMessage Terminal
end
  
```

Example: Cryptography (5)

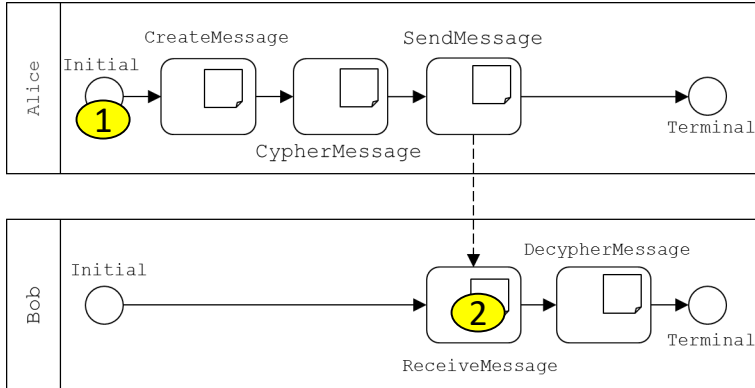
1



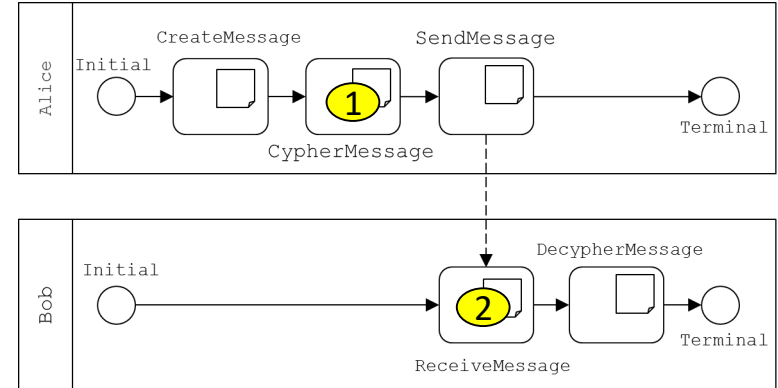
3



2



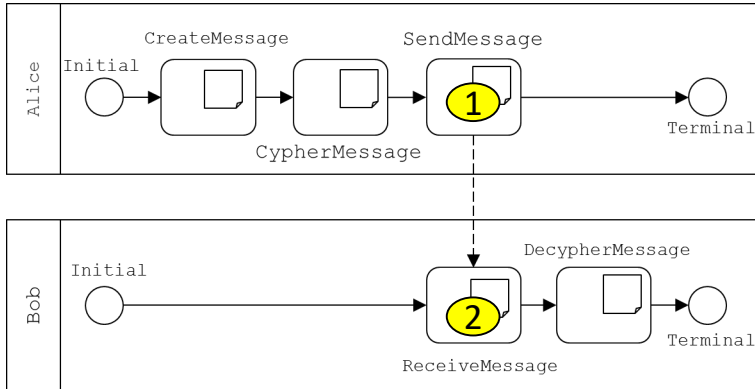
4



The process 2 is blocked until the reception of the message

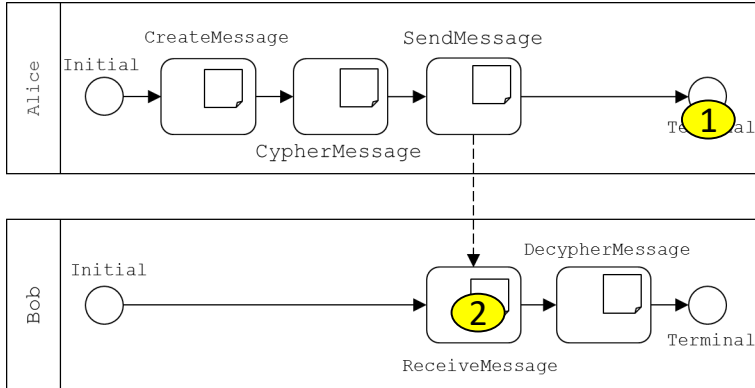
Example: Cryptography (6)

5



block Alice
block message
port status CYPHERED
block Bob

6

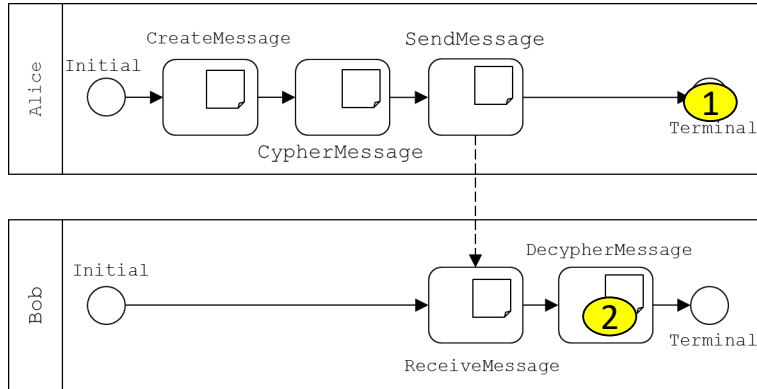


block Alice
block Bob

The message is sent but not yet received

Example: Cryptography (6)

5



block Alice

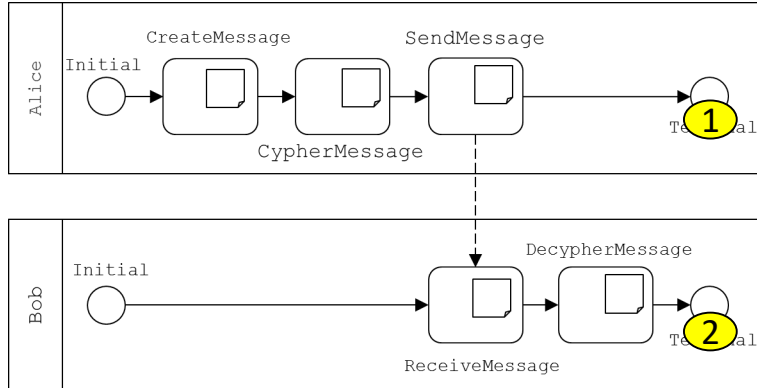
block Bob

block receivedMessage

port status CYPHERED

The name of the block has been changed

6



block Alice

block Bob

block receivedMessage

port status DECYPHERED

Exercises

Exercise: Dynamic Car Assembly (revisited)

Design a model that use fail instructions rather than test gateways to solve the dynamic car assembly exercise.

Exercise: Master Thesis

Bob is doing his master project under the supervision of Alice. He has to do some research and in parallel to write his master thesis. This requires some iterations with Alice until she gives eventually her approval.

Design a model to represent this process. First, just using ports, without any component creation. Second, with component creation and moving. Third with component creation, sending and reception.

Exercises

Exercise: Queues

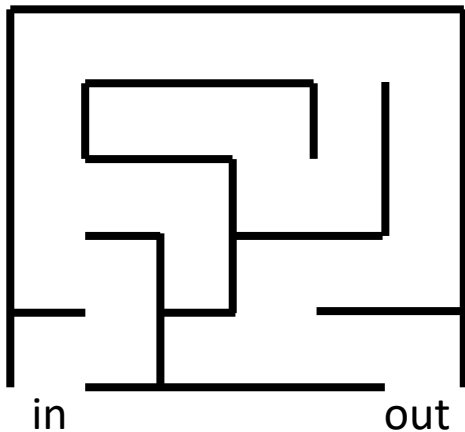
In an shop, clients must choose one of two queues at the cashier. They are served in the order of arrival in the queue they choose.

Design a model for such a system and simulate it.

Hint: use three processes, one to create new clients and one for each queue.

Exercise: Maze

Design a Scola model to get out of the following maze.



Hint: recall Tom Thumb.

Exercises

Exercise: Eratosthenes

Design a model to calculate prime numbers lower than 100 using Eratosthenes' Sieve.

The idea is to have two nested loops: the outer one to generate candidate numbers (from 3 to 100 in order) and the inner one to test candidates. The test consist in comparing (via a modulo) the candidate with all prime numbers found so far.

Hint: Prime numbers are store as integer ports $p1=2$, $p2=3$, $p3=5...$ into a block Primes.

Exercise: Ferry

A ferry carries trucks from the left bank to the right bank of a river. It goes forth and back as long as there are trucks to carry. It can contain only one truck at a time.

Design a Scola model to represent this ferry.

ASSERTIONS

Assertions

So far, descriptions of systems we have seen consisted in hierarchies of blocks, and ports.

It is however often suitable/necessary to describe not only the structural decomposition of the system, but also connections existing between its components. These connections makes the information circulate through the components of the system. Information is to be taken in a broad sense, including flow of matters, energy, information...

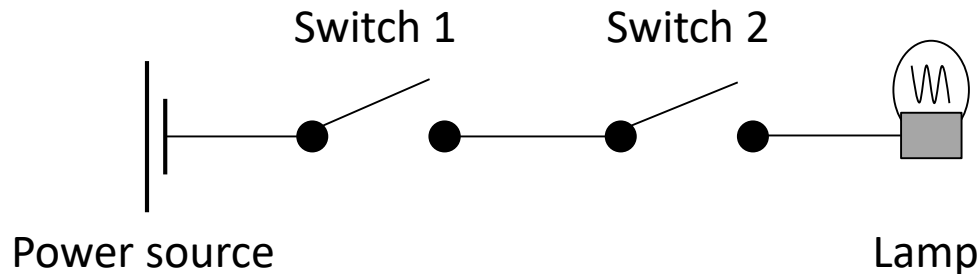
Scola provides the concept of assertion, stemmed from the AltaRica modeling language, to describe connections and their semantics.

An **assertion** is an instruction, or a group of instructions, that updates the values of ports after the execution of a task.

As assertions may be spread all over the system, the result of the update should not depend on the order of the execution of instructions of the assertion. This is the reason why, a **fixpoint mechanism** is used for assertions: the assertion is re-executed until the values of ports stabilized. It is up to the analyst to ensure that this stabilization process terminates.

Electric Circuit (1)

Consider a small electric circuit consisting of a power source two switches and a lamp in series. All components are assumed to be perfectly reliable.



Modeling this system is easy: the system is decomposed into four subsystems, one per component. The scenario is made of two lanes, one of each switch. The other components are actually passive.

However, we would like to determine automatically when the lamp is powered, depending on the states of the switches. This is achieved by means of assertions.

Electric Circuit (2)

```
block ElectricCircuit
  block PowerSource
    Boolean outPower true
  end
  block Switch1
    Boolean _closed true
    Boolean inPower false
    Boolean outPower false
    ...
  end
  block Switch2
    Boolean _closed true
    Boolean inPower false
    Boolean outPower false
    ...
  end
  block Lamp
    Boolean on true
    Boolean inPower false
    ...
  end
  ...
end
```

```
scenario Light as ElectricCircuit
  scenario Switch1Lane as Switch1
    state Initial
    task Switch
      set _closed (not _closed)
    end
    next Initial Switch
    next Switch Switch
  end
  scenario Switch2Lane as Switch2
    state Initial
    task Switch
      set _closed (not _closed)
    end
    next Initial Switch
    next Switch Switch
  end
end
```

Assertions must link all ports together

Electric Circuit (3)

```
block Switch1
  Boolean _closed false
  Boolean inPower false
  Boolean outPower false
  assertion Powering
    set outPower (if _closed inPower false)
  end
end

...
block Lamp
  Boolean on false
  Boolean inPower false
  assertion Powering
    set on inPower
  end
end

...
assertion Powering
  set Switch1.inPower PowerSource.outPower
  set Switch2.inPower Switch1.outPower
  set Lamp.inPower Switch2.outPower
end
```

Assertions have a name.

They consists of a block of instructions.

They can be associated with any block.

Electric Circuit (4)

At system level, we have the following assertions.

```
set Switch1.outPower (if Switch1._closed Switch1.inPower false)
set Switch2.outPower (if Switch2._closed Switch2.inPower false)
set Lamp.on Lamp.inPower
set Switch1.inPower PowerSource.outPower
set Switch2.inPower Switch1.outPower
set Lamp.inPower Switch2.outPower
```

step	Power Source	Switch 1			Switch 2			Lamp	
	outPower	_closed	inPower	outPower	_closed	inPower	outPower	on	inPower
0	true	true	false	false	true	false	false	false	false
1	true	true	true	false	true	false	false	false	false
2	true	true	true	true	true	true	false	false	false
3	true	true	true	true	true	true	true	false	true
4	true	true	true	true	true	true	true	true	true
5	true	true	true	true	true	true	true	true	true

Electric Circuit (5)

At system level, we have the following assertions.

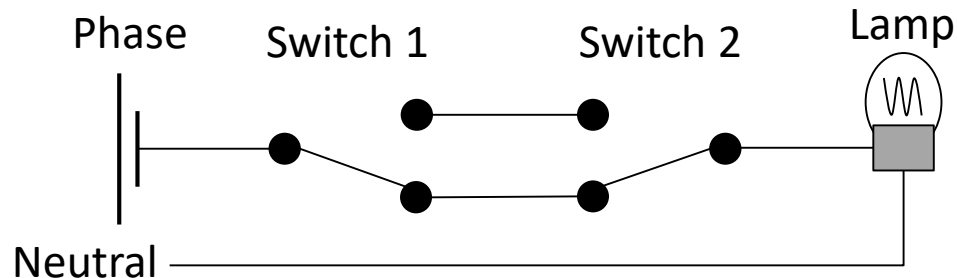
```
set Switch1.outPower (if Switch1._closed Switch1.inPower false)
set Switch2.outPower (if Switch2._closed Switch2.inPower false)
set Lamp.on Lamp.inPower
set Switch1.inPower PowerSource.outPower
set Switch2.inPower Switch1.outPower
set Lamp.inPower Switch2.outPower
```

step	Power Source	Switch 1		Switch 2				Lamp	
	outPower	_closed	inPower	outPower	_closed	inPower	outPower	on	inPower
0	true	false	true	true	true	true	true	true	true
1	true	false	true	false	true	false	true	true	true
2	true	false	true	false	true	false	false	false	true
3	true	false	true	false	true	false	false	false	false
4	true	false	true	false	true	false	false	false	false

Exercises

Exercise: Two-Way Switch

Modify the code proposed in this section so to model a two-way switch.



Exercise: Wages

Alice, Bob and Carol are salespersons. Their monthly wages are calculated as follows. Fixed salary + 4% of the growth revenue they generate + 800€ if the sum of the two preceding numbers is below 9000€ and 400€ if it above. Design a model to calculate their wages.

Name	Gr. Rev.	Salary	Var. Part	Bonus	Total
Alice	47 500	8 000	1 900	400	10 300
Bob	38 900	6 700	1556	800	9 056
Carol	51 600	9 000	2 064	400	11 464

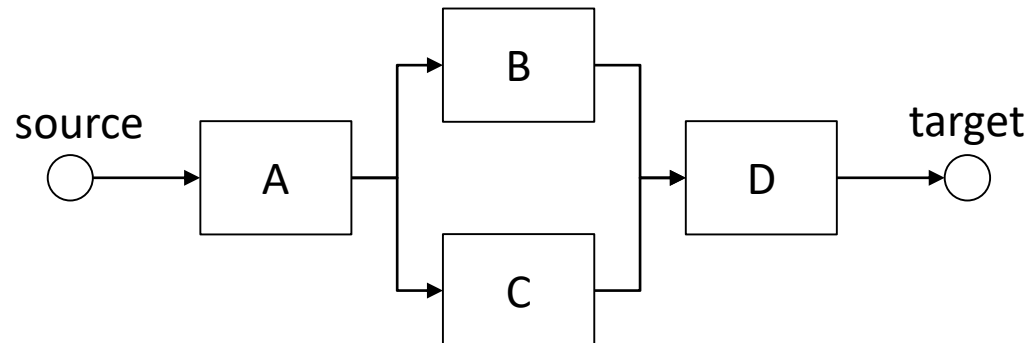
Exercises

Exercise: 2-out-of-3 system

A 2-out-of-3 system is a system that works if at least two out of its 3 components are working. Design a model for such a system and simulate it.

Exercise: Bridge

Components A, B, C and D of the following reliability block diagram may fail and be repaired. The system described by the diagram is working if there is a working path from the source node to the target node. Design a model for such a system and simulate it.



STRUCTURING CONSTRUCTS

Object-Oriented versus Prototype-Oriented Modeling

It is often the case, when modeling a system (whether with Scola or with another language), that the system under study involves several identical or at least similar components, see e.g. the Bridge exercise of the previous section.

So far, when such situation occurred we just duplicated the code, possibly for both component and scenario descriptions. This is both tedious and error prone.

All advanced programming and modeling languages provide thus constructs to describe identical components only once, then to indicate that identical components are just copies of the reference one.

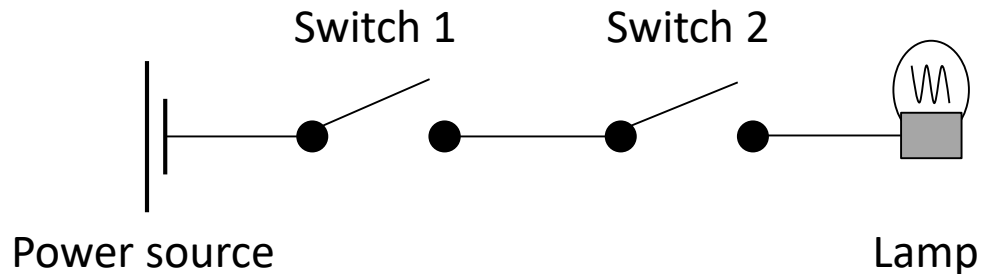
There are two paradigms to implement this mechanism:

- The **prototype-oriented paradigm**, in which it is possible to **clone** an already declared component.
- The **object-oriented paradigm**, in which reference components are declared separately as **classes**. It is then possible to introduce in the model **instances**, i.e. copies, of these classes. Classes are thus on-the-shelf, reusable modeling components.

Scola, following in that S2ML, implements both paradigm.

Use Case

As an illustration, we shall consider again the small electric circuit of the previous section.



In this example, switches 1 and 2 are identical.

In the previous section, we simply duplicated the code for both the system and the scenario description.

Cloning (1)

The `clones` directive makes it possible to duplicate a block or a scenario.

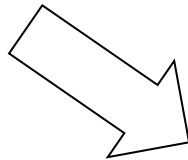
```
block ElectricCircuit
  block PowerSource
    ...
  end
  block Switch1
    ...
  end
  clones Switch1 as Swicth2
  end
  block Lamp
    ...
  end
end

scenario Light as ElectricCircuit
  scenario Switch1Lane as Switch1
    ...
  end
  clones Switch1Lane as Switch2Lane as Switch2
  end
end
```

In our example, clones and cloned components are strictly identical. It is however possible to add more components to the clone or to modify initial values of ports.

Cloning (2)

```
block Switch1
  Boolean _closed true
  Boolean inPower false
  Boolean outPower false
  assertion Powering
    set outPower (if _closed inPower false)
end
end
clones Switch1 as Switch2
  set _closed false
  integer number 1001
end
```



```
block Switch2
  Boolean _closed false
  Boolean inPower false
  Boolean outPower false
  assertion Powering
    set outPower (if _closed inPower false)
  end
  integer number 1001
end
```

Classes & Instances (1)

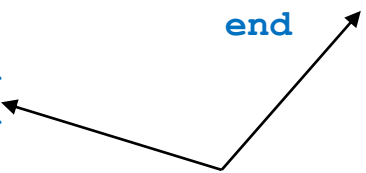
Another way to achieve the same goal consists in defining classes, i.e. on-the-shelf reusable modeling components, then to instantiate these classes into the model. Classes are thus independent from the model.

```
class Switch(block)
...
end

block ElectricCircuit
  block PowerSource
    ...
  end
  Switch Switch1 end
  Switch Switch2 end
  block Lamp
    ...
  end
...
end

scenario SwitchLane(scenario)
...
end

scenario Light as ElectricCircuit
  SwitchLane Switch1Lane as Switch1 end
  SwitchLane Switch2Lane as Switch2 end
end
```



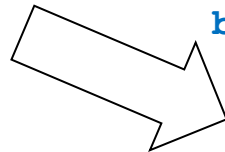
The block `ElectricCircuit` declares two instances of the class `Switch`. The scenario `Light` declares two instances of the class `SwitchLane`.

As for cloning, it is possible to change the initial values of ports and to add new elements.

Classes & Instances (2)

```
class Switch(block)
  Boolean _closed true
  Boolean inPower false
  Boolean outPower false
  assertion Powering
    set outPower (if _closed inPower false)
end
end
```

```
block ElectricCircuit
...
Switch Switch2
  set _closed false
  integer number 1001
end
...
end
```



```
block Switch2
  Boolean _closed false
  Boolean inPower false
  Boolean outPower false
  assertion Powering
    set outPower (if _closed inPower false)
  end
  integer number 1001
end
```

Inheritance (1)

In the previous example, the **class** `Switch` **inherits** from the base class `block`, while the class `SwitchLane` inherits from the base class `scenario`. We say also that `Switch` **derives** from `block` and that `SwitchLane` derives from `scenario`. In Scola, a class may derive from another class. In any case, it derives eventually either from the base class `block` or from the base class `scenario`.

If a class B derives from a class A, all elements of A are copied in B when B is instantiated.

```
class Connection(block)
  Boolean inPower false
  Boolean outPower false
end

class Switch(Connection)
  Boolean _closed true
  assertion Powering
    set outPower (if _closed inPower false)
  end
end
```


Inheritance (2)

More generally, it is possible in Scola to make any block inherit from another block and any scenario inherit from another scenario, with the following constraints:

- A prototype of block (resp. scenario) can inherit from another prototype of block (resp. scenario) or from a class deriving from block (resp. scenario).
- A class deriving from block (resp. scenario) can inherit from another class deriving from block (resp. scenario).

```
class Connection(block)
  Boolean inPower false
  Boolean outPower false
end
```

```
block Switch1
  inherits Connection
  Boolean _closed true
  assertion Powering
    set outPower (if _closed inPower false)
  end
end
```

Exercises

Exercise: Electric Circuit

Design the complete model of the electric circuit presented in this section. First without cloning nor classes, then with cloning and finally with classes.

Exercise: Bridge

Same question with the Bridge exercise of the previous section.

Exercise: Collaborative Report

Alice and Bob write a report. Alice makes version 0, then each of them read the report in turn. After reading they can decide either to finalize it, which stops the writing process, or to improve it and to pass it to their colleague.

Design a object-oriented Scola model for this scenario.

REFERENCES

References

- (Batteux & al. 2015) Michel Batteux, Tatiana Prosvirnova and Antoine Rauzy. System Structure Modeling Language (S2ML). AltaRica Association. 2015. archive hal-01234903, version 1.
- (Issad & al. 2018) Mélissa Issad, Leïla Kloul and Antoine Rauzy. Scenario-Oriented Reverse Engineering of Complex Railway System Specifications. Journal of Systems Engineering. Wiley Online Library. 21:2. pp. 91–104. March, 2018. doi:10.1002/sys.21413.
- (Milner 1999) Robin Milner. Communicating and Mobile Systems: The pi-calculus. Cambridge University Press. Cambridge, CB2 8BS, United Kingdom. ISBN 978-0521658690. 1999.
- (White & Miers 2008) Stephen White and Derek Miers. BPMN Modeling and Reference Guide: Understanding and Using BPMN. Future Strategies Inc.. Lighthouse Point, FL, USA. ISBN 978-0977752720. 2008.

GRAMMAR

Models

```
Model ::=
    Declaration*
```

```
Declaration ::=
    DomainDeclaration
    | BlockDeclaration
    | ScenarioDeclaration
    | ClassDeclaration
```

```
DomainDeclaration ::=
    domain Identifier "{" Identifier ("," Identifier)* "}" end
```

Blocks & Ports

BlockDeclaration ::=

block Identifier BlockField* **end**

BlockField ::=

PortDeclaration | BlockDeclaration | AssertionDeclaration
| InheritsDirective | ClonesBlockDirective | BlockClassInstance
| SetInstruction

PortDeclaration ::=

port Identifier Expression

SetInstruction ::=

set Expression Expression *# set path-to-port value*

Assertions

AssertionDeclaration ::=

assertion Identifier AssertionInstruction* **end**

AssertionInstruction ::=

fail

| SetInstruction

| **if** Expression **then** AssertionInstruction (**else** AssertionInstruction)?

| **begin** AssertionInstruction* **end**

Scenarios, Connections, Tasks & States

```
ScenarioDeclaration ::=
    scenario Identifier (as Path)? ScenarioField* end
```

```
ScenarioField ::=
    StateDeclaration | TaskDeclaration | GatewayDeclaration
    | ScenarioDeclaration | NextDirective
    | ClonesScenarioDirective | InheritsDirective | ScenarioClassInstance
```

```
StateDeclaration ::=
    state Identifier
```

```
TaskDeclaration ::=
    task Identifier Instruction* end
```

```
NextDirective ::=
    next Path Path
```

Classes & Instances

```
ClassDeclaration ::=
    BlockClassDeclaration | ScenarioClassDeclaration
```

```
BlockClassDeclaration ::
    class Identifier "(" block | Identifier ")" BlockField* end
```

```
BlockClassInstance ::=
    Identifier Identifier BlockField* end
```

```
ScenarioClassDeclaration ::
    class Identifier "(" scenario | Identifier ")"
    (as Path)? ScenarioField* end
```

```
ScenarioClassInstance ::=
    Identifier Identifier (as Path)? ScenarioField* end
```

Clones & Inherits Directives

ClonesBlockDirective ::=

clones Path BlockField* **end**

ClonesScenarioDirective ::=

clones Path (**as** Path)? ScenarioField* **end**

InheritsDirective ::=

inherits Path

Gateways (1)

```
GatewayDeclaration ::=
    TestDeclaration | ChoiceDeclaration
    | ForkDeclaration | JoinDeclaration
    | SplitDeclaration | MergeDeclaration
    | MeetDeclaration
```

```
TestDeclaration ::=
    test Identifier CaseDeclaration+ end
```

```
CaseDeclaration ::=
    case Identifier BooleanExpression
```

Gateways (2)

```
ChoiceDeclaration ::=
    choice Identifier BranchDeclaration+ end
ForkDeclaration ::=
    fork Identifier BranchDeclaration+ end
JoinDeclaration ::=
    join Identifier BranchDeclaration+ end
SplitDeclaration ::=
    split Identifier BranchDeclaration+ end
MergeDeclaration ::=
    merge Identifier BranchDeclaration+ end
MeetDeclaration ::=
    meet Identifier BranchDeclaration+ end

BranchDeclaration ::=
    branch Identifier
```

Instructions

Instruction ::=

```
    set Expression Expression
    # set path-to-port value
|   new port Expression Expression
    # new port path-to-port initial-value
|   new block Expression
    # new block path-to-block
|   delete Expression
    # delete path-to-item
|   move Expression Expression
    # delete path-to-source-item path-to-target-item
|   send Expression Expression
    # delete path-to-item path-to-receiver
|   receive Expression Expression Expression
    # receive path-to-item path-to-sender path-to-target-item
|   clone Expression Expression
    # clone path-to-source-item path-to-target-item
|   if Expression then Instruction (else Instruction)?
|   begin instruction* end
```

Expressions & Boolean Expressions

```
Expression ::=
    none
    | Path
    | BooleanExpression | ArithmeticExpression | StringExpression
    | ConditionalExpression | PathExpression
```

```
Path ::= Identifier ( "." Identifier ) *
```

```
Identifier ::= [a-zA-Z_][a-zA-Z0-9_]*
```

```
BooleanExpression ::=
    false | true
    | "(" BooleanOperator Expression+ ")"
    | "(" Comparator Expression Expression ")"
```

```
BooleanOperator ::= and | or | not
```

```
Comparator ::= eq | df | lt | gt | leq | geq
```

Arithmetic & String Expressions

ArithmeticExpression ::=

```
    Number
  |   "(" AssociativeArithmeticOperator Expression+ ")"
  |   "(" UnaryArithmeticOperator Expression ")"
  |   "(" BinaryArithmeticOperator Expression Expression ")"
  |   "(" integer Expression ")"
  |   "(" real Expression ")"
```

Number ::= [-+]?[0-9]+(.[0-9]*)([eE][-+]?[0-9]+)?

AssociativeArithmeticOperator ::= **add** | **sub** | **mul** | **div** | **min** | **max** | **count**

UnaryArithmeticOperator ::= **opp** | **inv** | **abs** | **exp** | **log** | **sqrt** | **ceil** | **floor**

BinaryArithmeticOperator ::= **pow** | **mod**

StringExpression ::=

```
    String
  |   "(" StringOperator Expression+ ")"
  |   "(" string Expression ")"
```

String ::= [""]([[^]""]|\\")*[""] | ['']([[^]''|\\'])*['']

StringOperator ::= **append**

Conditional & Path Expressions

ConditionalExpression ::=

```
"(" if BooleanExpression Expression Expression ")"
```

PathExpression ::=

```
"(" is_port PathExpression ")"  
|  "(" is_block PathExpression ")"  
|  "(" is_assertion PathExpression ")"  
|  "(" size PathExpression ")"  
|  "(" element PathExpression ArithmeticExpression ")"  
|  "(" append PathExpression+ ")"  
|  "(" symbol StringExpression? ")"
```

SEMANTICS

Operational Semantics

The operational semantics of a Scola model is defined as the set of possible executions of one or more processes making the system evolve.

An **execution** is a finite sequence of states.

Each **state** is characterized by:

- A system
- A set of processes.

A **system** is a hierarchy of blocks and ports. Moreover, each block maintains a reception set. This **reception set** contains systems that have been sent to the block, but not yet received by the block.

A **process** is characterized by its number and its location.

The **location** of a process is either a state, a task or a gateway.

Moreover:

- Each process may have a **parent process** and a set of **child processes**.
- Join gateways maintain a **set of in-coming processes**.
- Merge gateways maintain a set of **queues of processes**, one queue per in-coming location.

SOLUTIONS TO THE EXERCISES

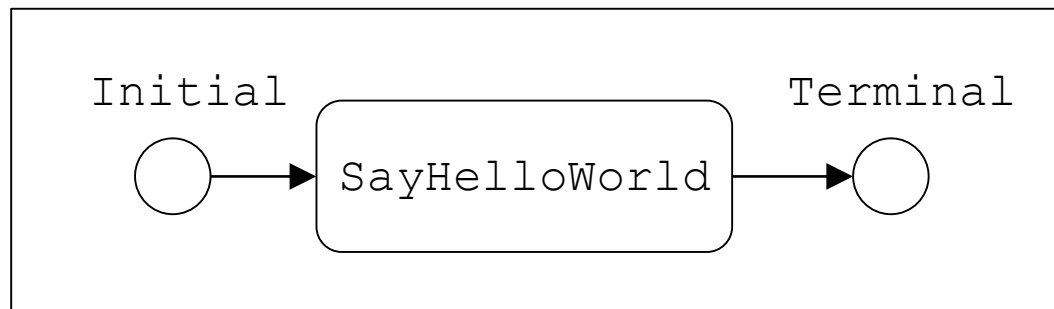
Solutions to the Exercises

The solution (the code) to all exercises proposed in this tutorial is provided with the distribution of Scola tools in the folder "examples".

This folder contains also the code for all examples presented in the tutorial.

This section contains only the solutions to some selected examples.

Hello World! (1)

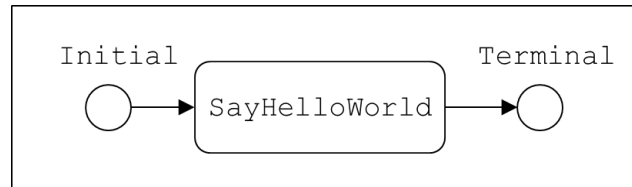


The system is made of a single (implicit) block.

The scenario consists in starting from an initial state, going through a task SayHelloWorld and ending up into a terminal state.

Hello World! (2)

```
scenario HelloWorld
  state Initial
  task SayHelloWorld end
  state Terminal
  next Initial SayHelloWorld
  next SayHelloWorld Terminal
end
```

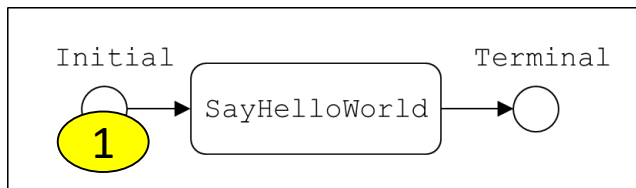


- Tasks are container for instructions. This is the reason why their declarations end with the keyword "end".
- Next instructions represent arrows. They are used to chain states, tasks and gateways.
- The order of declaration of elements is irrelevant:

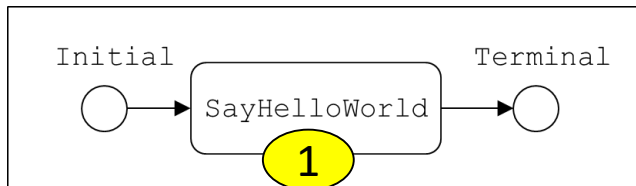
```
scenario HelloWorld
  state Initial
  next Initial SayHelloWorld
  task SayHelloWorld end
  next SayHelloWorld Terminal
  state Terminal
end
```

Hello World! (3)

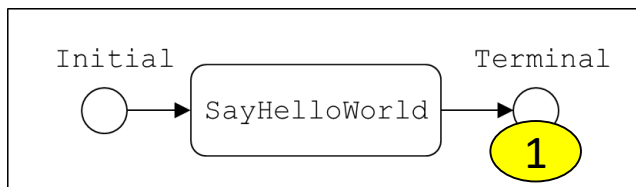
Scola models describe the evolution of processes. A **process** has a number, possibly a **parent process** and a number of **child processes**. Processes can be dynamically created by the execution of a model. At any step of the execution, a process is either inactive, or it is located on a state, a task or a gateway.



Step 1: a process number 1 is created on the initial state.

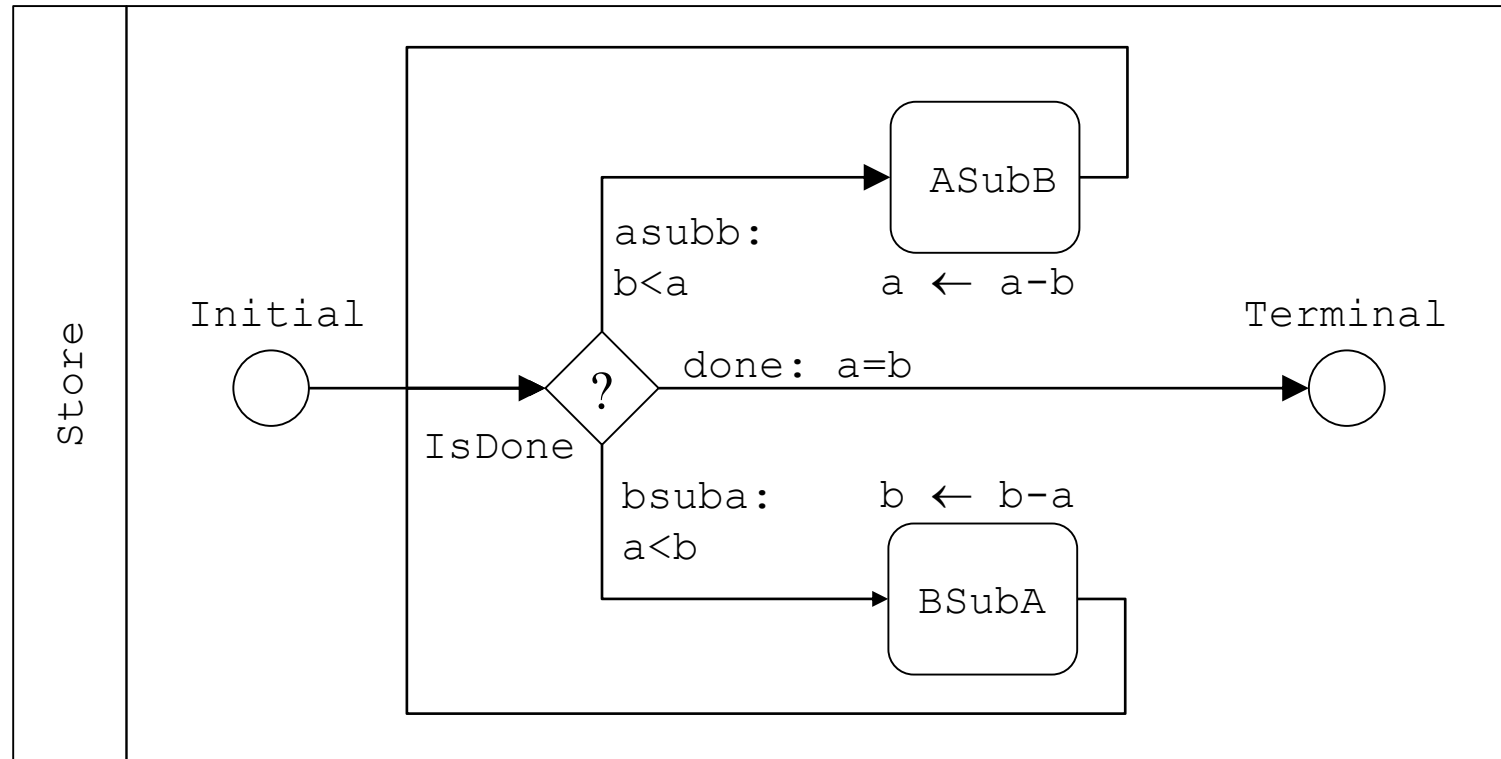


Step 2: the process moves to the task SayHelloWorld and performs this task.



Step 3: the process moves to the terminal state. Its execution is finished.

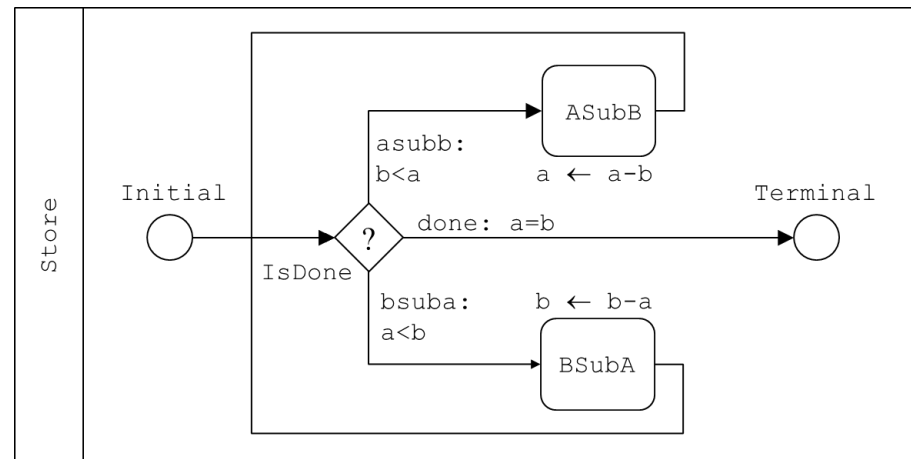
Greatest Common Divisor (1)



Greatest Common Divisor (2)

```

scenario GCD as Store
  state Initial
  test IsDone
    case done (eq a b)
    case asubb (lt b a)
    case bsuba (lt a b)
  end
  task ASubB
    set a (sub a b)
  end
  task BSubA
    set b (sub b a)
  end
  state Terminal
  next Initial IsDone
  next Reset Increment
  next IsDone.done Terminal
  next IsDone.asubb ASubB
  next IsDone.bsuba BSubA
  next ASubB IsDone
  next BSubA IsDone
end
  
```



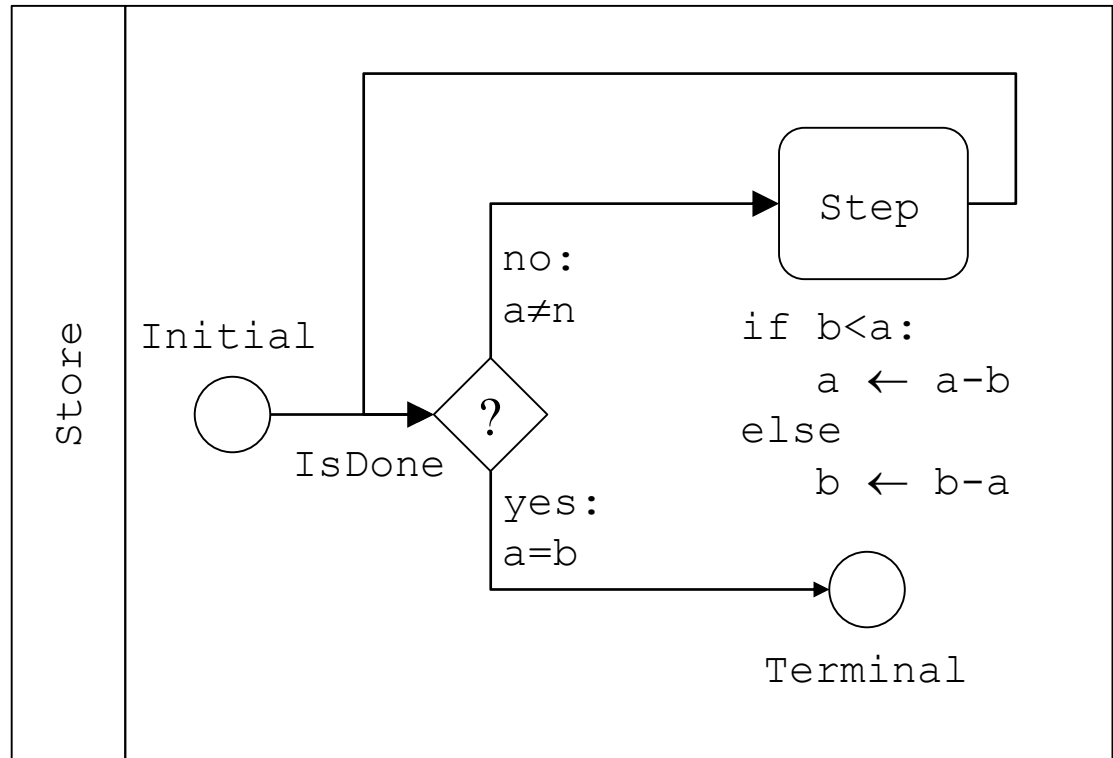
Greatest Common Divisor (3)

```

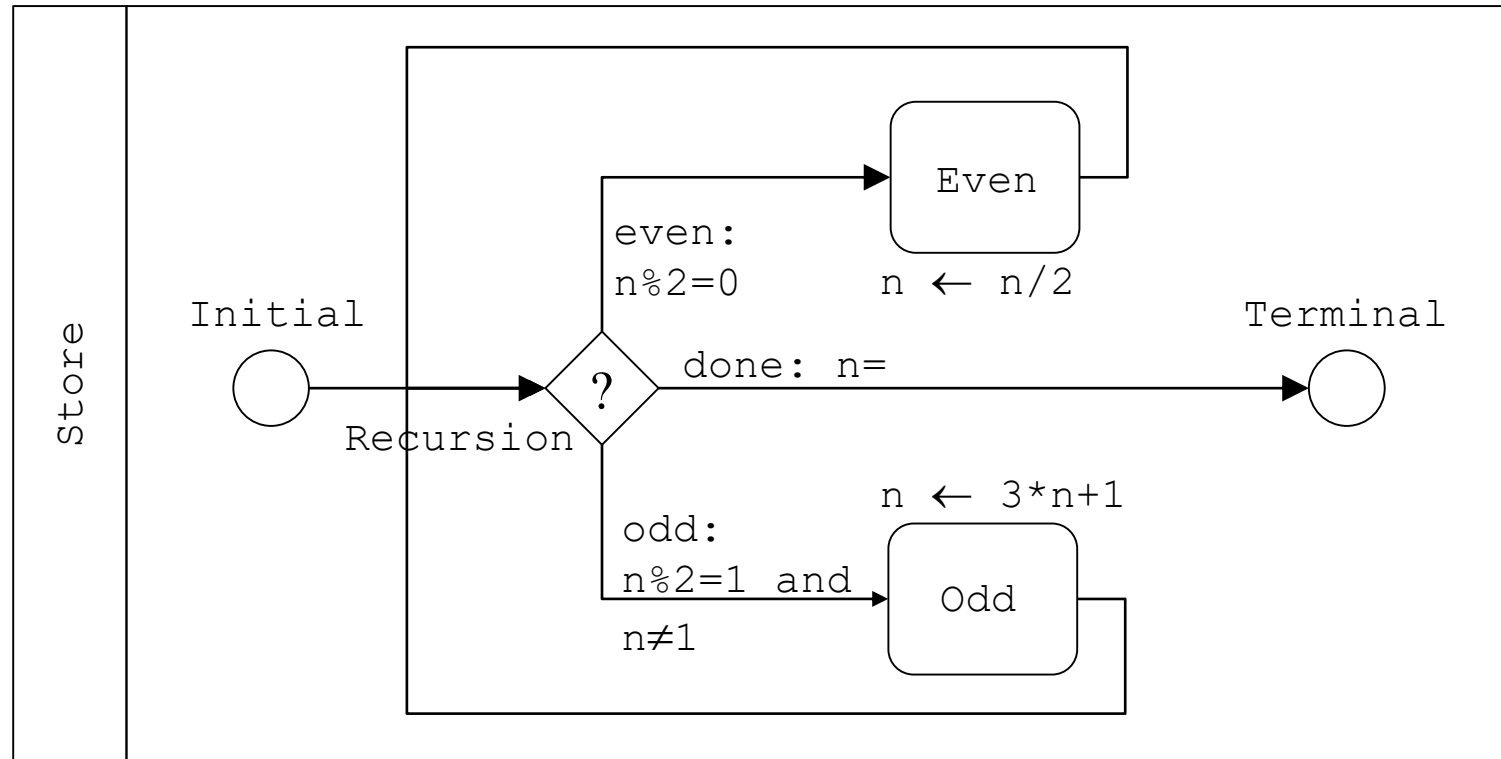
block Store
  integer a 96
  integer b 81
end

scenario GCD as Store
  state Initial
  test IsDone
    case yes (eq a b)
    case no (df a b)
  end
  task Step
    if (lt a b) then
      set b (sub b a)
    else
      set a (sub a b)
    end
  end
  state Terminal
  next Initial IsDone
  next IsDone.yes Terminal
  next IsDone.no Step
  next Step IsDone
end

```

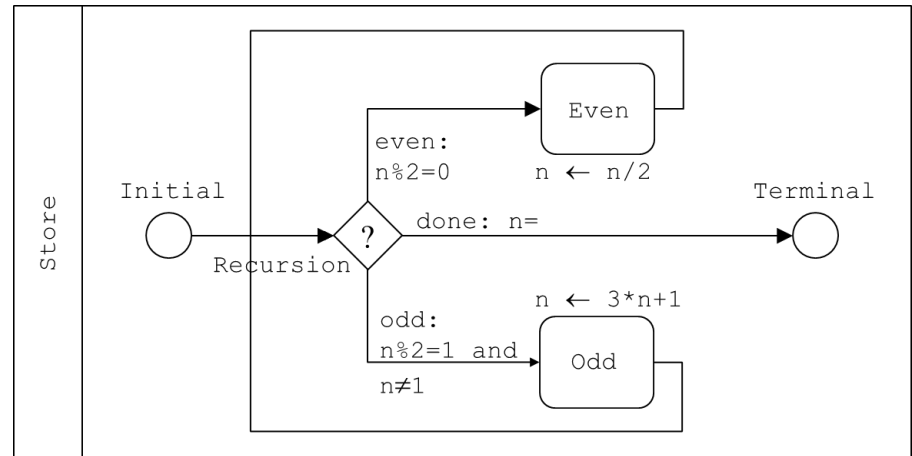


Syracuse (1)



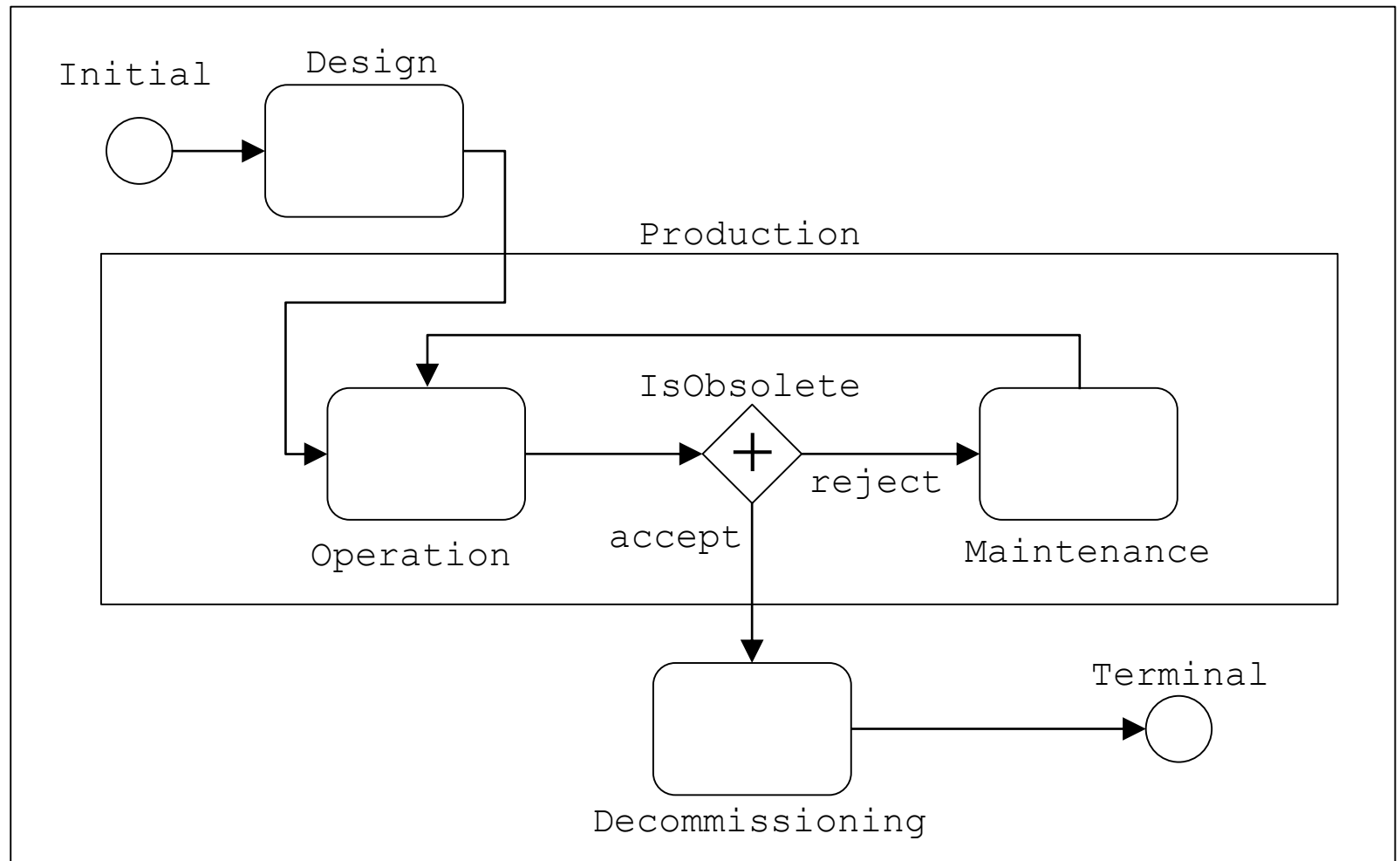
Syracuse (2)

```
scenario Syracuse as Store
  state Initial
  test Recursion
    case done (eq n 1)
    case even (eq (mod n 2) 0)
    case odd (and (eq (mod n 2) 1) (df n 1))
  end
  task Even
    set n (div n 2)
  end
  task Odd
    set n (add (mul n 3) 1)
  end
  state Terminal
  next Initial Recursion
  next Recursion.done Terminal
  next Recursion.even Even
  next Recursion.odd Odd
  next Even Recursion
  next Odd Recursion
end
```

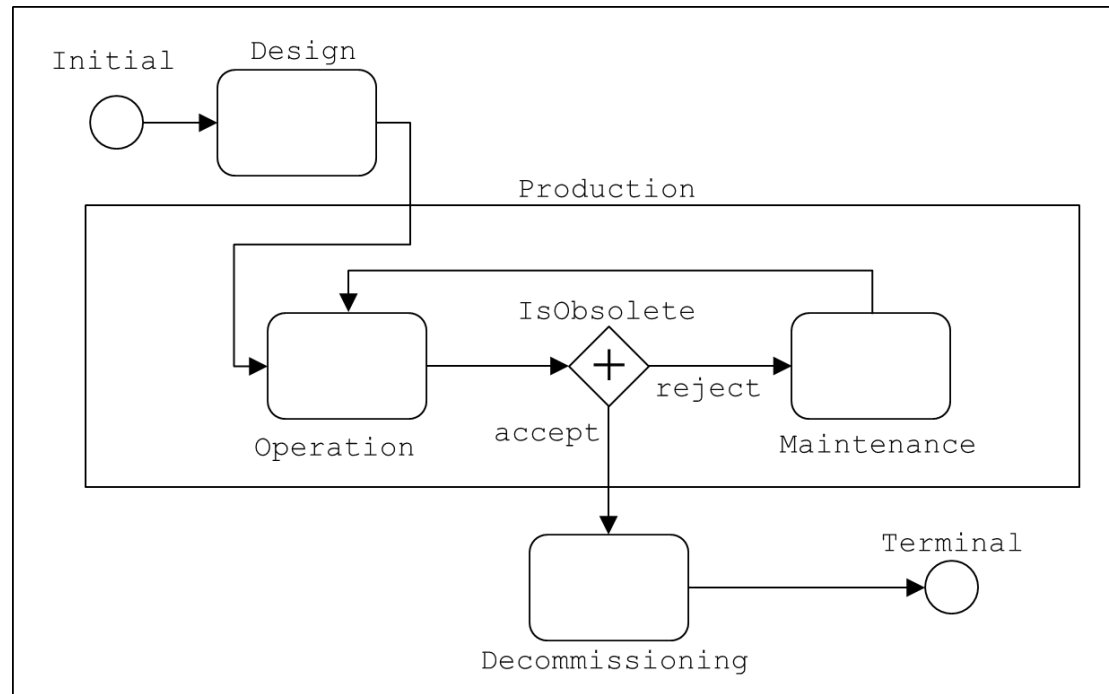


Whether this series converges to 1 for all value of n is still an open question.

Life-Cycle (1)



Life-Cycle (2)



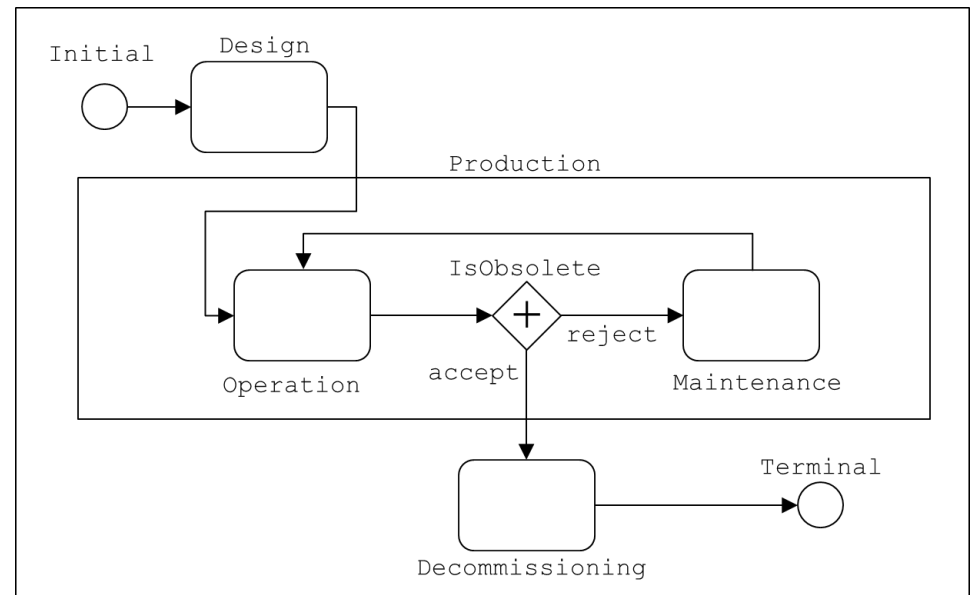
This scenario applies on a system made of a single (implicit) component. It describes the life-cycle of this system.

It involves the sub-scenario `Production` and the choice gateway `IsObsolete`. Choice gateways describe non-deterministic choices: it is up to the user to tell the simulator which branch to take during the simulation.

Life-Cycle (3)

```

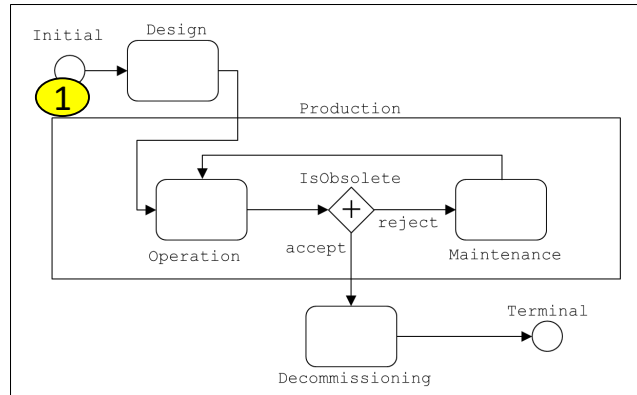
scenario Lifecycle
  state Initial
  task Design end
  scenario Production
    task Operation end
    choice IsObsolete
      branch accept
      branch reject
    end
    task Maintenance end
  next Operation IsObsolete
  next IsObsolete.reject Maintenance
  next Maintenance Operation
end
task Decommissioning end
state Terminal
next Initial Design
next Design Production.Operation
next Production.IsObsolete.accept Decommissioning
end
  
```



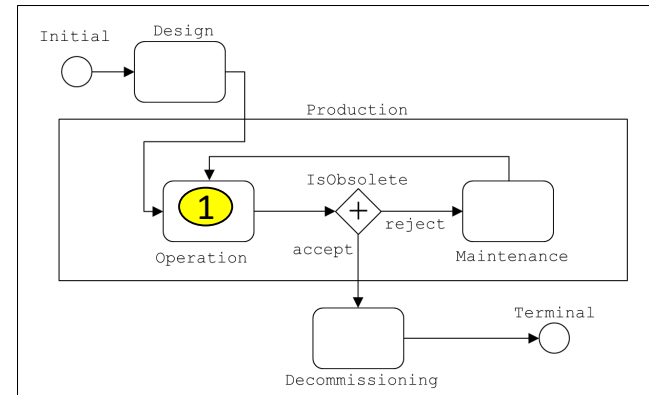
- Sub-scenarios are like macro-states in statecharts

Life-Cycle (4)

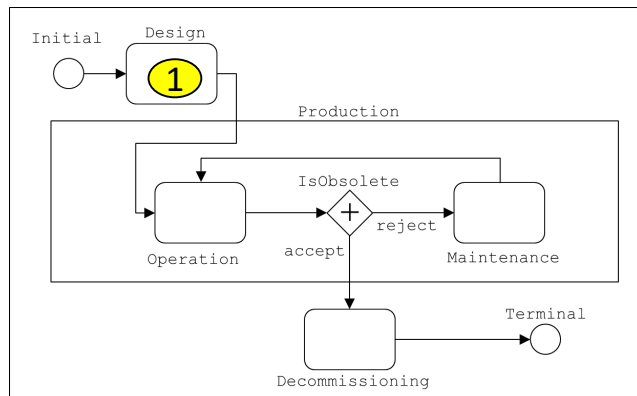
1



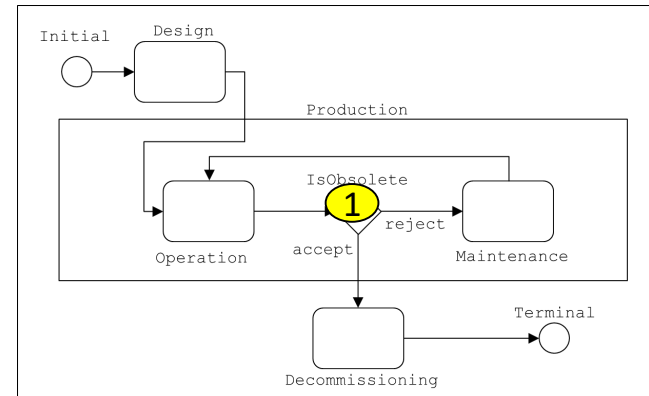
3



2

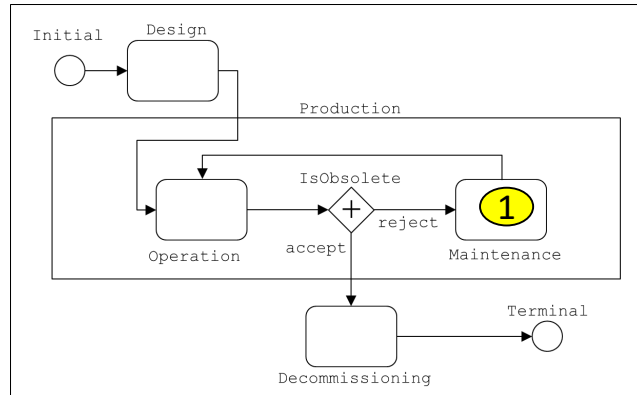


4

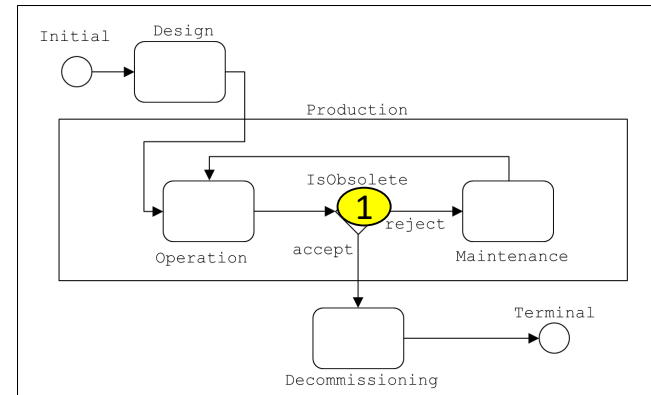


Life-Cycle (5)

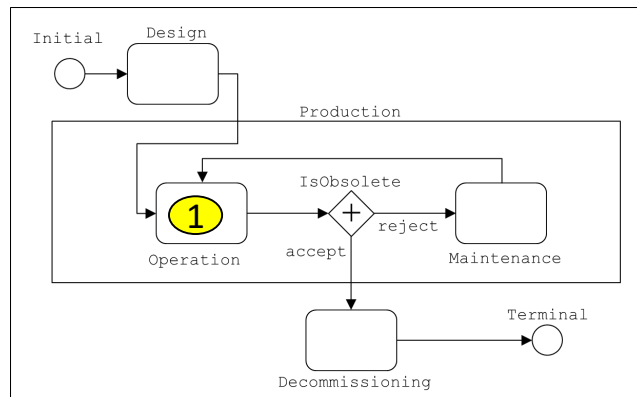
5



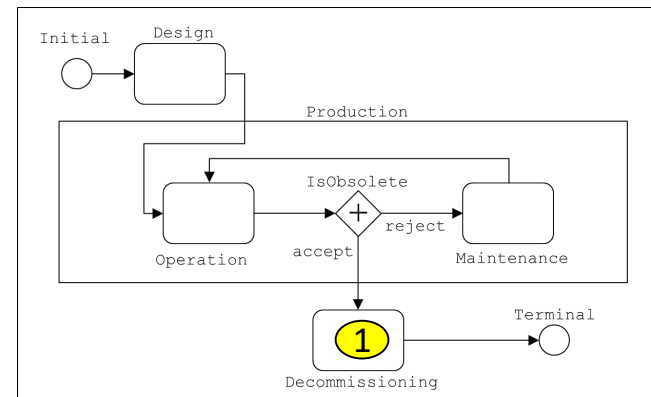
7



6

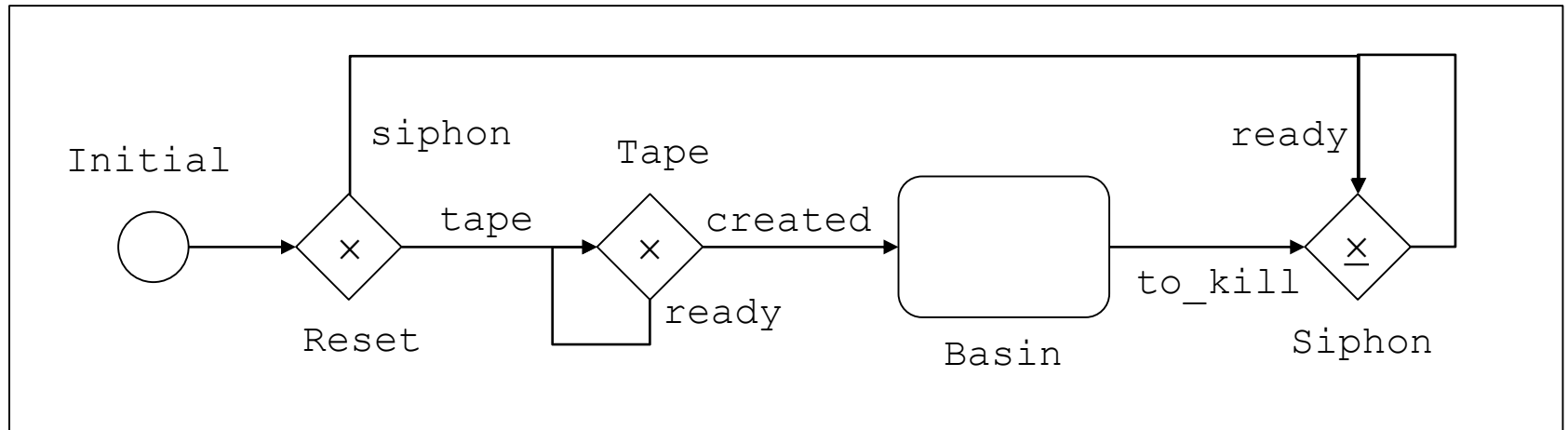


8



and so on...

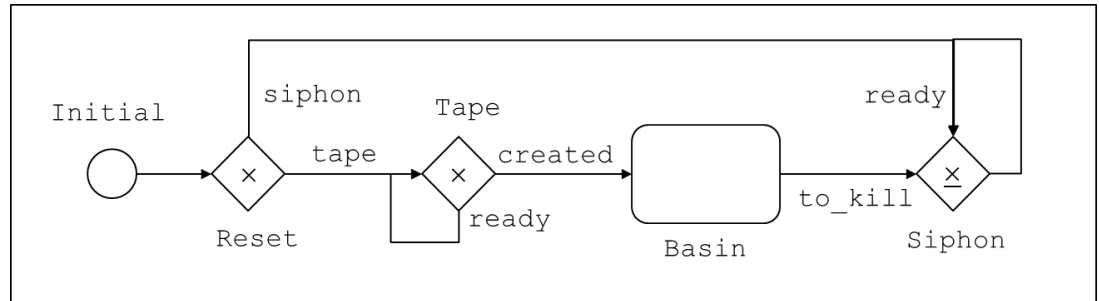
Tapes and Siphons (1)



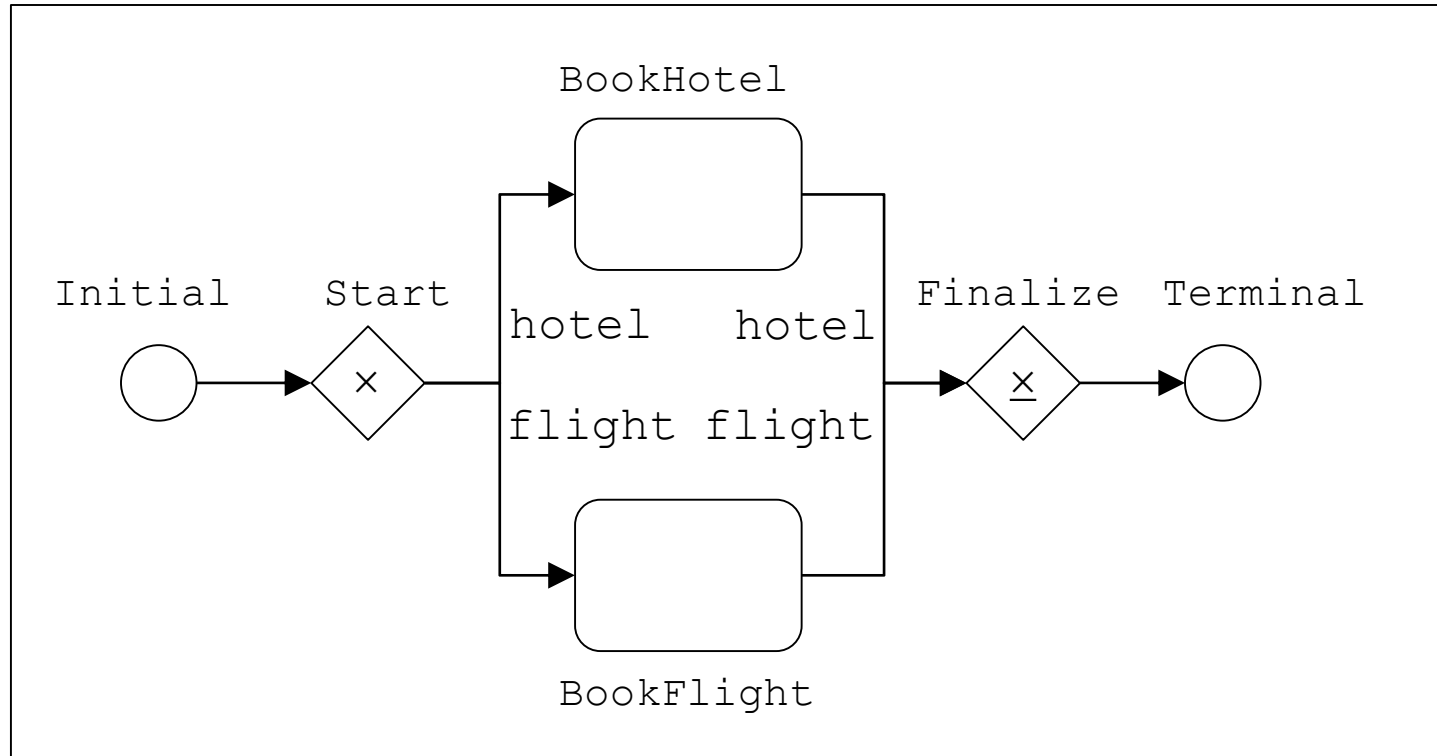
A model that, at the one end, creates as many processes as the analyst wishes (Tape) and, at the other end, kills these processes (Siphon).

Tapes and Siphons (2)

```
scenario TapeAndSiphon
  state Initial
  fork Reset
    branch tape
    branch siphon
  end
  fork Tape
    branch ready
    branch created
  end
  task Basin end
  join Siphon
    branch ready
    branch to_kill
  end
  next Initial Reset
  next Reset.tape Tape
  next Reset.siphon Siphon.ready
  next Tape.ready Tape
  next Tape.created Basin
  next Basin Siphon.to_kill
  next Siphon Siphon.ready
end
```



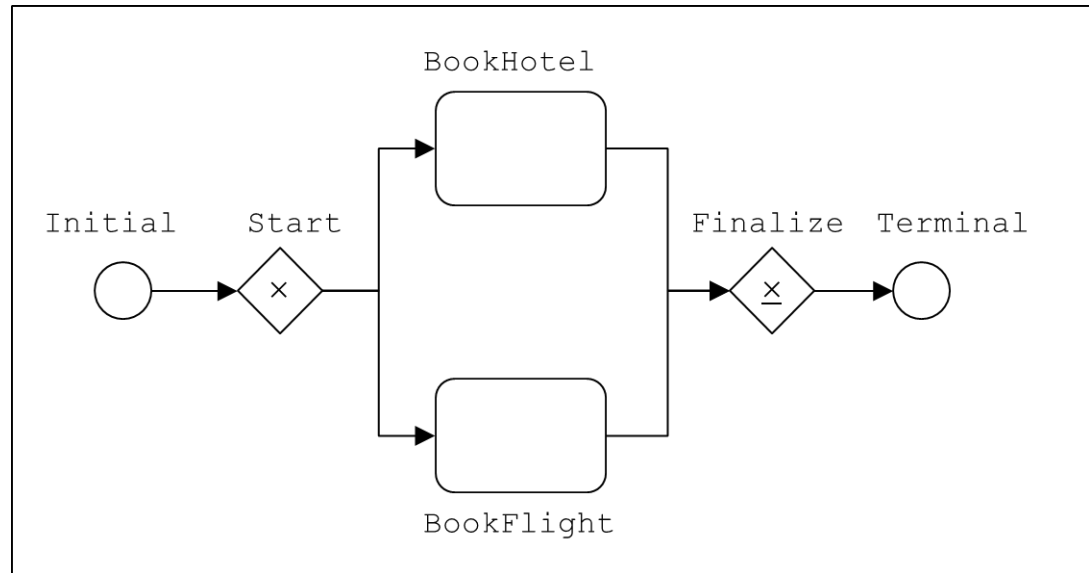
Travel Reservation (1)



In this example, the fork gateway **Start** creates two processes out of one while the join gateway **Finalize** creates one process out of two.

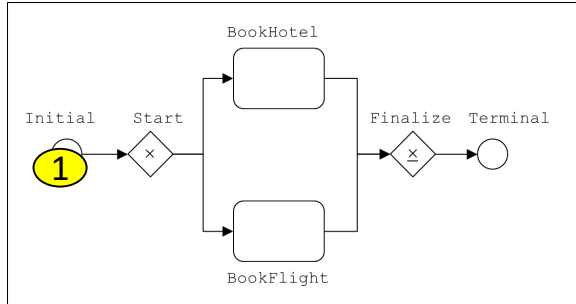
Travel Reservation (2)

```
scenario TravelReservation
  state Initial
  fork Start
    branch hotel
    branch flight
  end
  task BookHotel end
  task BookFlight end
  join Finalize
    branch hotel
    branch flight
  end
  state Terminal
  next Initial Start
  next Start.hotel BookHotel
  next Start.flight BookFlight
  next BookHotel Finalize.hotel
  next BookFlight Finalize.flight
  next Finalize Terminal
end
```

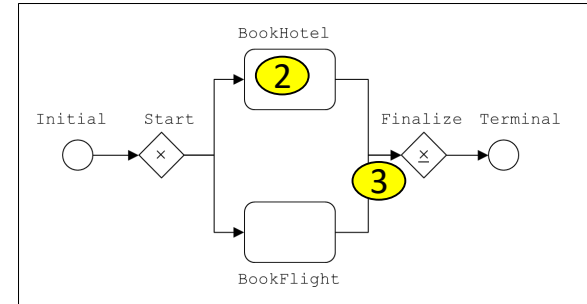


Travel Reservation (3)

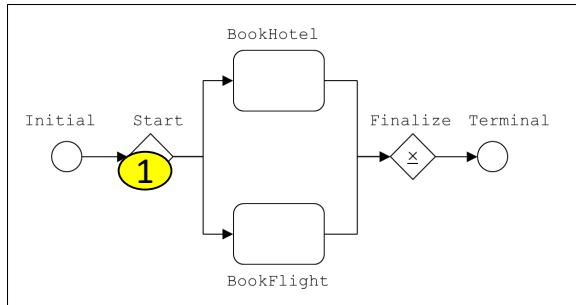
1



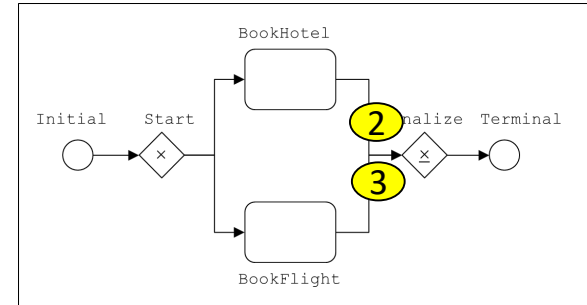
4



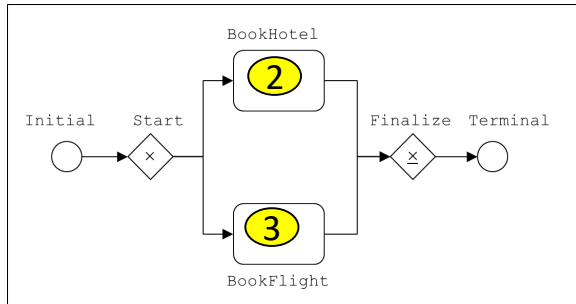
2



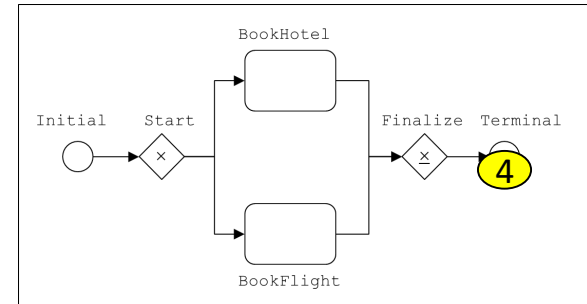
5



3

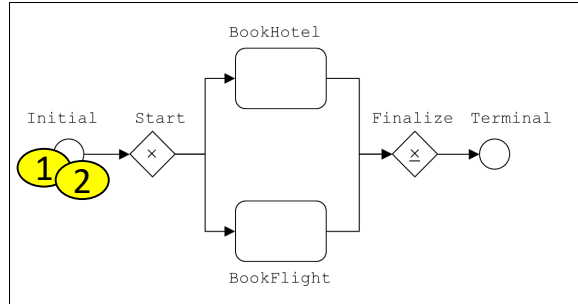


6

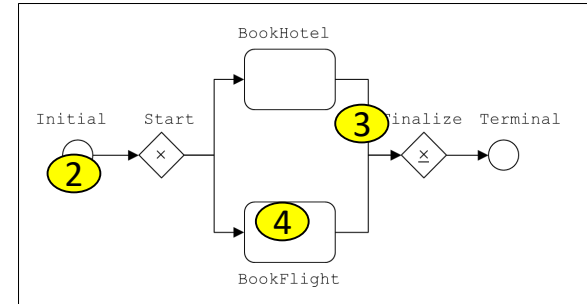


Travel Reservation (4)

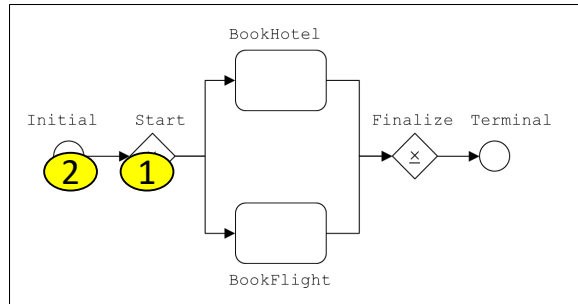
1



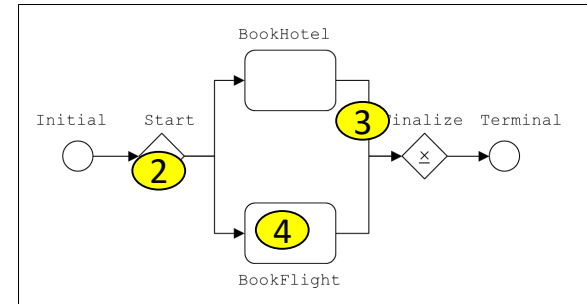
4



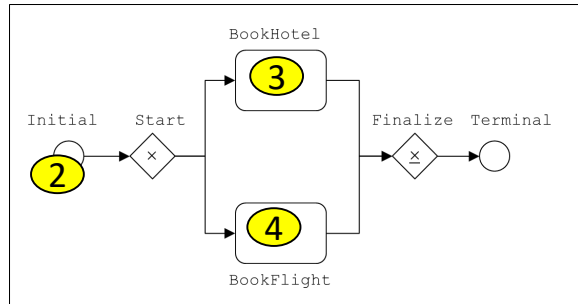
2



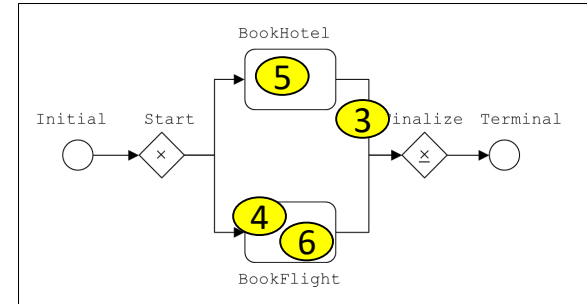
5



3

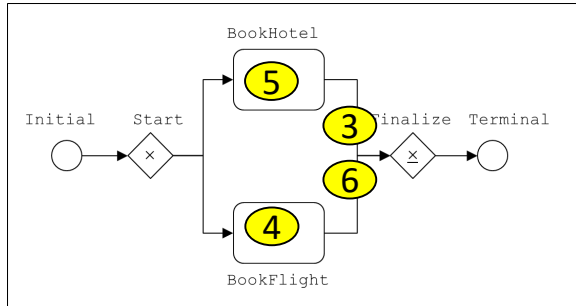


6

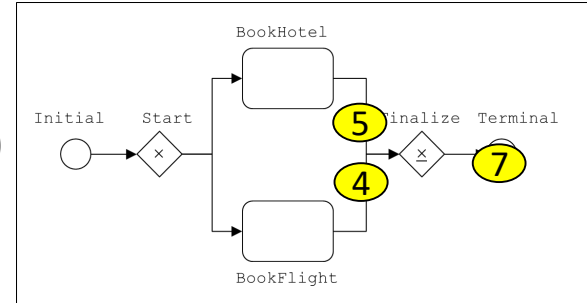


Travel Reservation (5)

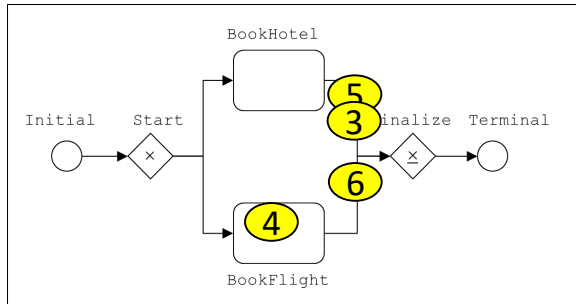
7



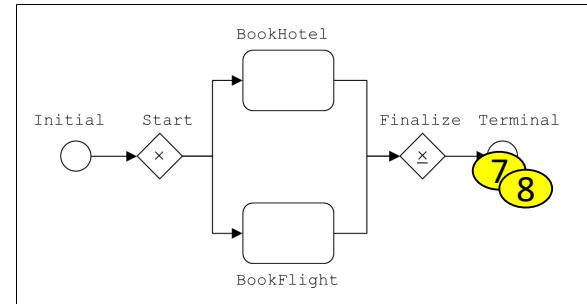
10



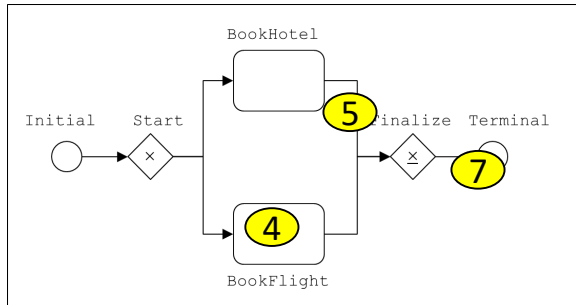
8



11



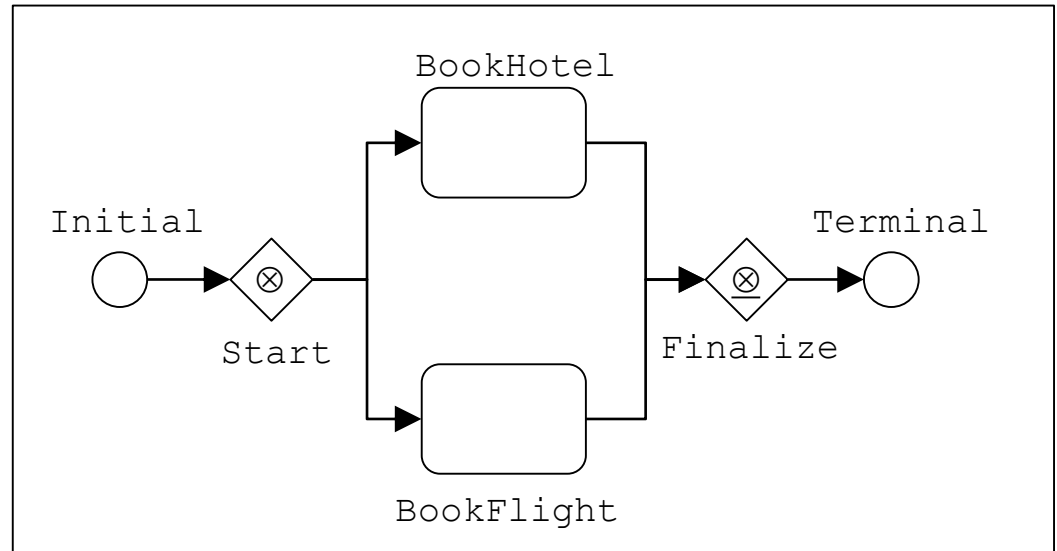
9



Is there a problem?

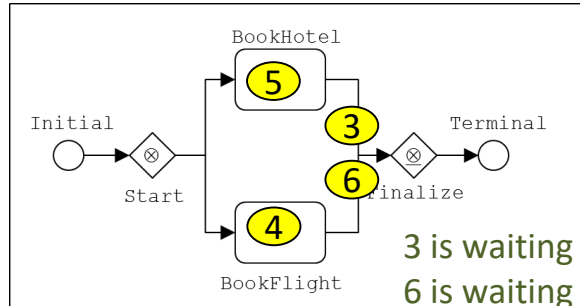
Travel Reservation (6)

```
scenario TravelReservation
  state Initial
  split Start
    branch hotel
    branch flight
  end
  task BookHotel end
  task BookFlight end
  merge Finalize
    branch hotel
    branch flight
  end
  state Terminal
  next Initial Start
  next Start.hotel BookHotel
  next Start.flight BookFlight
  next BookHotel Finalize.hotel
  next BookFlight Finalize.flight
  next Finalize Terminal
end
```

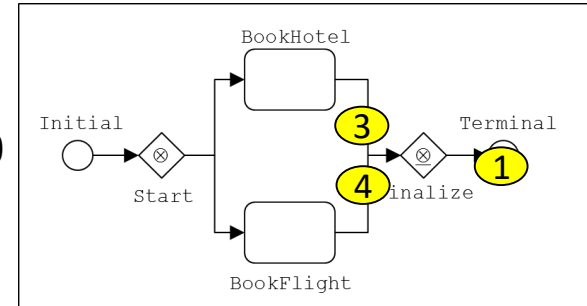


Travel Reservation (7)

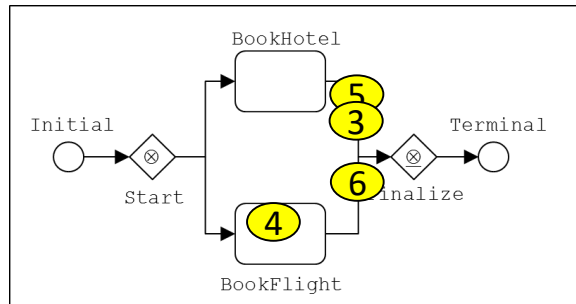
7



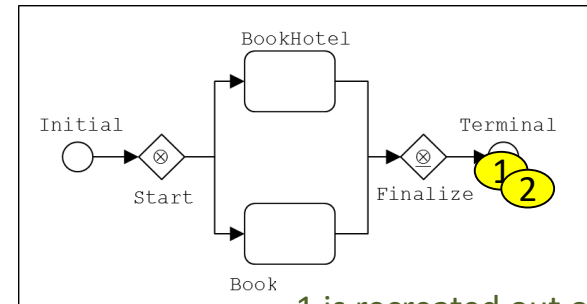
10



8



11



9

