Utrecht University

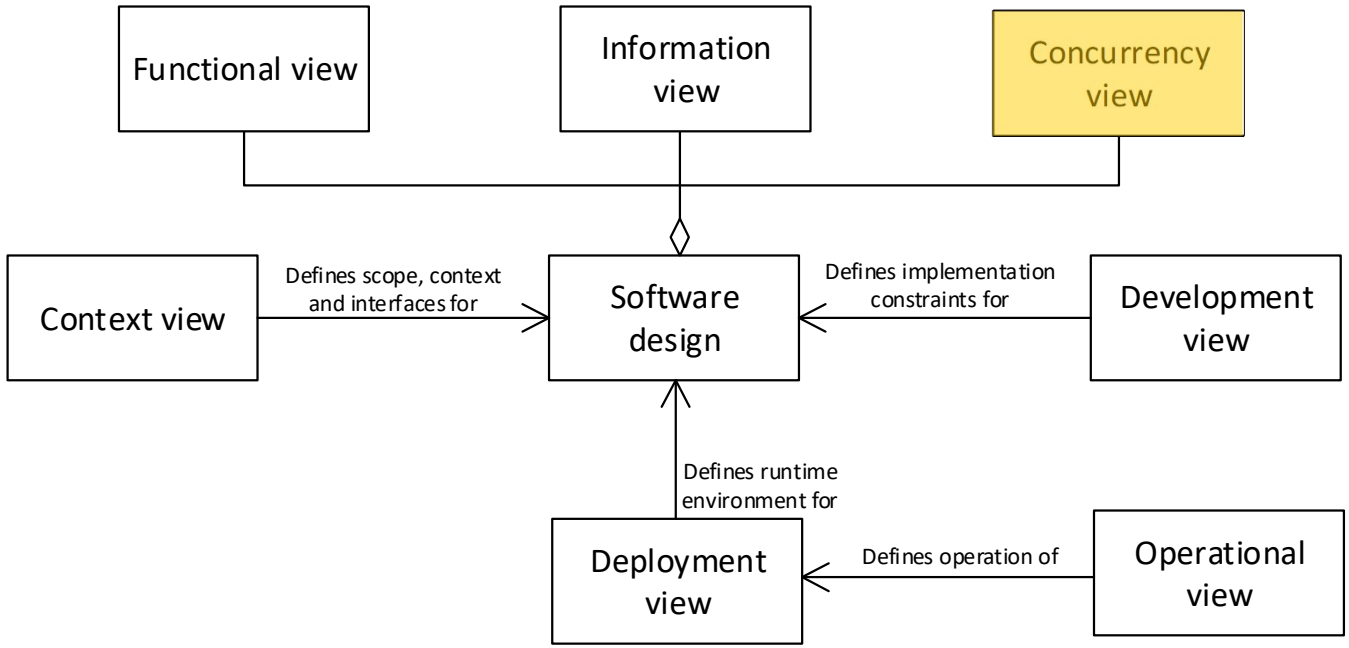
26-2-2024

Lecture 7: concurrency viewpoint II

Jan Martijn van der Werf

Utrecht University

Viewpoint catalog

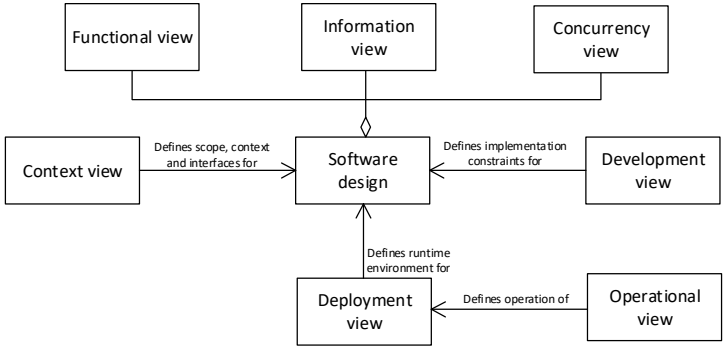


Viewpoint:

Collection of patterns, templates and conventions for constructing one type of view. It defines the stakeholders whose concerns are reflected in the viewpoint and the guidelines, principles, and template models for constructing its views

Utrecht University

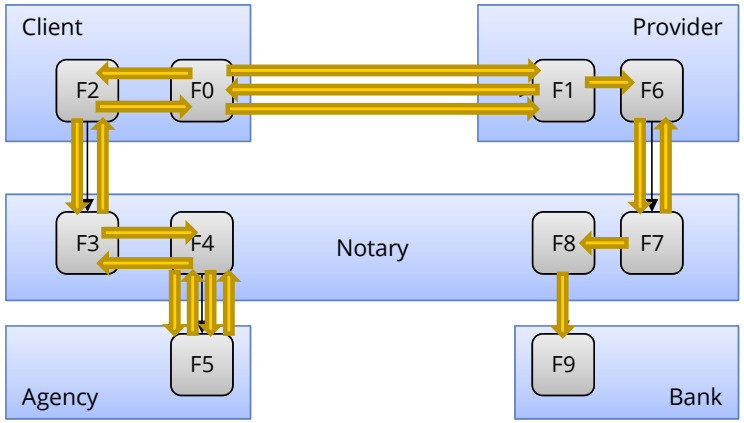
Concurrency view



- Concurrency view:
  - **Describes the concurrency structure of the system and maps functional elements to concurrency units to clearly identify the parts of the system that can execute concurrently and how this is coordinated and controlled**
- Concerns
  - **Task structure, mapping of functional elements to tasks,**
  - **Inter-process communication,**
  - **State management,**
  - **Synchronization and integrity,**
  - **Supporting scalability, task failure,**
  - **Startup and shutdown, re-entrancy**
- Models and views
  - **System level concurrency models**
  - **State models, protocol models**

Utrecht University

Logical model and scenarios



- Scenario is a sequence of function calls
- Formal definition:

  Given a logical model $(C, F, h, \rightarrow)$**,**
  a scenario is a partial order over the function calls,
  i.e., $\sigma \in (\rightarrow)^*$ **such that:**

  Functions can only start if being called before:

  $$\forall 1 < i < |\sigma| : \left(\exists 1 < j < i : \pi_3(\sigma(j)) = \pi_1(\sigma(i))\right)$$

Problem: scenarios can be conflicting!

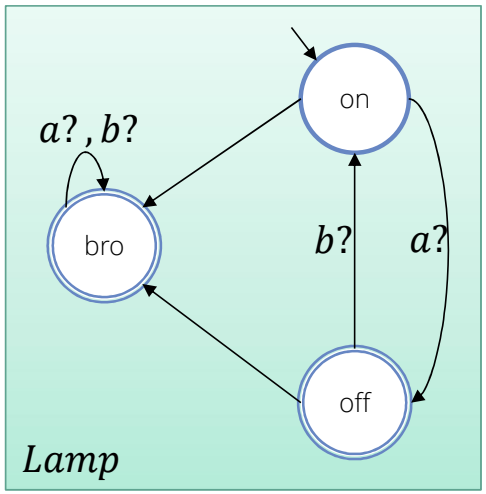
Key:
F0: Request service
F1: Handle service request
F2: Request approval
F3: Receive approval request
F4: Validate client
F5: Do credibility check
F6: Request payment
F7: Check payment request
F8: Send payment
F9: Make payment

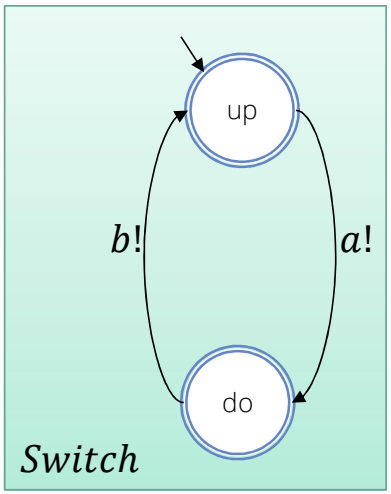
Given a set A, a sequence of length $n \in \mathbb{N}$ is a function $\sigma : \{1..n\} \rightarrow A$

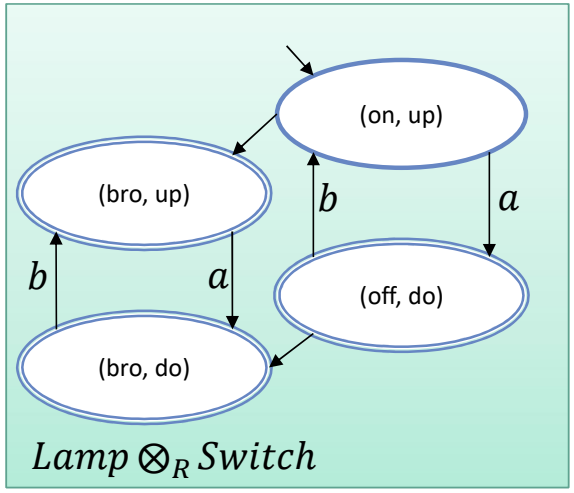
- We write $\sigma = \langle a_1, \ldots, a_n \rangle$ if $\sigma(i) = a_i$ for all $1 \leq i \leq n$.
- We denote its length by $|\sigma|$
- If $n = 0$, we call it the empty sequence, and denote it with ϵ
- The set of all finite sequences over A is denoted by A^*

Utrecht University

Composing interface automata



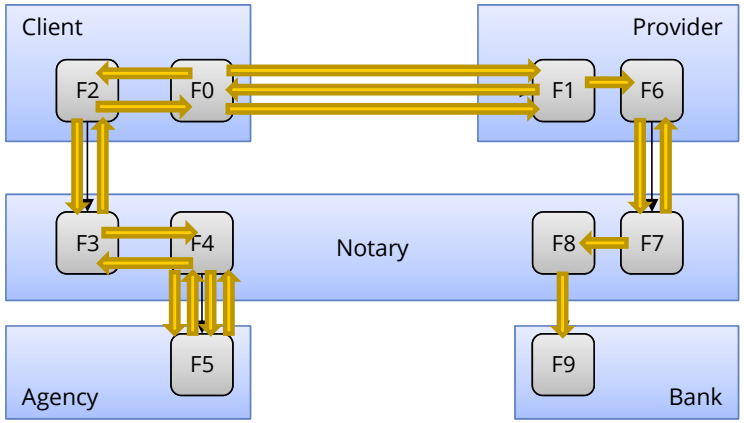




$R = \{a, b\}$

Utrecht University

Communication mechanisms



- Synchronous communication
  You have to know in which state the other is!
- Asynchronous communication
  Send messages to the other, the state of the other is unknown!

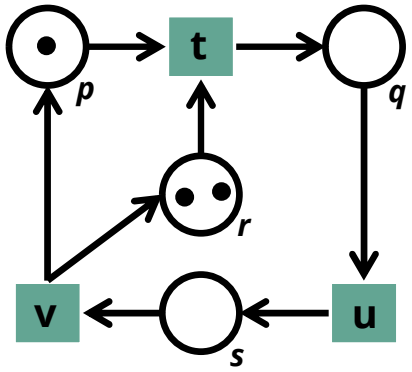
Key:
F0: Request service
F1: Handle service request
F2: Request approval
F3: Receive approval request
F4: Validate client
F5: Do credibility check
F6: Request payment
F7: Check payment request
F8: Send payment
F9: Make payment

Utrecht University

Petri nets

Utrecht University

Petri nets: formal definition



$N = ((P, T, F), m_0)$ with:

$P = \{p, q, r, s\}$

$T = \{t, u, v\}$

$F = \{(p,t), (r,t), (t,q), (q,u)$
$(u,s), (s,v), (v,r), (v,p)\}$

$m_0 = [\, p,\ r^2 \,]$

(reads as: $[\, p^1,\ r^2, q^0, s^0]$)

- A Petri net is a 4-tuple N = $((P, T, F), m_0)$ with:

- P: a (finite) set of places
- T: a (finite) set of transitions
- P and T are disjoint ($P \cap T = \emptyset$)
- F is the flow relation that defines the arcs

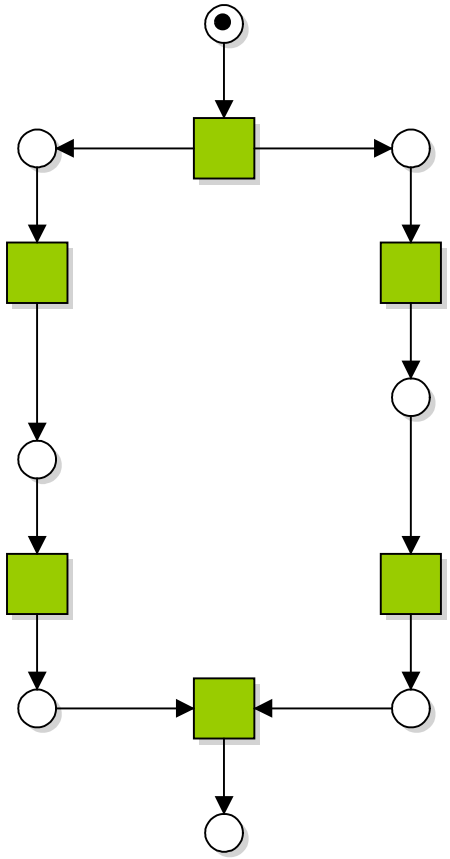
$$F \subseteq (P \times T) \cup (T \times P)$$

- m_0 is the initial marking, gives the tokens per place

$$m_0 : P \rightarrow \mathbb{N}$$

Utrecht University

Petri nets and communication mechanisms

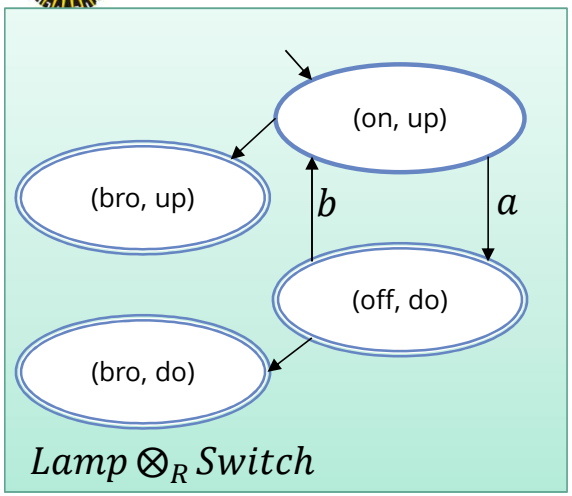


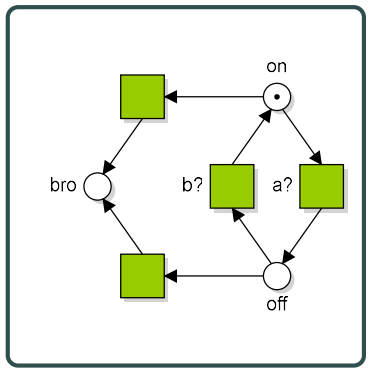
- Synchronous communication
  - **Transitions that consume from multiple places**
- Asynchronous communication
  - **Places resemble pools of messages to be handled**
  - **Random access of messages in these pools**

Both communication mechanisms in a single, **graphical** formalism!

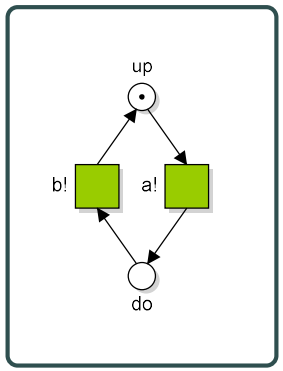
Utrecht University

Back to our lamps: now as Petri nets

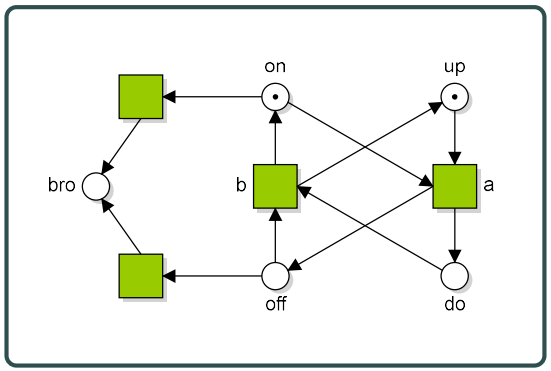




Lamp

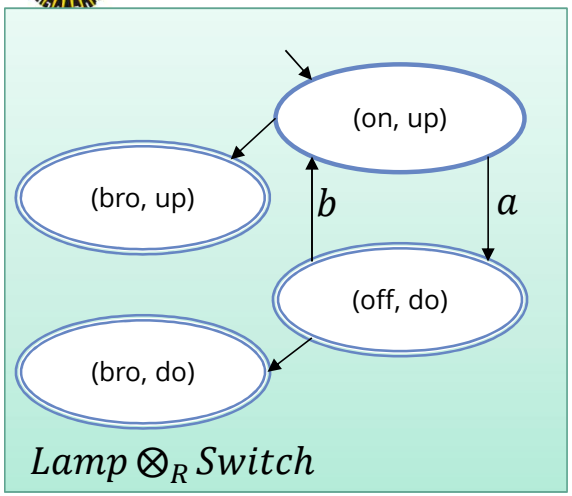


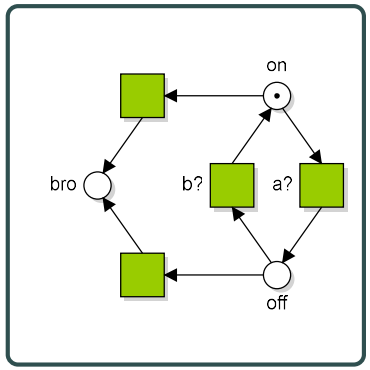
Switch



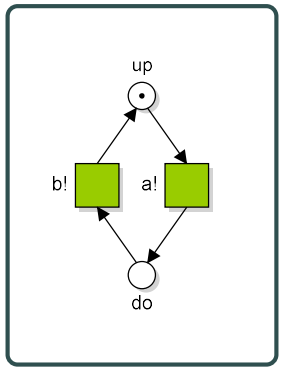
Utrecht University

Back to our lamps: now as Petri nets

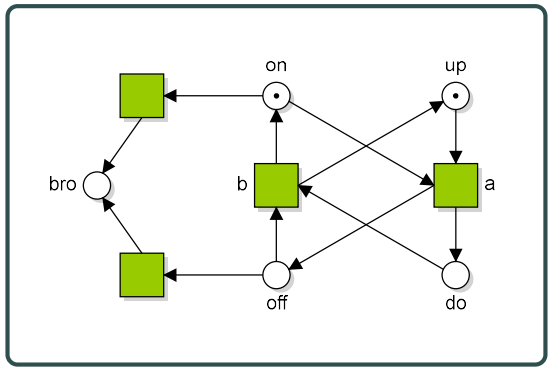


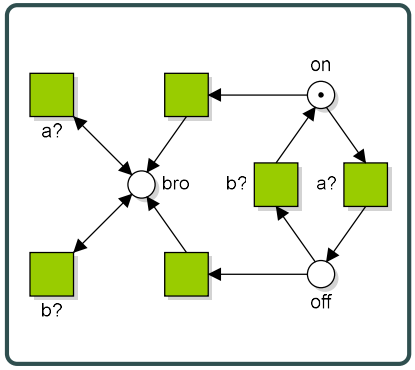


Lamp

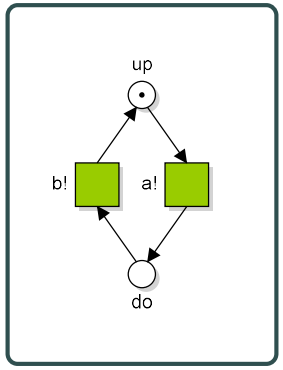


Switch





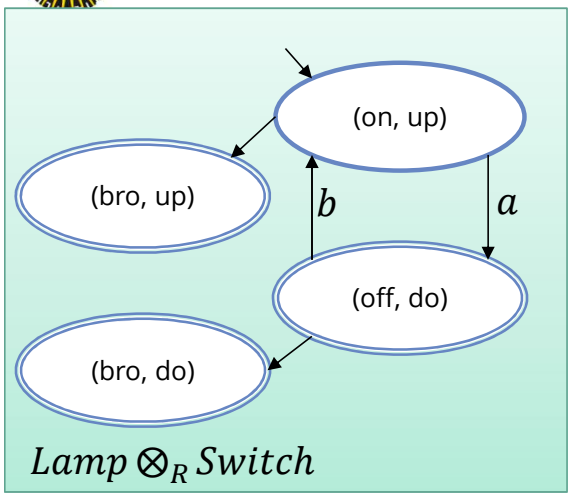
Lamp

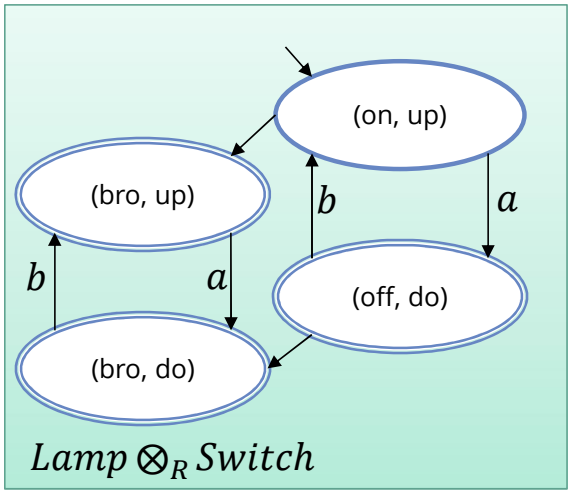


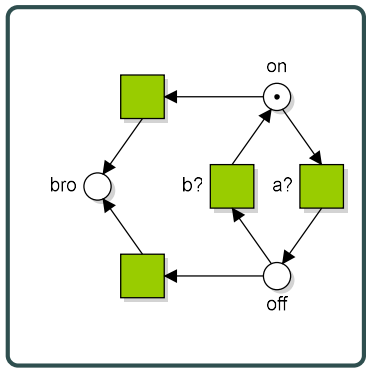
Switch

Utrecht University

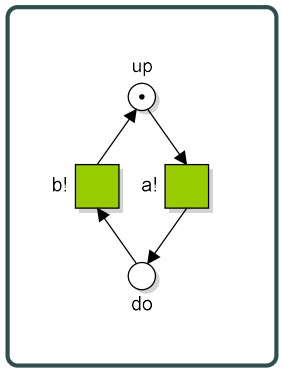
Back to our lamps: now as Petri nets
Synchronous communication



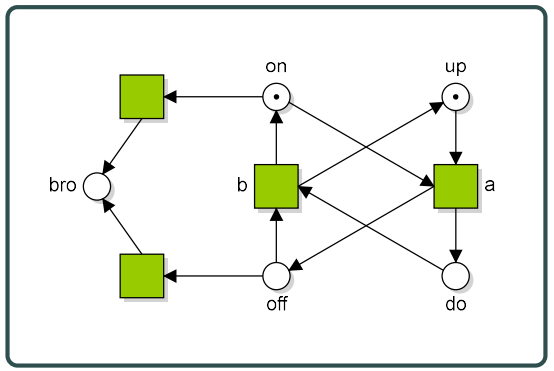


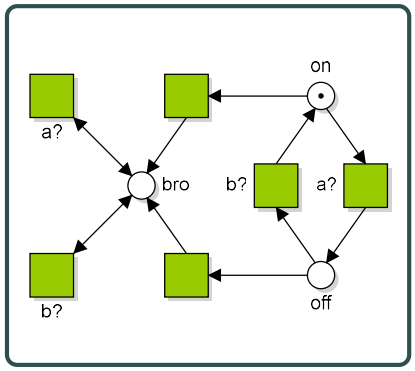


Lamp

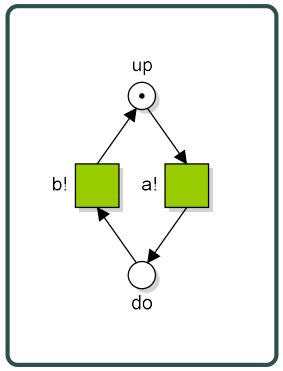


Switch

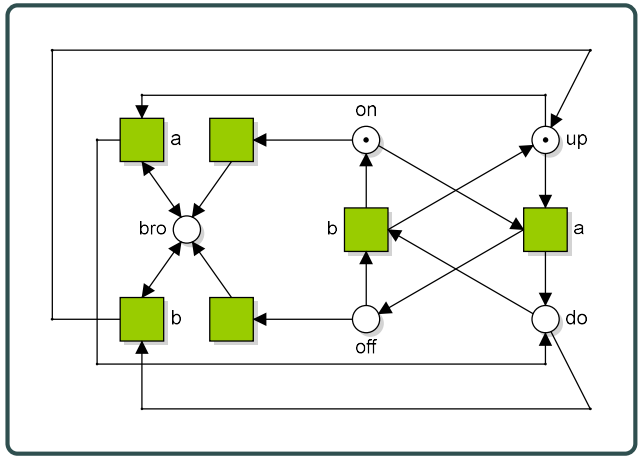




Lamp

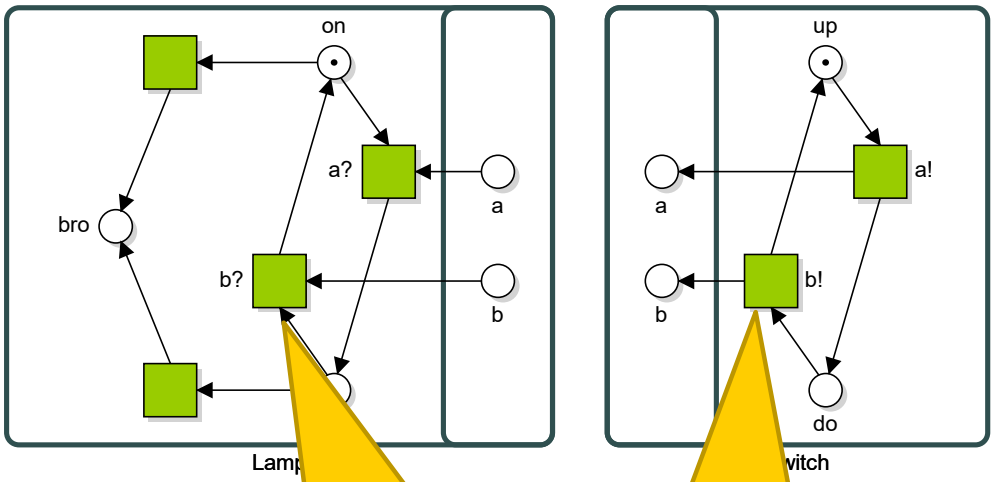


Switch



Utrecht University

Back to our lamps: now as Petri nets
Asynchronous communication!

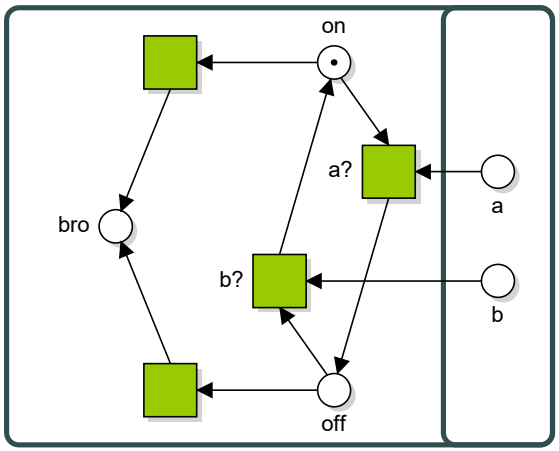


Transition receives a message by consuming a token from “message place” b

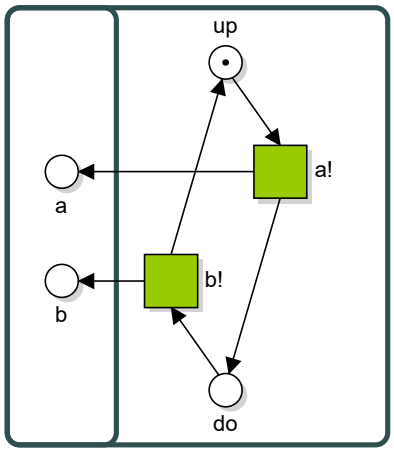
Transition sends a message by placing a token in the “message place” b

Utrecht University

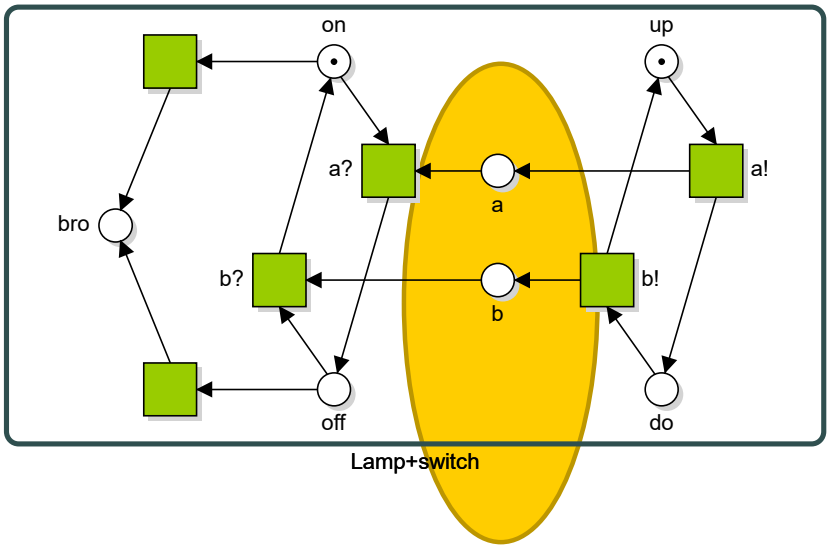
Back to our lamps: now as Petri nets
Asynchronous communication!



Lamp



Switch

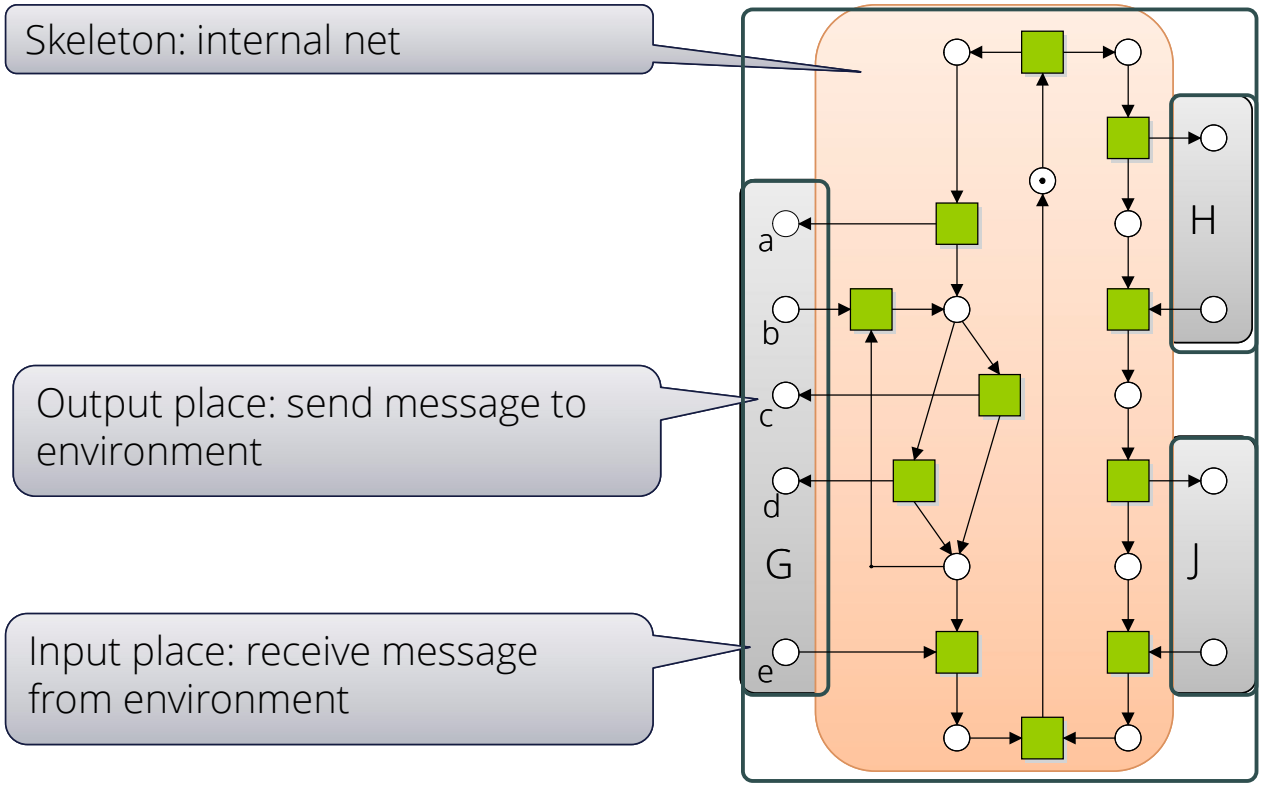


Lamp+switch

Message buffers

Utrecht University

Open nets – asynchronous communication



Utrecht University

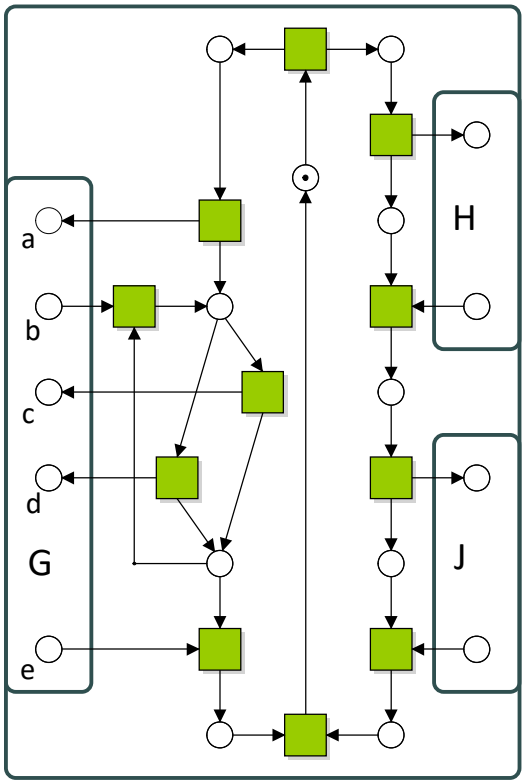
Open nets – verification

1. Stable state
   - **Each component is in rest**
   - **No tokens in the interface**
   - **Typically the initial marking**

Correctness:

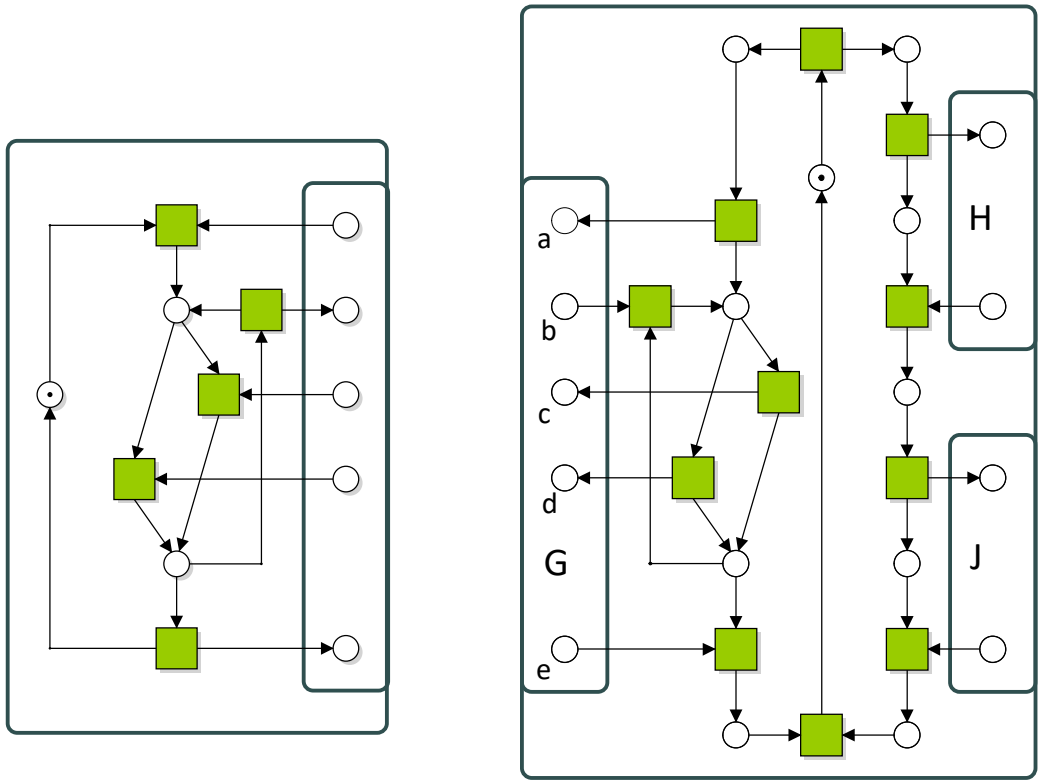
1. Weakly terminating
   - **Always possible to reach a stable state**
2. Proper completion
   - **If a marking covers a stable state, it is a stable state.**

3. Defined on skeleton!



Utrecht University

Open nets – composition



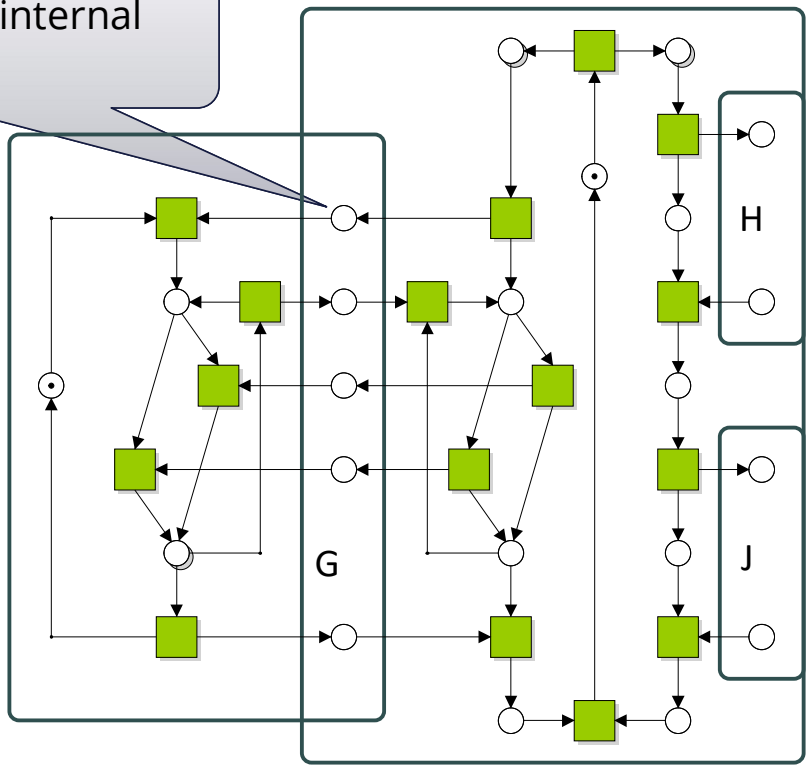
Composition:

- glue interface places with the same name
- Input place of the one should be output of the other

Utrecht University

Open nets – composition

Glued places become internal places of the skeleton

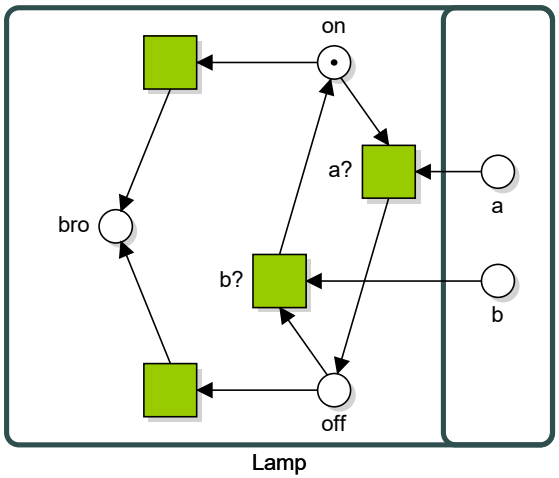


Composition:

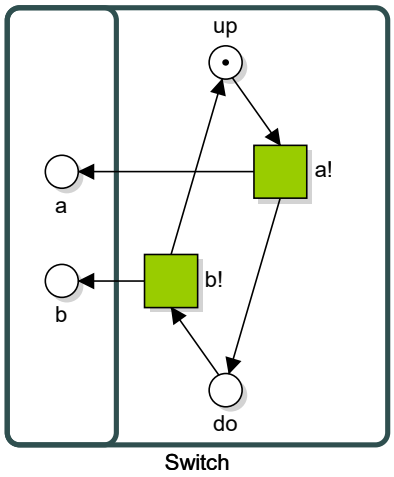
- glue interface places with the same name
- Input place of the one should be output of the other

Utrecht University

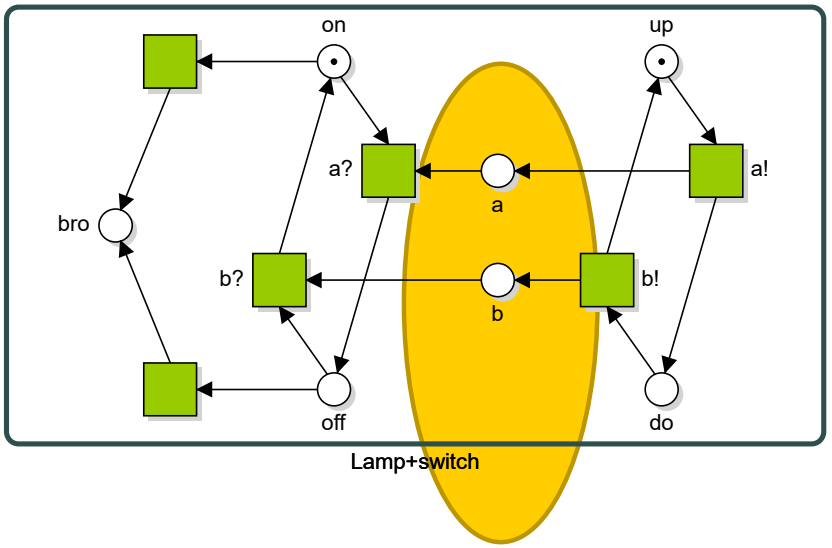
Remember our lamp?



Lamp



Switch



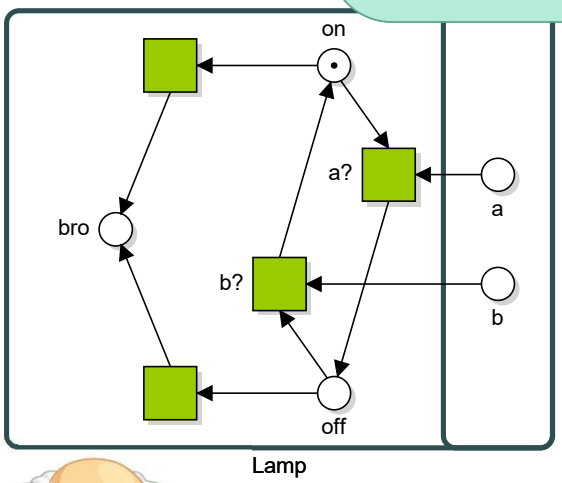
Lamp+switch

Message buffers

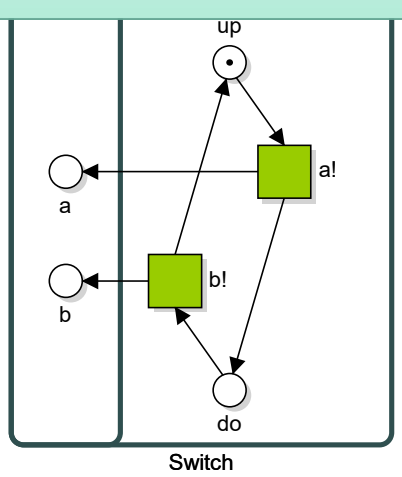
Utrecht Univ

Correctness:

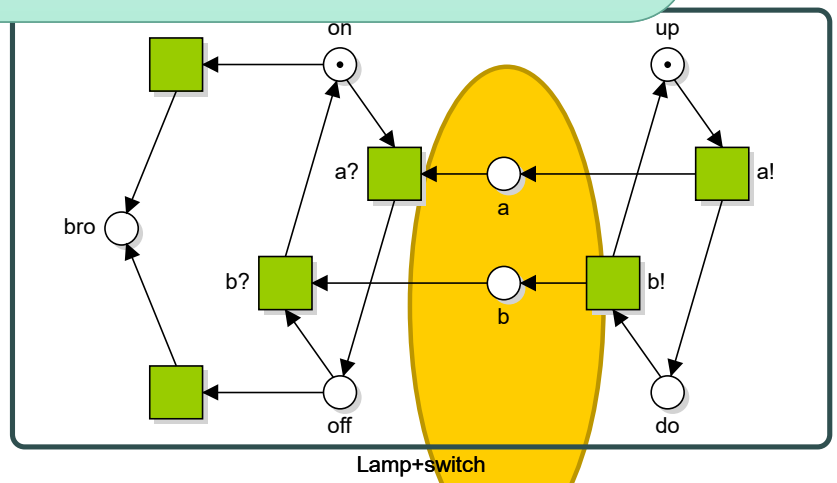
1. Weakly terminating
   Always possible to reach a stable state
2. Proper completion
   If a marking covers a stable state, it is a stable state.



Lamp



Switch



Lamp+switch

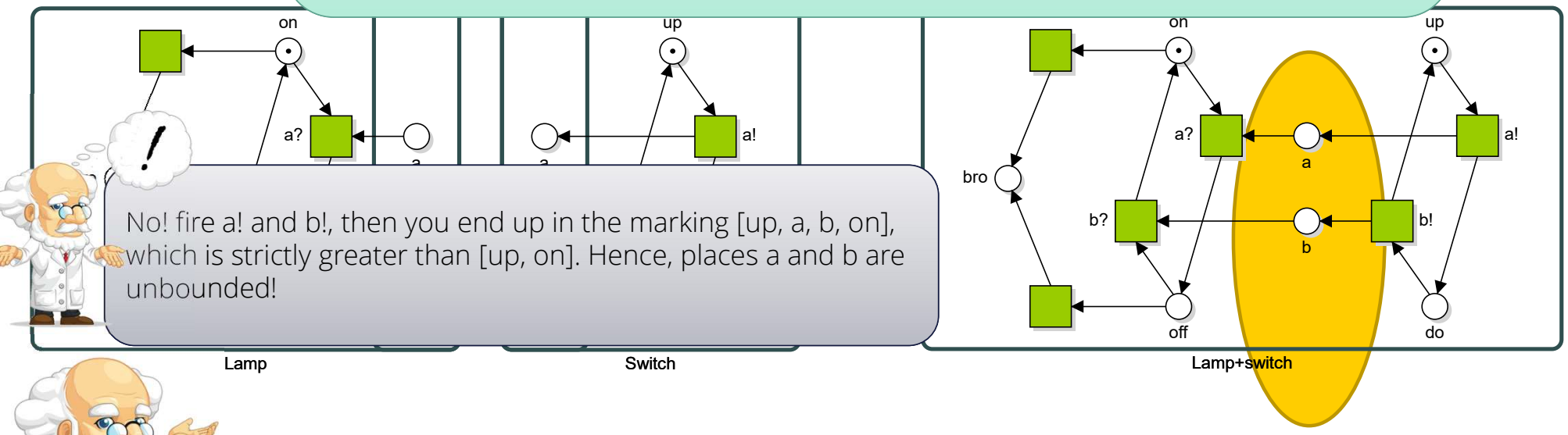
Message buffers

Is this net correct?

Utrecht Univ

Correctness:

1. Weakly terminating
   Always possible to reach a stable state
2. Proper completion
   If a marking covers a stable state, it is a stable state.

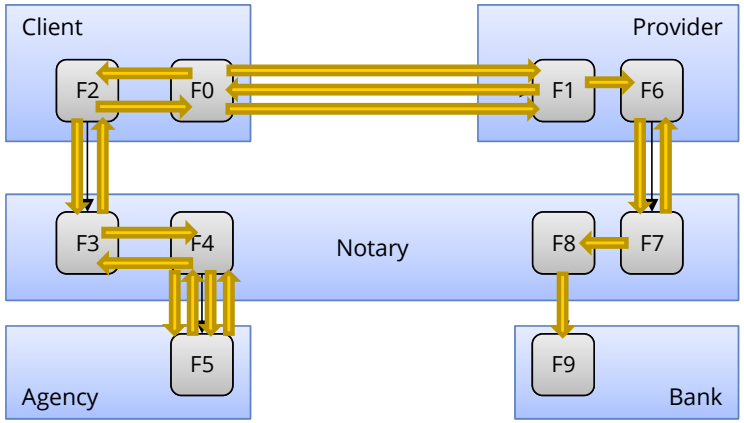


No! fire a! and b!, then you end up in the marking [up, a, b, on], which is strictly greater than [up, on]. Hence, places a and b are unbounded!

Is this net correct?

Utrecht University

Create correct open nets for F0 and F1

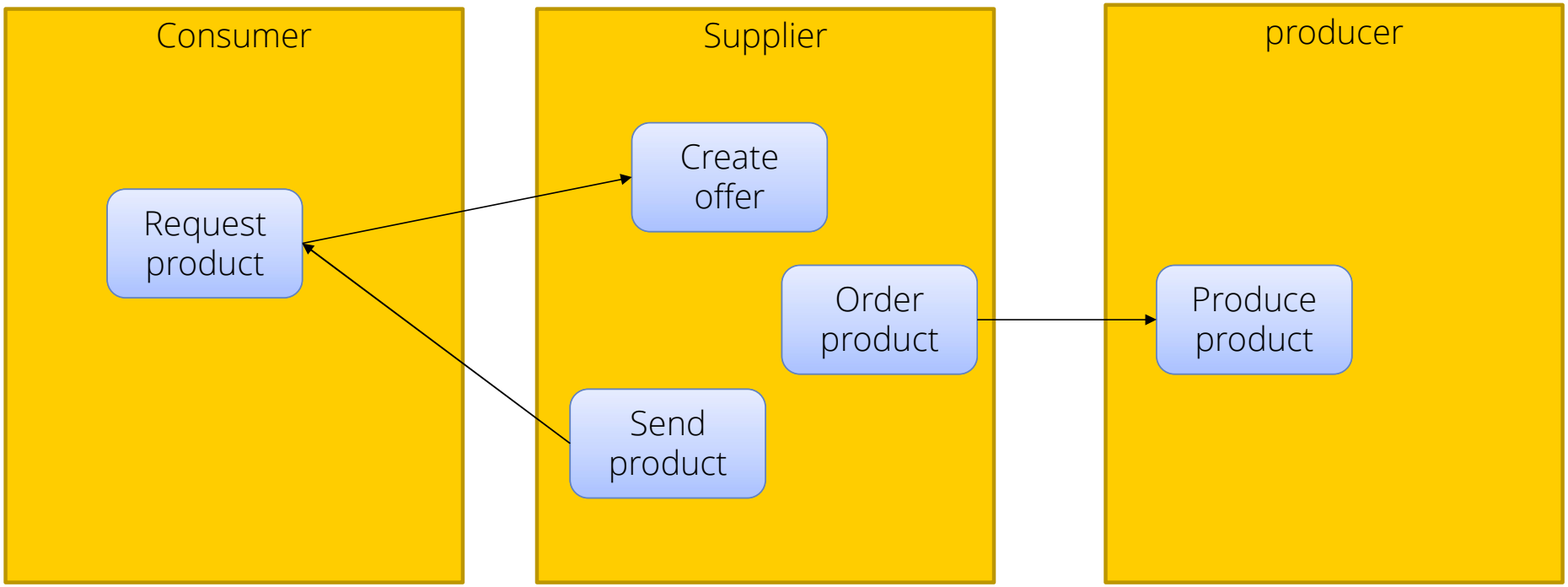


- The client asks for permission. Upon receiving the permission, (s)he asks for a service of the provider, and either accepts or denies the offer received by the provider. Upon accepting, the provider offers the service, and regularly sends an invoice, which needs to be signed and returned by the client. Once denied, either (s)he stops, or (s)he asks for a better offer.
  In case the client does not receive permission, (s)he tries again.

Key:
F0: Request service
F1: Handle service request
F2: Request approval
F3: Receive approval request
F4: Validate client
F5: Do credibility check
F6: Request payment
F7: Check payment request
F8: Send payment
F9: Make payment

Utrecht University

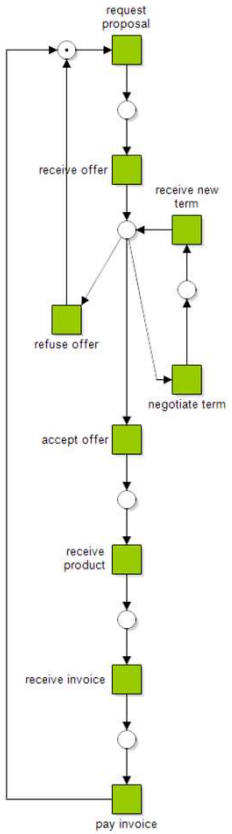
Modelling with Open nets
Step 1: create a logical model



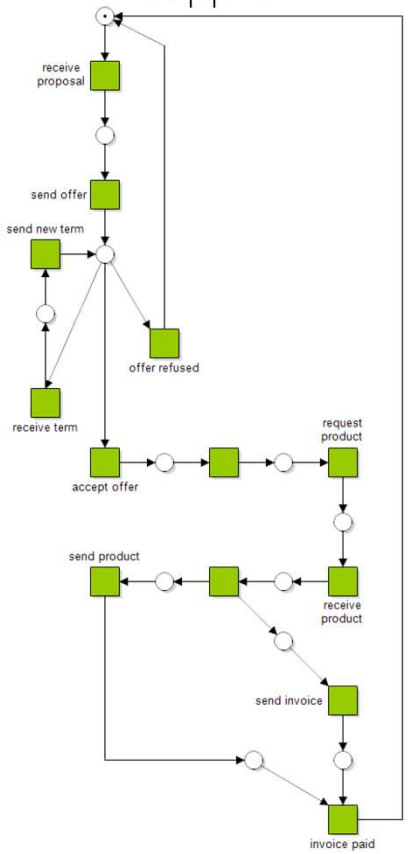
Utrecht University

Modelling with Open nets
Step 2: create an open net for each module

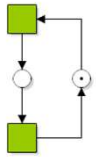
Consumer



Supplier

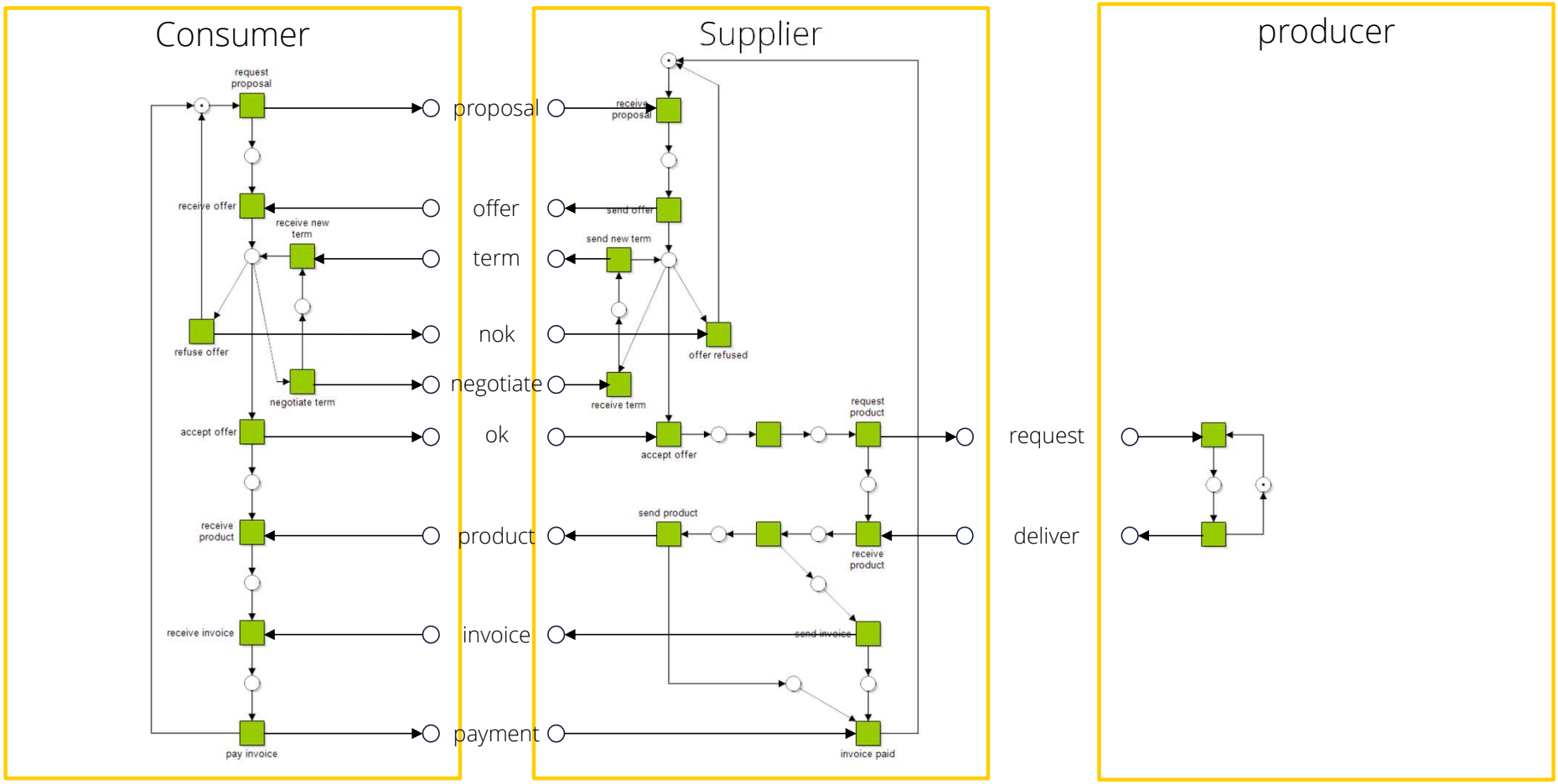


producer



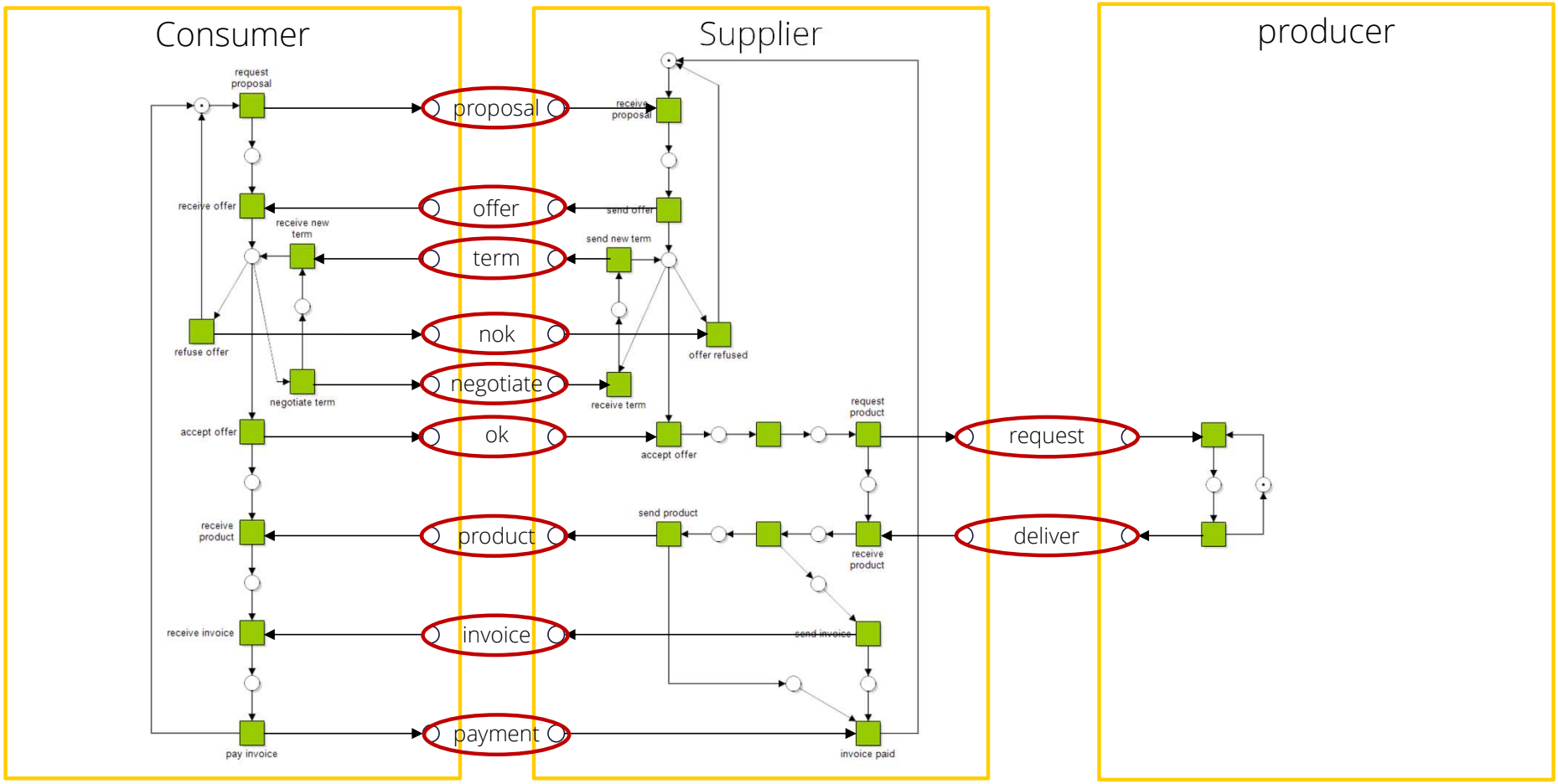
Utrecht University

Modelling with Open nets
Step 3: define the interface places for the modules



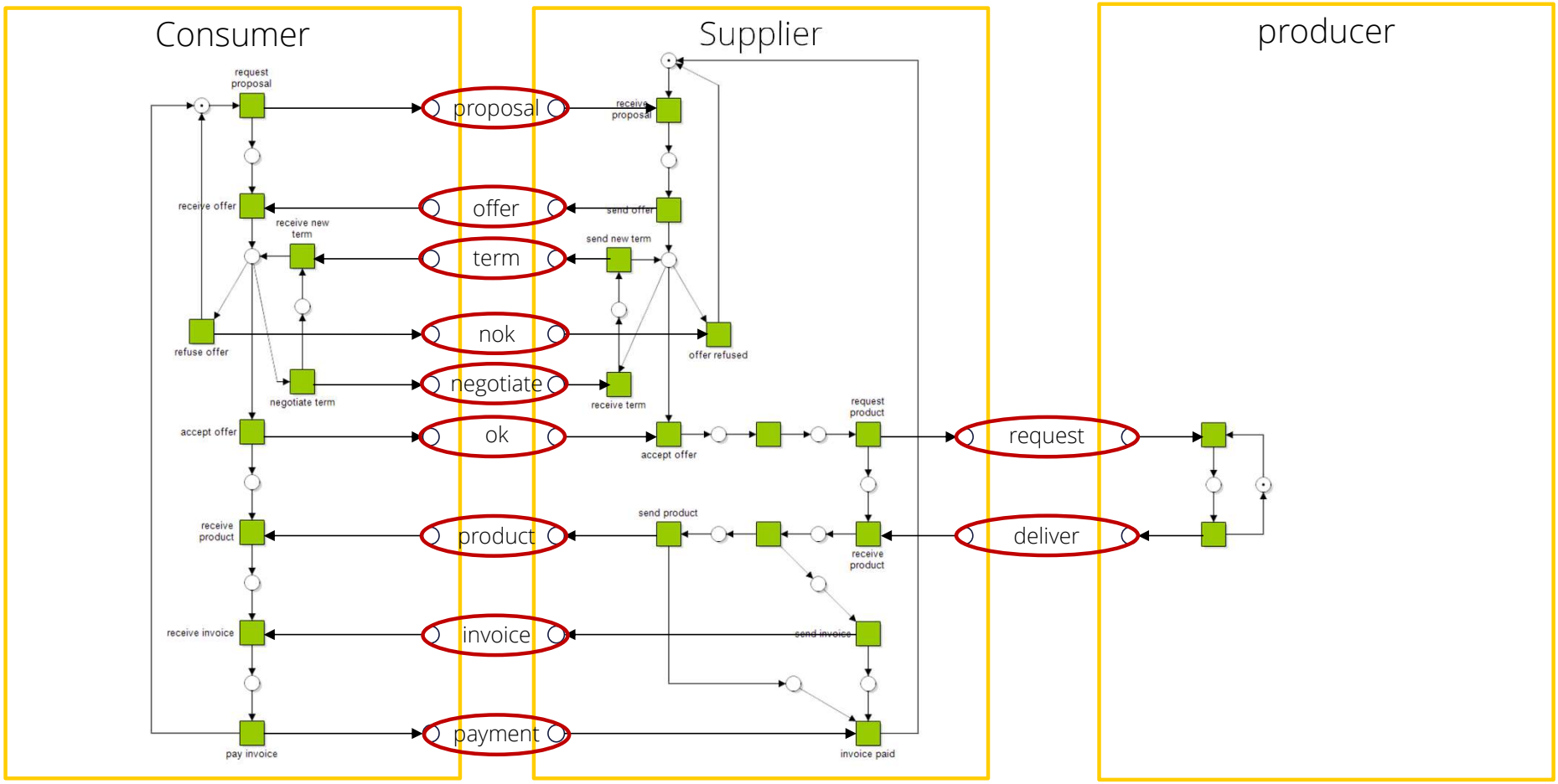
Utrecht University

Modelling with Open nets
Step 4: compose the modules



Utrecht University

Modelling with Open nets
Step 5: analyze the model

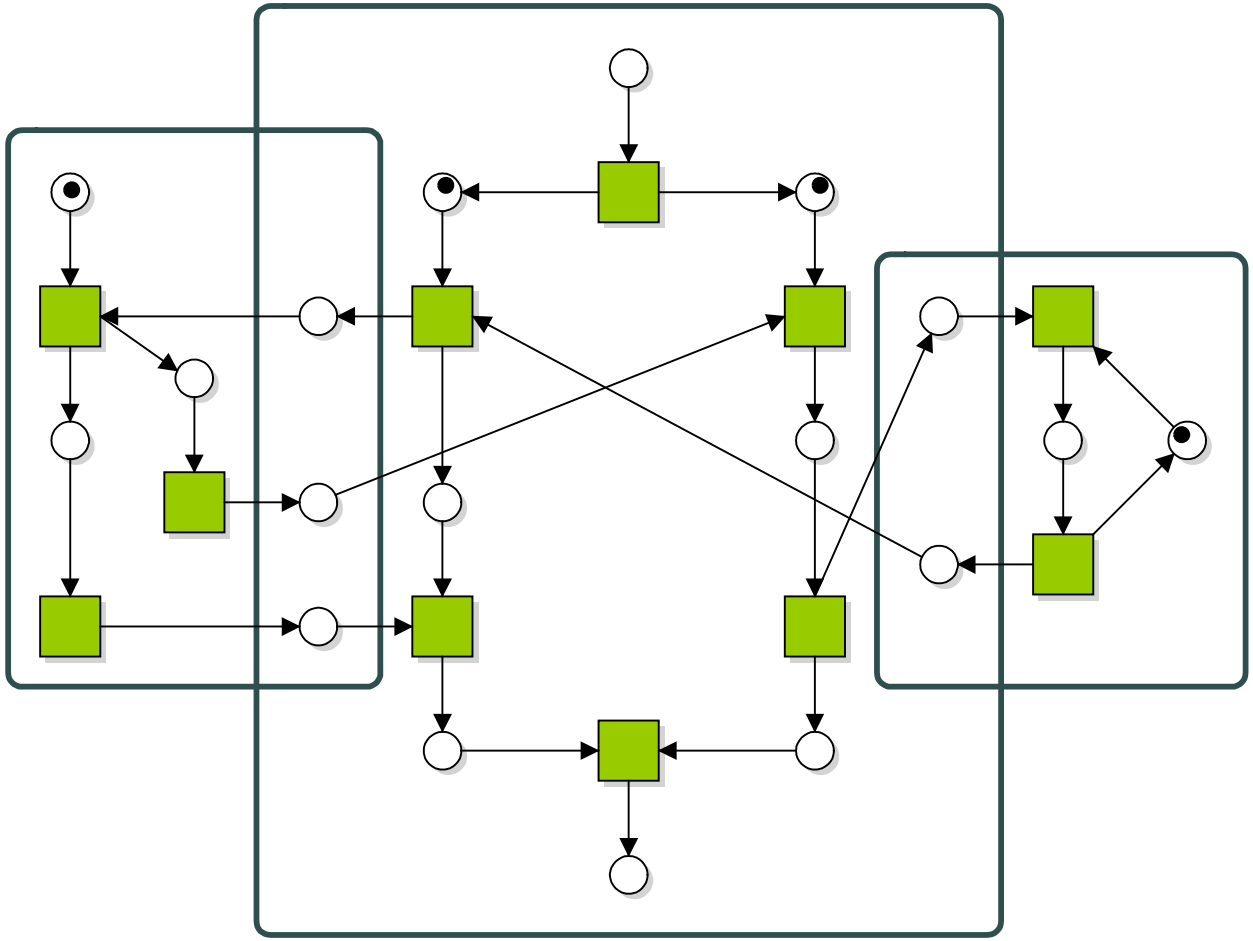


Utrecht University

Correctness:

1. Weakly terminating
   Always possible to reach a stable state
2. Proper completion
   If a marking covers a stable state, it is a stable state.

How to check 3 components?

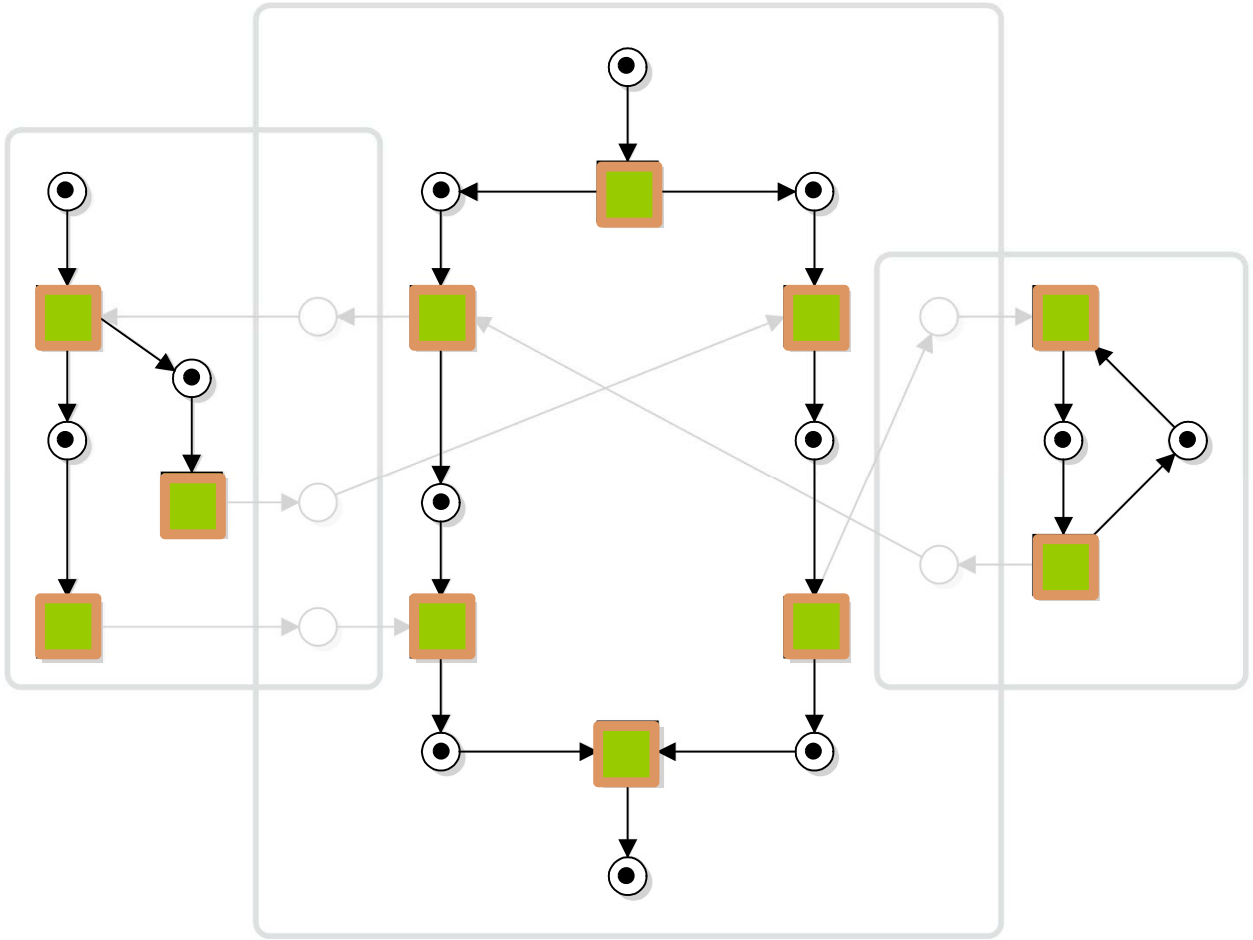


Utrecht University

How to check 3 components?

Correctness:

1. Weakly terminating
   Always possible to reach a stable state
2. Proper completion
   If a marking covers a stable state, it is a stable state.

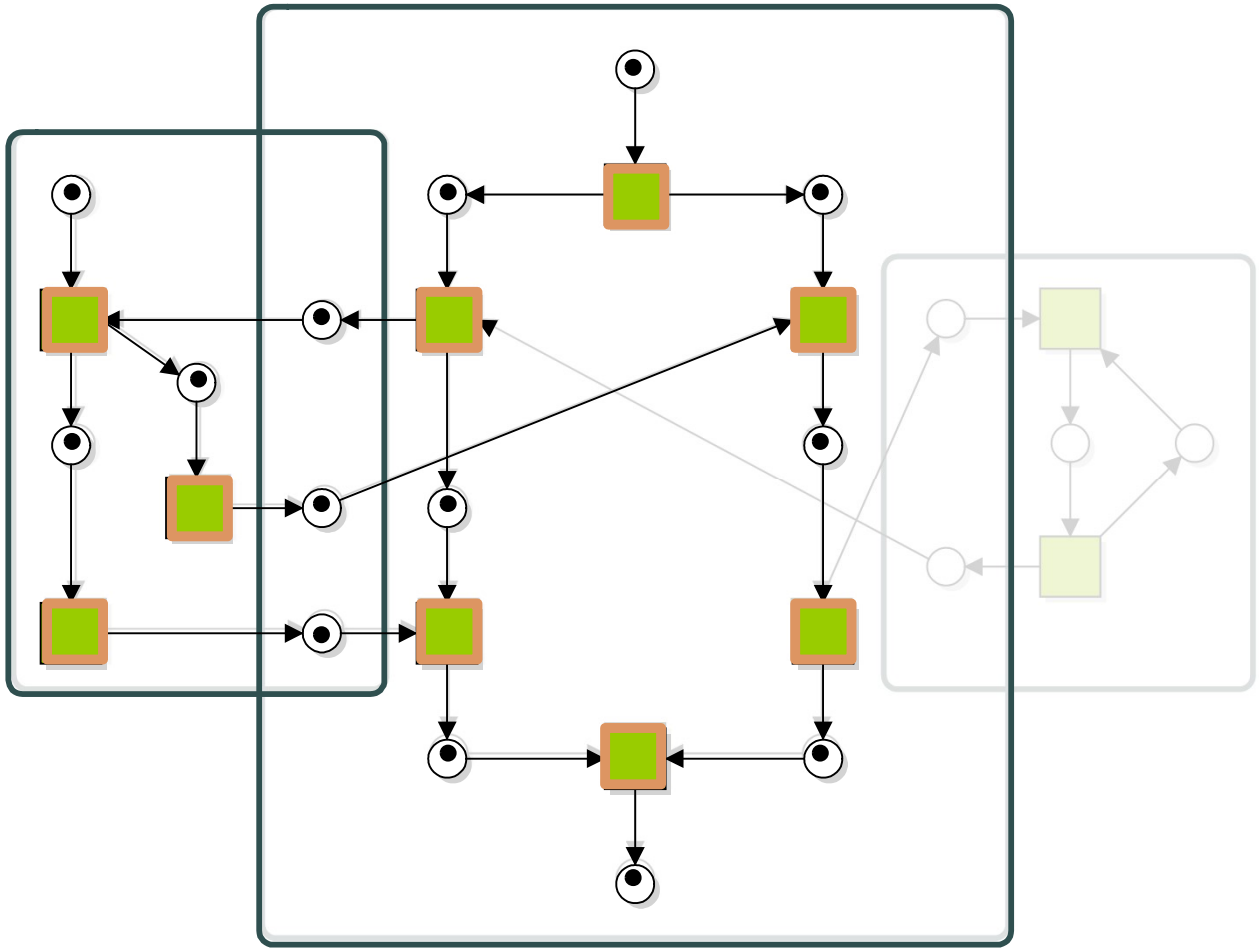


Utrecht University

How to check 3 components?

Correctness:

1. Weakly terminating
   Always possible to reach a stable state
2. Proper completion
   If a marking covers a stable state, it is a stable state.

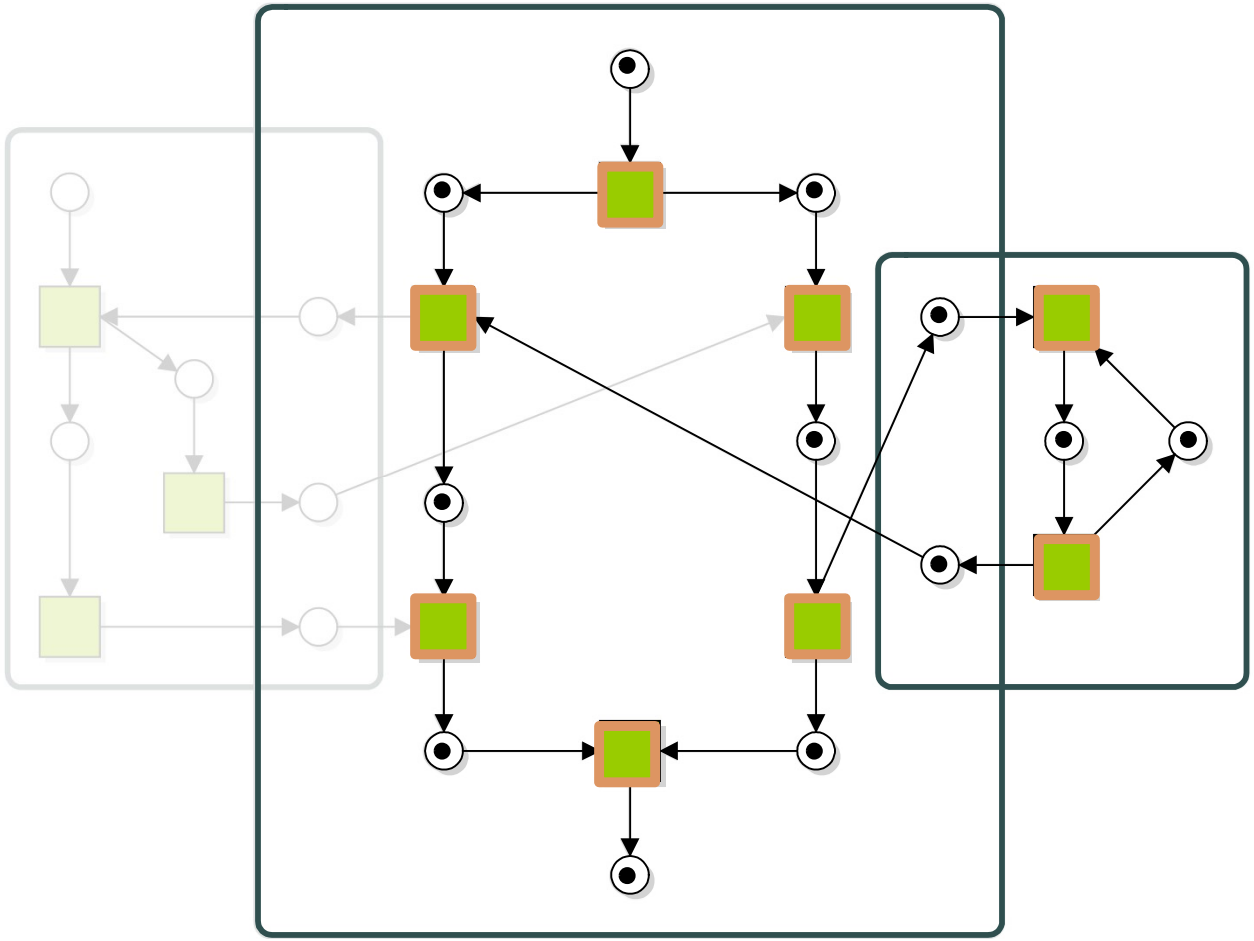


Utrecht University

How to check 3 components?

Correctness:

1. Weakly terminating
   Always possible to reach a stable state
2. Proper completion
   If a marking covers a stable state, it is a stable state.

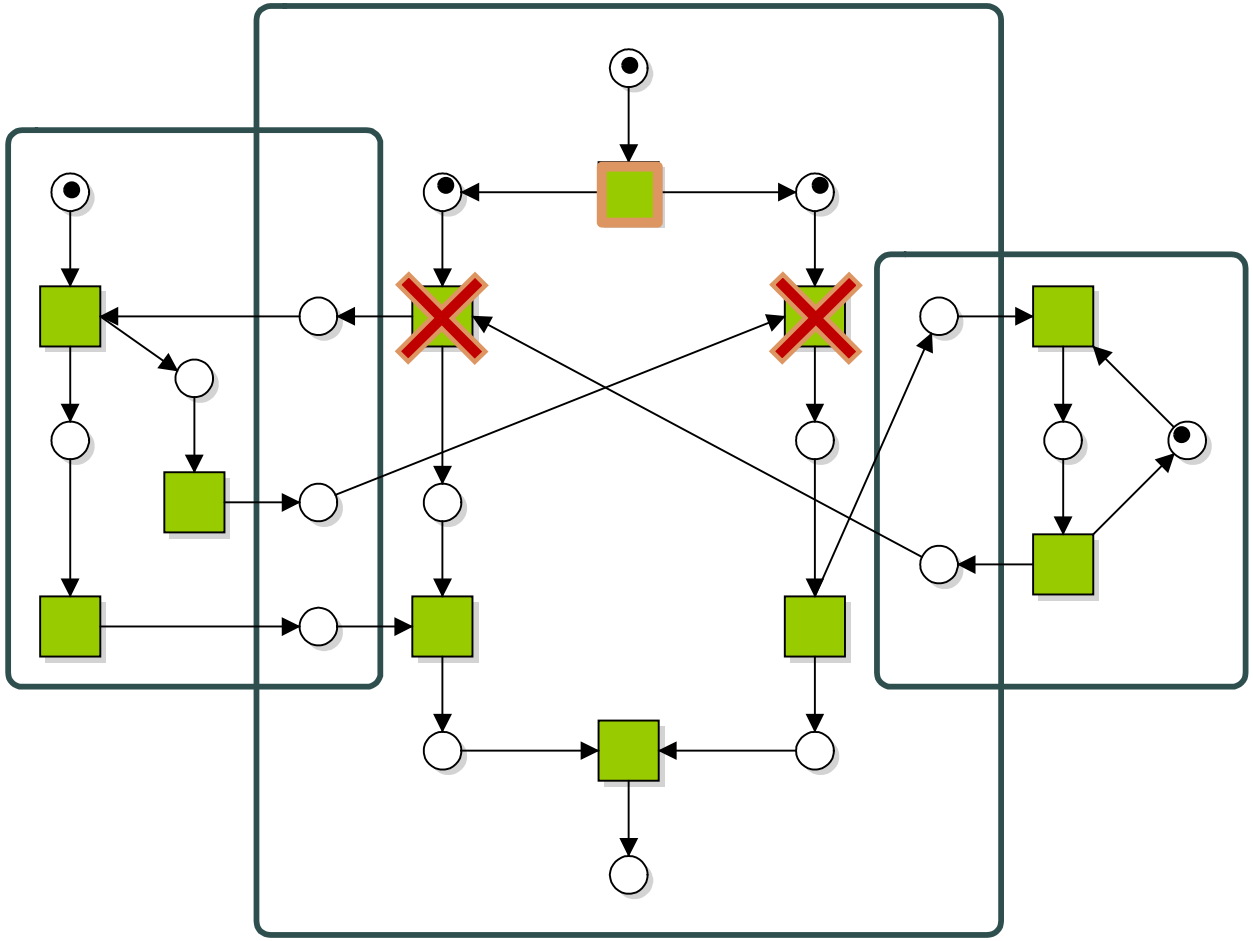


Utrecht University

Choreographies and their implementation

Correctness:

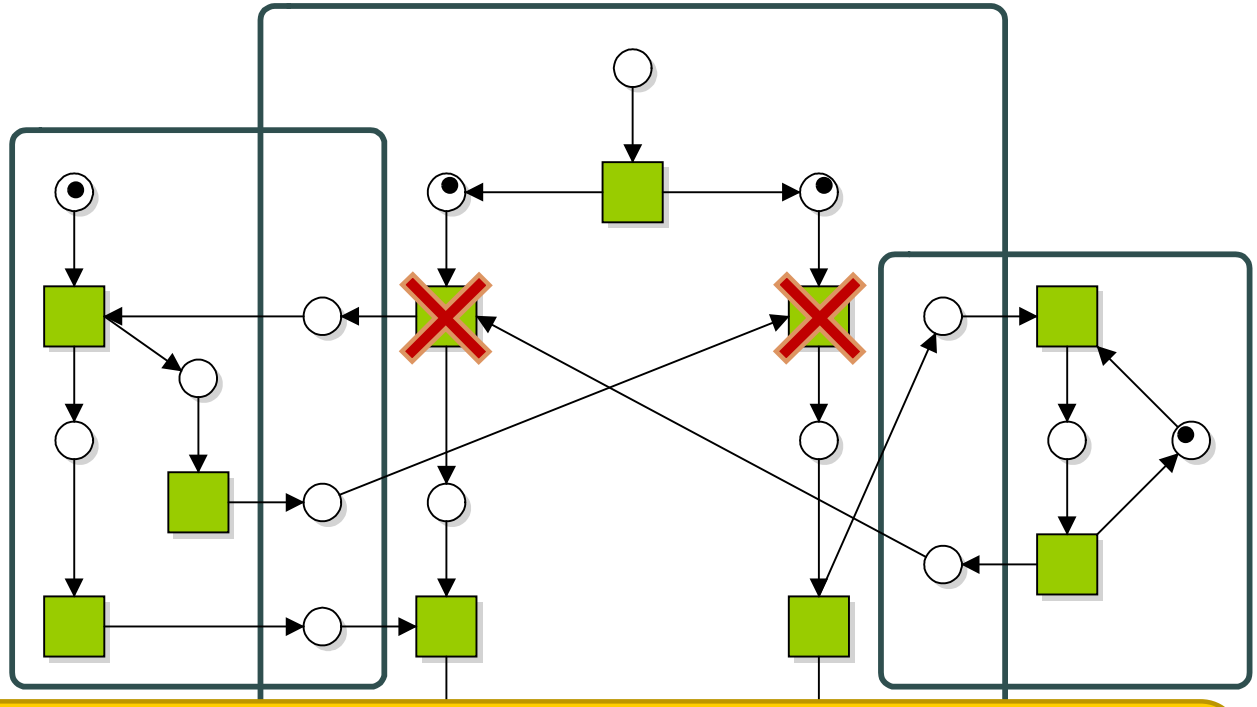
1. Weakly terminating
   Always possible to reach a stable state
2. Proper completion
   If a marking covers a stable state, it is a stable state.



Utrecht University

How to check 3 components?

Correctness:
1. Weakly terminating
   Always possible to reach a stable state
2. Proper completion
   If a marking covers a stable state, it is a stable state.



Correctness is not a sufficient condition to pairwise validate the component interactions

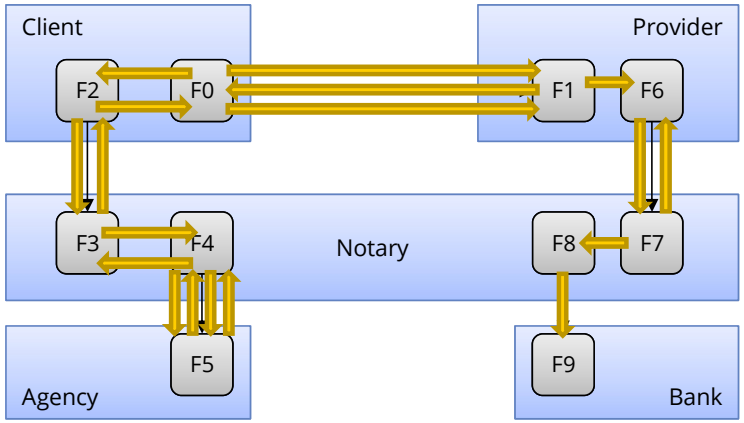
In general: this problem is **undecidable!**

Utrecht University

Choreographies: a different perspective on concurrency

Utrecht University

Logical model and scenarios



- Scenario is a sequence of function calls
- Formal definition:

  Given a logical model $(C, F, h, \rightarrow)$**,**
  a scenario is a partial order over the function calls,
  i.e., $\sigma \in (\rightarrow)^*$ **such that:**

  Functions can only start if being called before:

  $$\forall 1 < i < |\sigma| : \left(\exists 1 < j < i : \pi_3(\sigma(j)) = \pi_1(\sigma(i))\right)$$

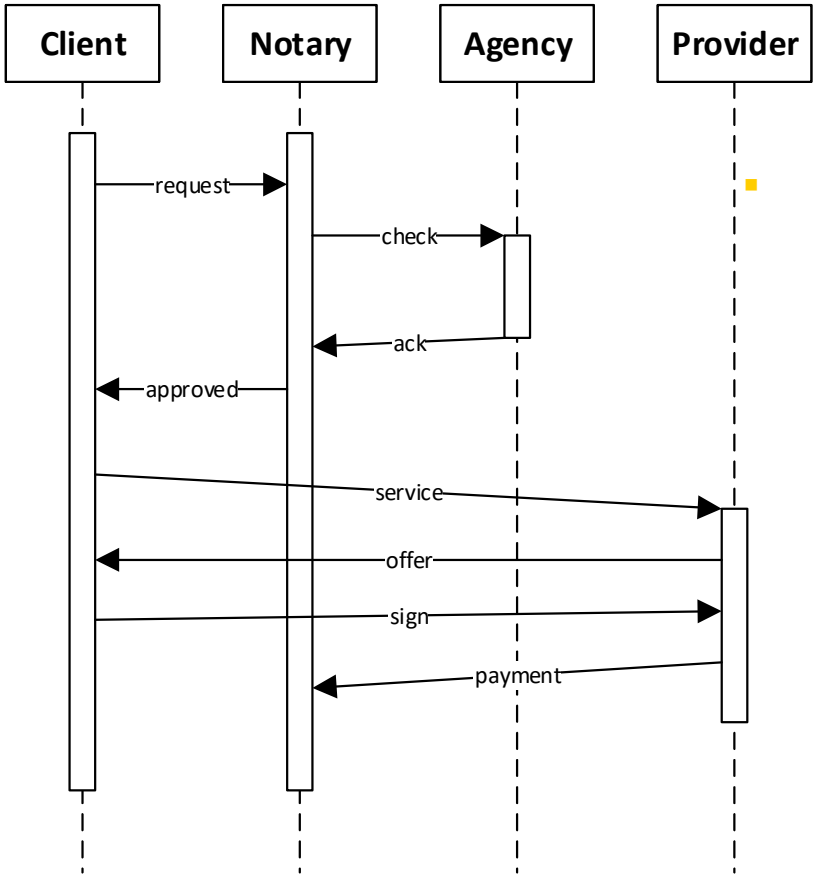
Key:
F0: Request service
F1: Handle service request
F2: Request approval
F3: Receive approval request
F4: Validate client
F5: Do credibility check
F6: Request payment
F7: Check payment request
F8: Send payment
F9: Make payment

Given a set A, a sequence of length $n \in \mathbb{N}$ is a function $\sigma : \{1..n\} \rightarrow A$

- We write $\sigma = \langle a_1, \ldots, a_n \rangle$ if $\sigma(i) = a_i$ for all $1 \leq i \leq n$.
- We denote its length by $|\sigma|$
- If $n = 0$, we call it the empty sequence, and denote it with ϵ
- The set of all finite sequences over A is denoted by A^*

Utrecht University

Sequence diagrams



Scenario is a sequence of function calls

- Formal definition:

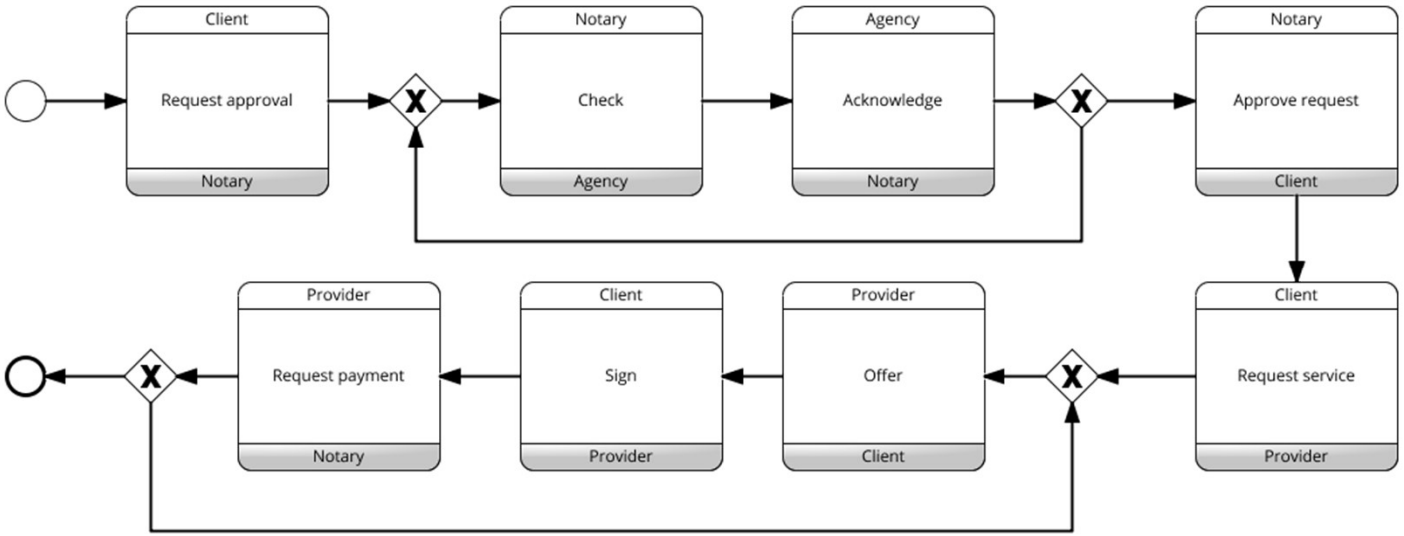
  Given a logical model $(C, F, h, \rightarrow)$**,**
  a scenario is a partial order over the function calls,
  i.e., $\sigma \in (\rightarrow)^*$ **such that:**
  Functions can only start if being called before:
  $\forall 1 < i < |\sigma| : \left(\exists 1 < j < i : \pi_3(\sigma(j)) = \pi_1(\sigma(i))\right)$

$\sigma =$ <(C,request, N) , (N,check, A) , (A,ack, N) , (N,approved, C) , (C, service, P) , (P, offer, C), (C, sign, P), (P, payment, N) >

Utrecht University

A choreography

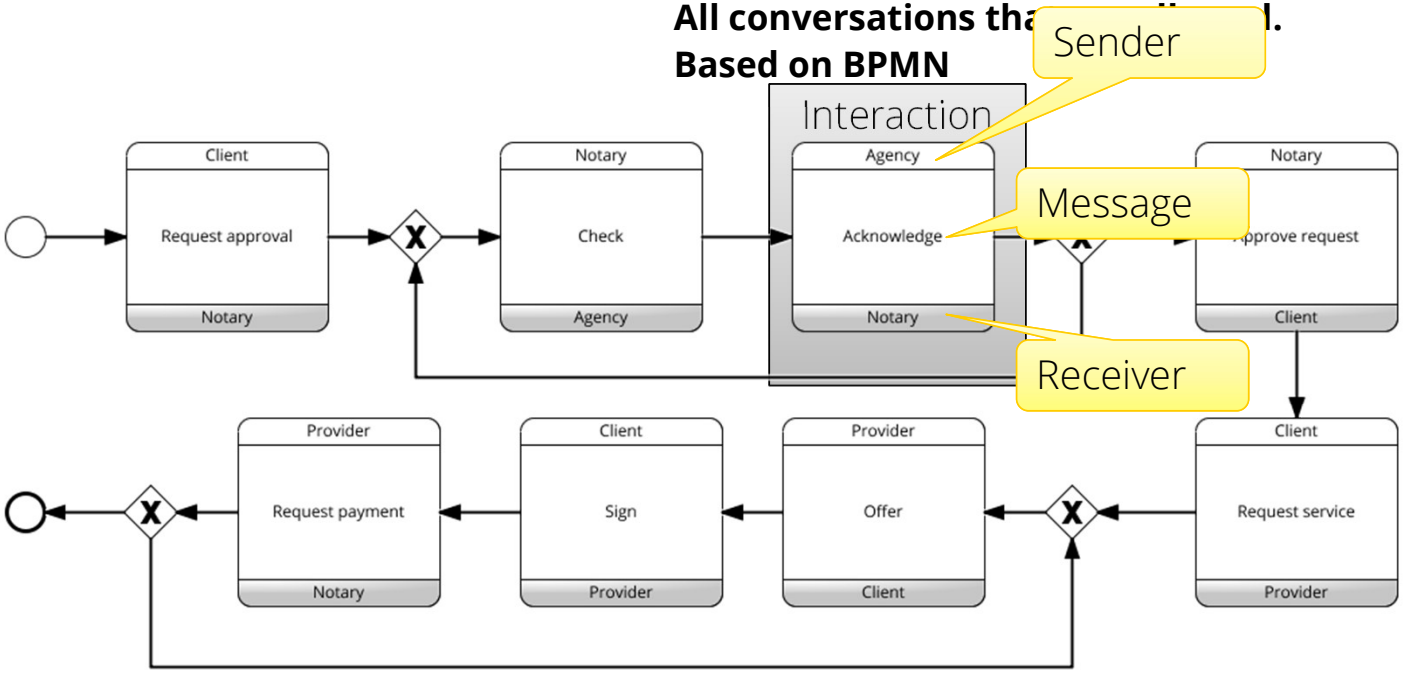
- Models conversations
  - **A conversation is a sequence of message exchanges**
- A Choreography:
  - **All conversations that are allowed.**
  - **Based on BPMN**



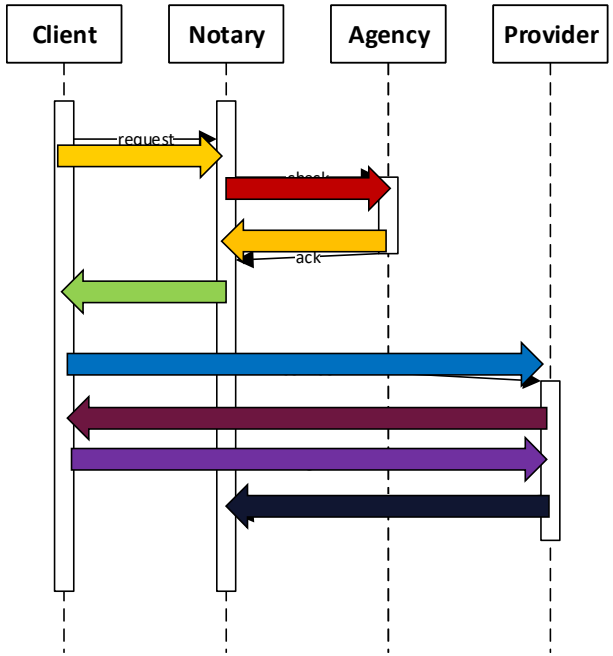
Utrecht University

A choreography

- Models conversations
  - **A conversation is a sequence of message exchanges**
- A Choreography:
  - **All conversations tha[illegible]l.**
  - **Based on BPMN**

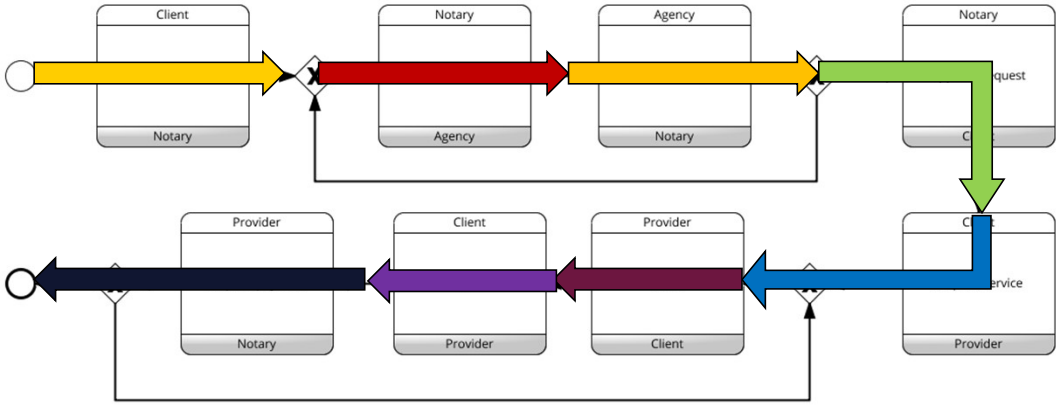


Utrecht University



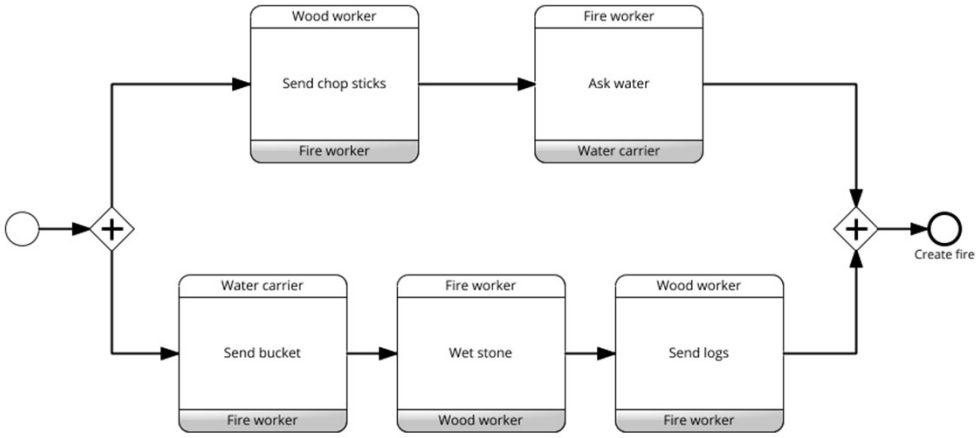
Choreographies and Message Sequence Charts

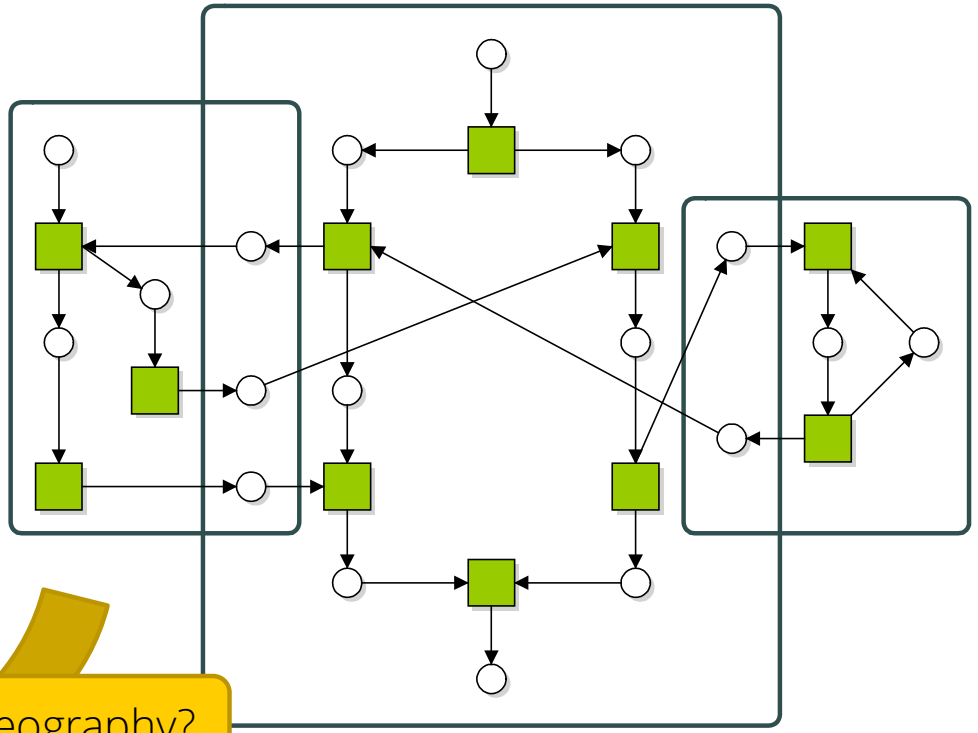
- A choreography describes all valid conversations



Utrecht University

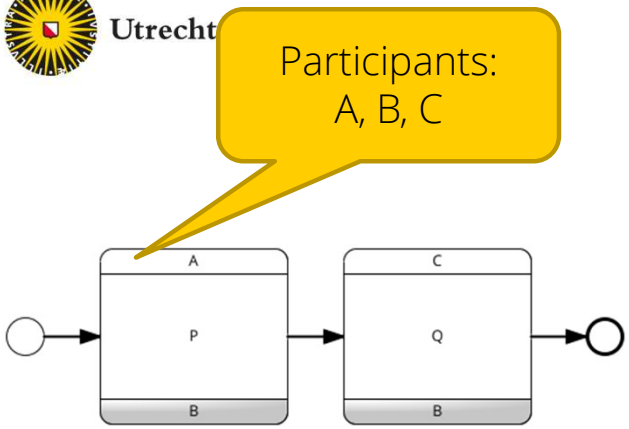
Back to our previous example...





Is this Petri net a realization of this choreography?

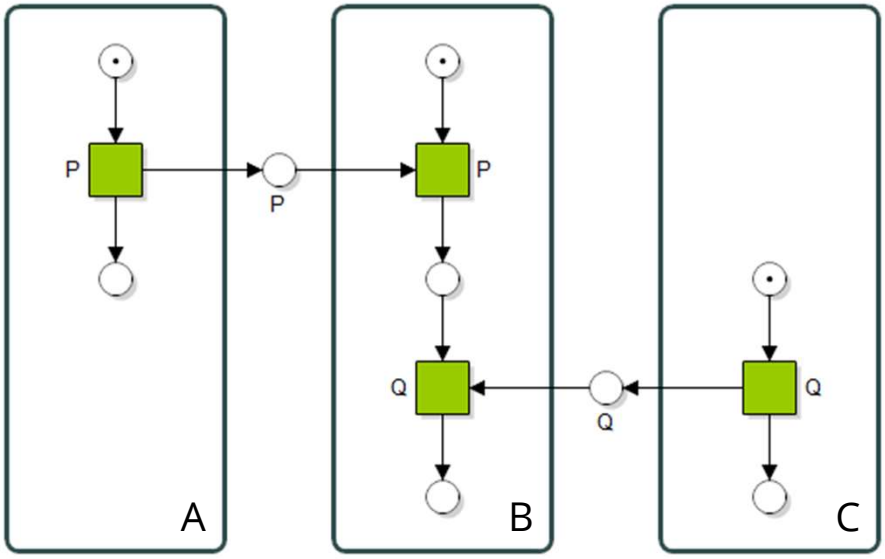
No! Actually: no choreography can be constructed with this PN as some realization!



Is every choreography "correct"? From choreography to Petri net

- Realizable:

  There exists an implementation "such that the set of conversations (possibly infinite!) the implementation supports is equal to the set of conversations in the choreography (possibly infinite!).



1: translate model to a Petri net "a la BPMN"
2: replicate net for each participant
3: create a place for each message
4: connect the participants to the messages

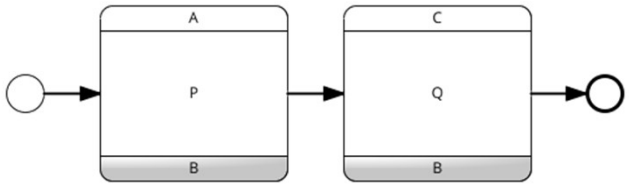
Sending participant produces token
Receiving participant consumes token

5: simplify the model using Murata rules

(Murata, 1989)

Utrecht University

Is every choreography “correct”?



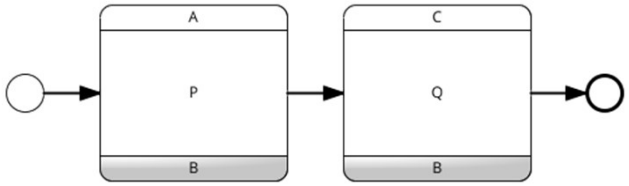
- Realizable:

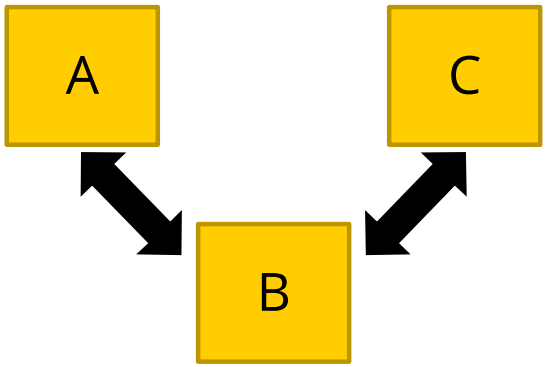
  There exists an implementation “such that the set of conversations (possibly infinite!) the implementation supports is equal to the set of conversations in the choreography (possibly infinite!).

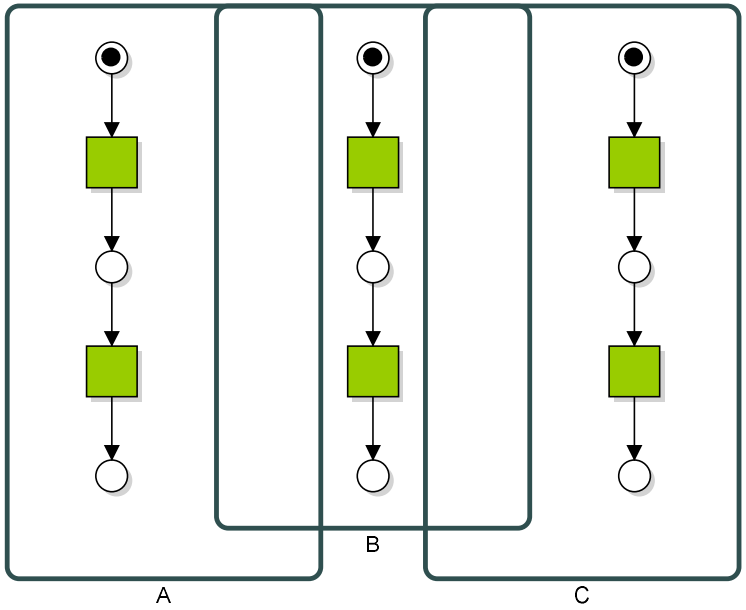
There may be many implementations that realize the same choreography!

Utrecht University

Is every choreography “correct”?

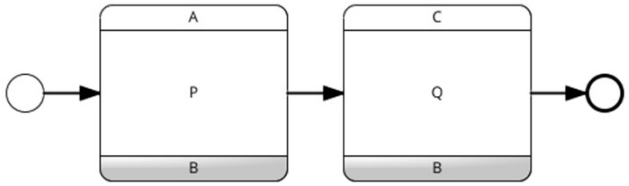


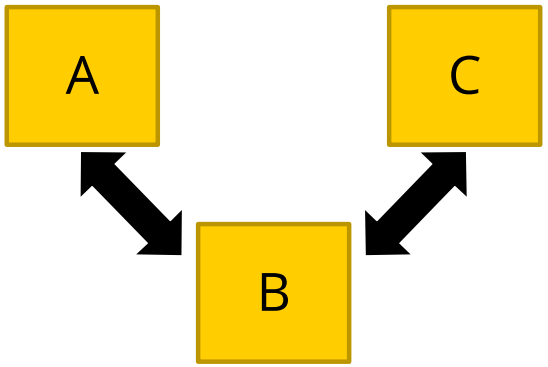


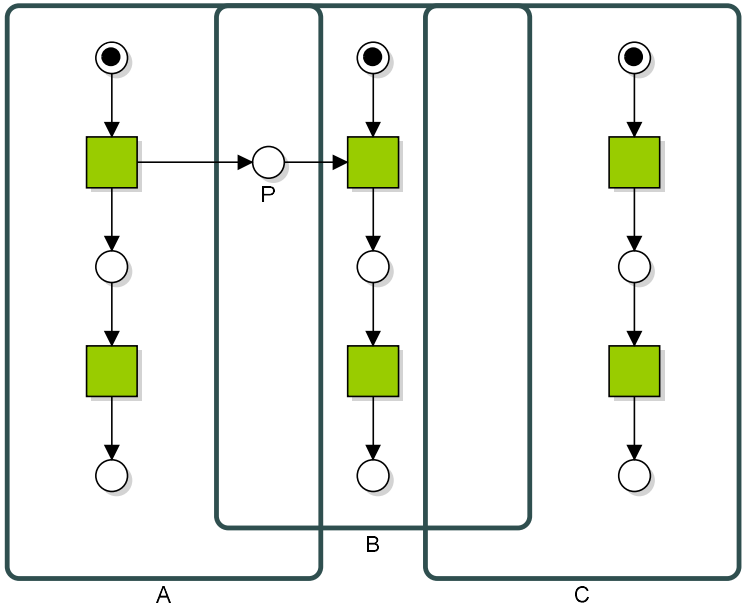


Utrecht University

Is every choreography “correct”?



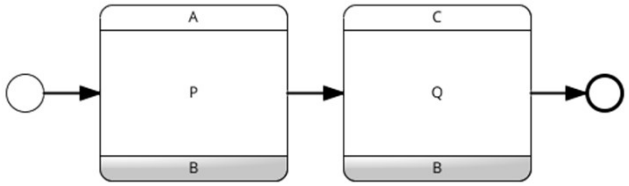


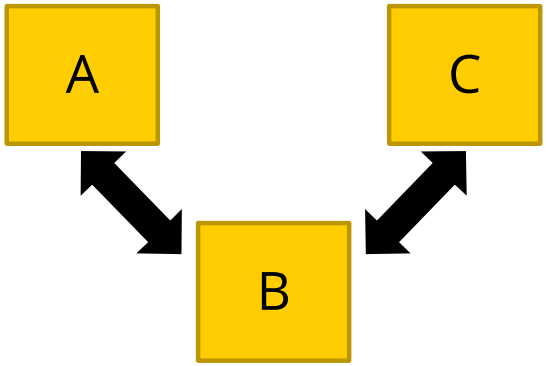


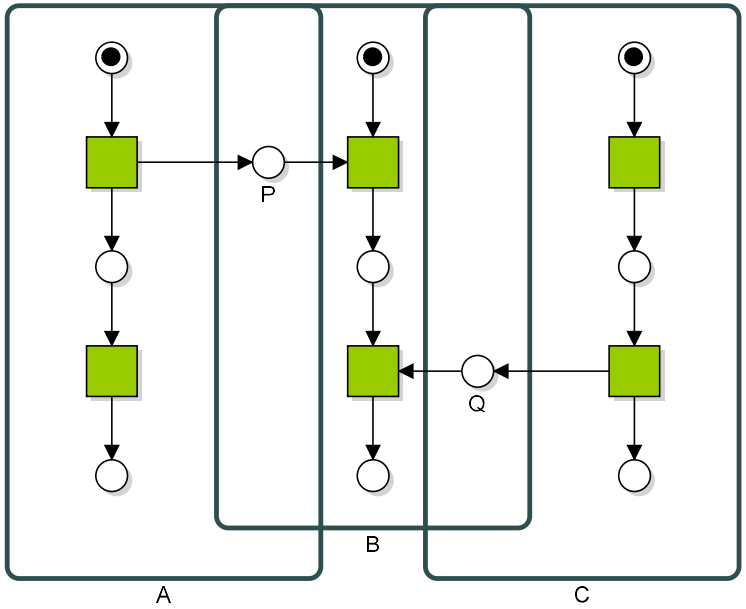
JMvdW2

Utrecht University

Is every choreography “correct”?





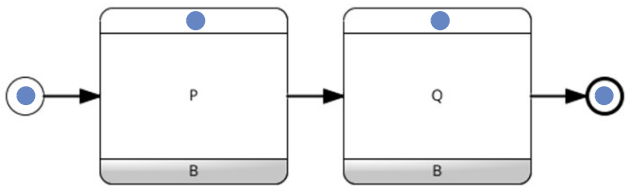


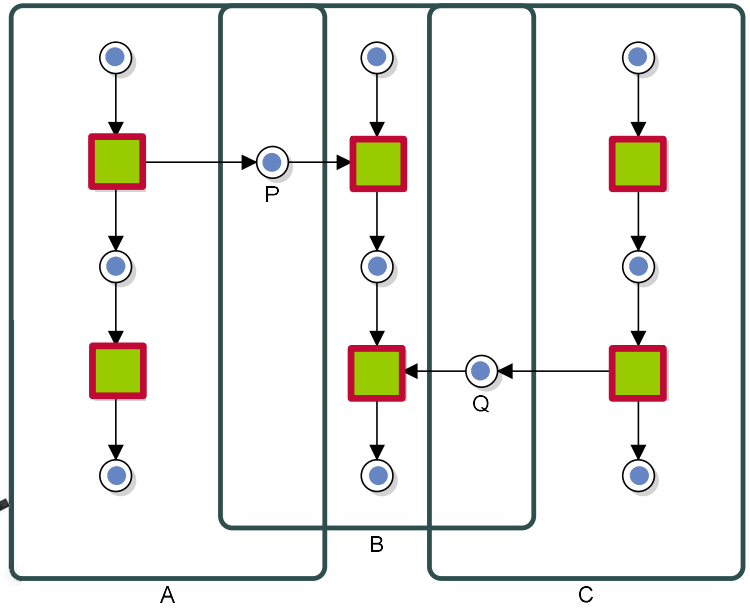
Slide 46

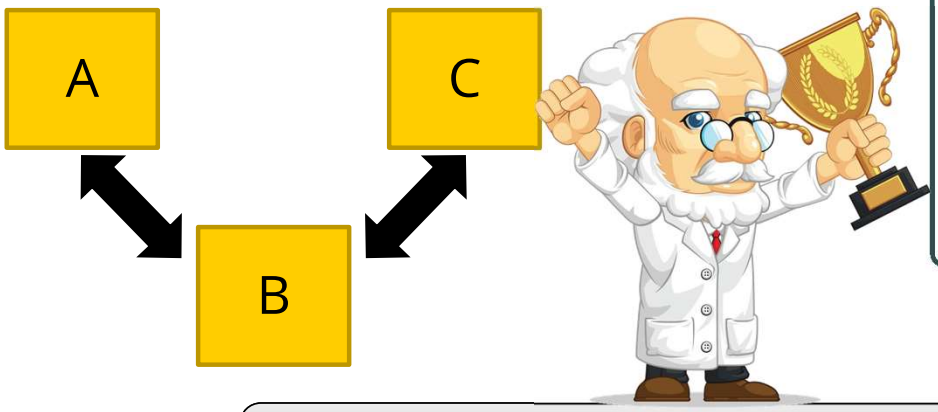
JMvdW2 Toevoegen blauwe en rode run (realiseerbaar en niet realiseerbaar)
Jan Martijn van der Werf; 21-10-2021

Utrecht University

Is every choreography “correct”?



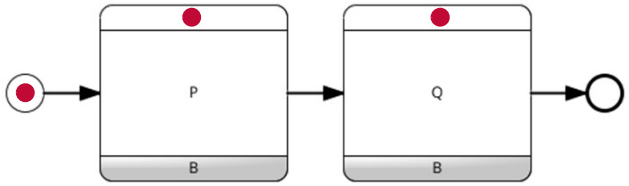


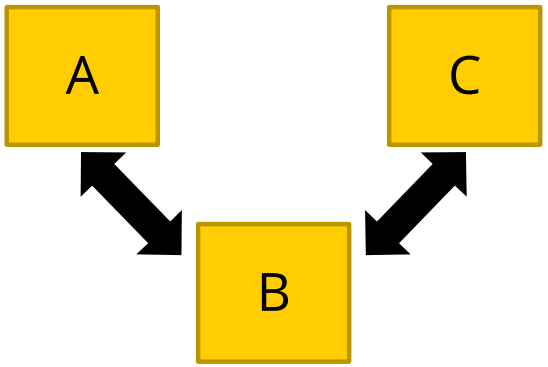


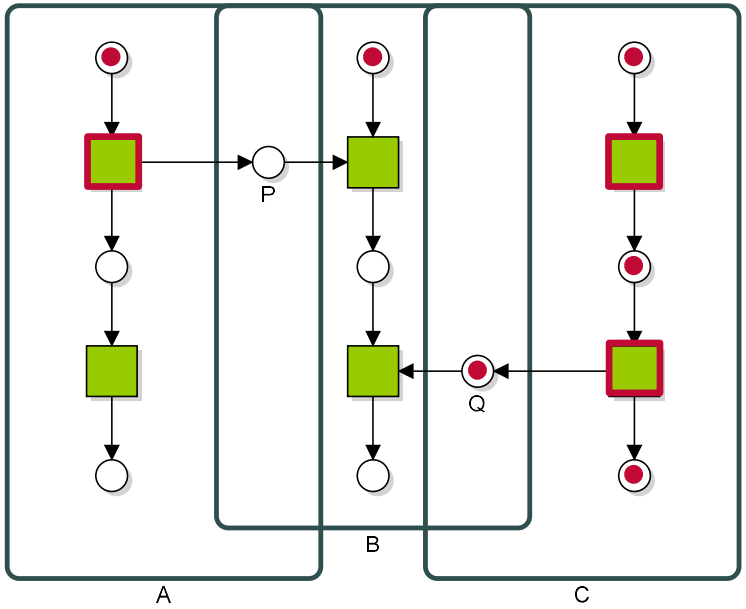
This is a run of the choreography realized by the system

Utrecht University

Is every choreography “correct”?

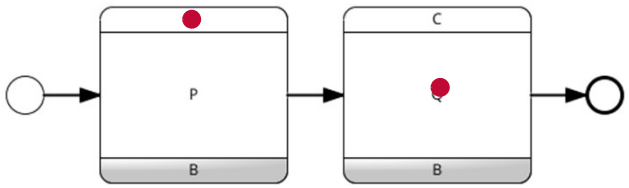


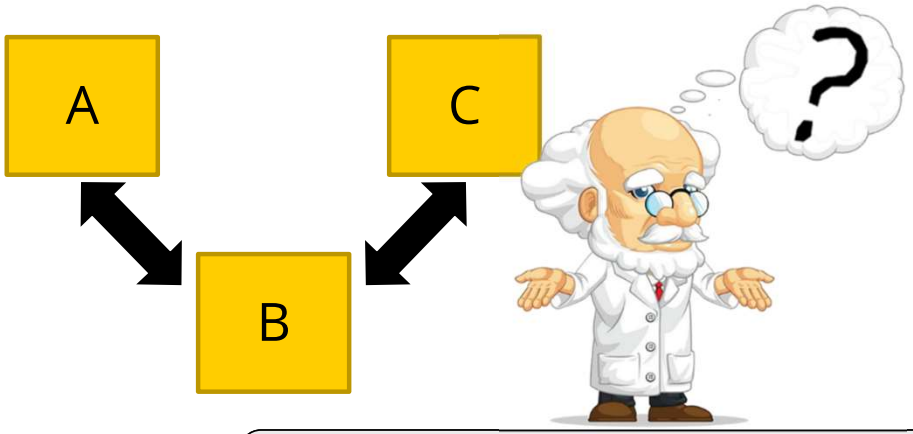


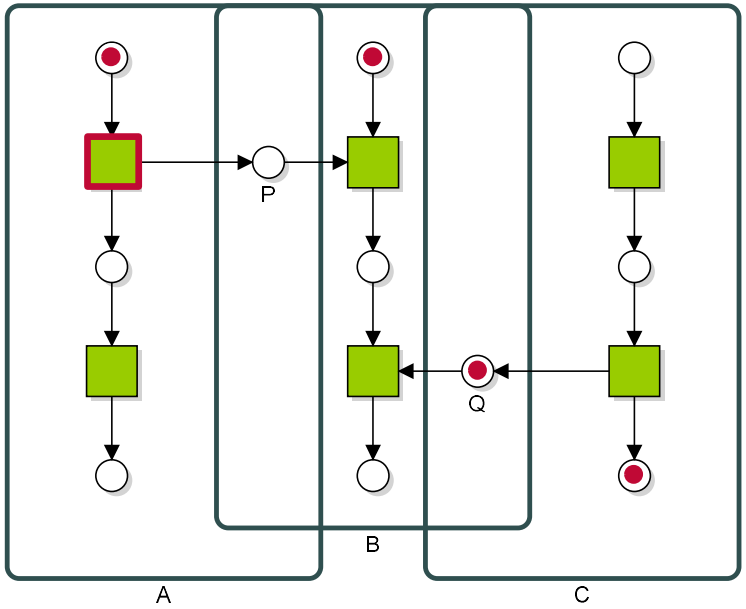


Utrecht University

Is every choreography “correct”?



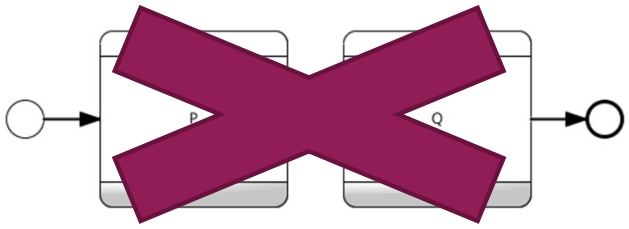


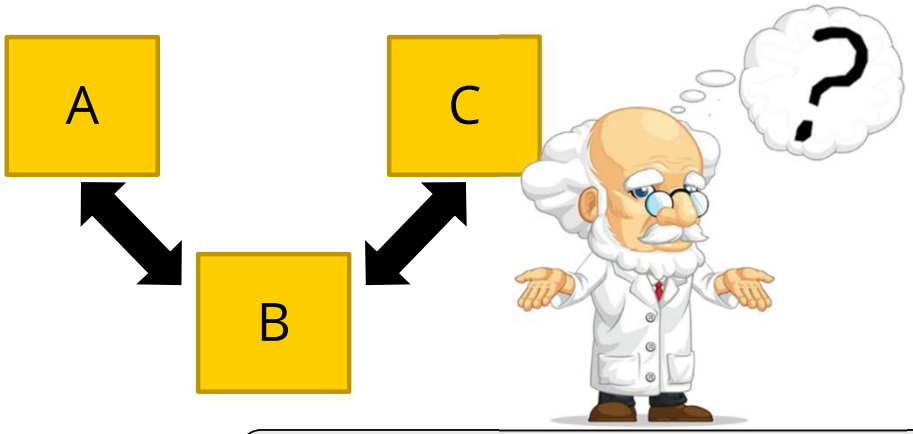


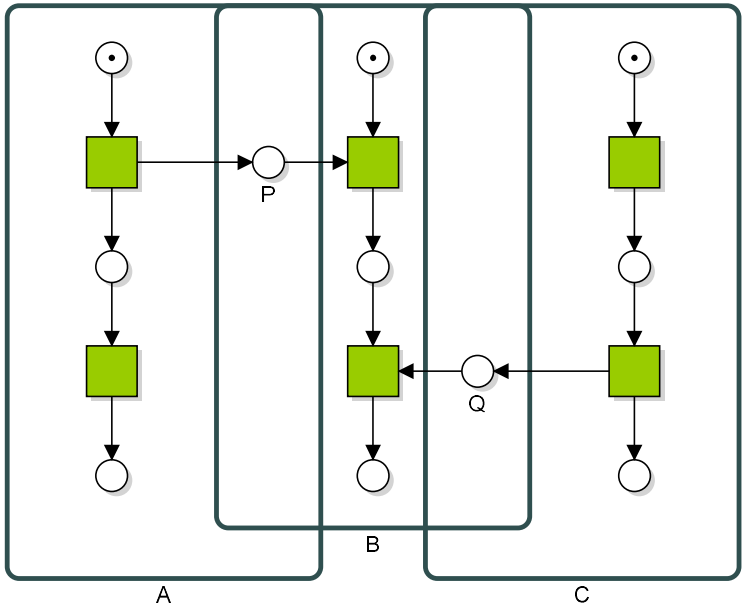
How does C know whether A sent its message?

Utrecht University

Is every choreography “correct”?



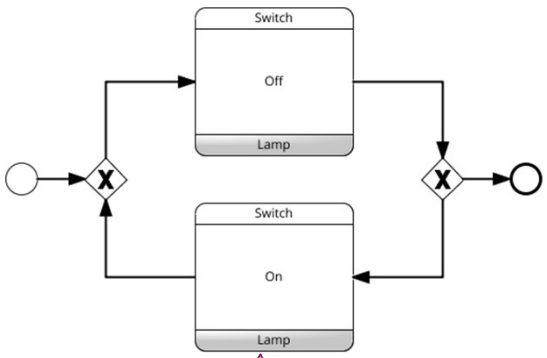




How does C know whether A sent its message?

Utrecht University

Rules on choreographies
A sufficient condition



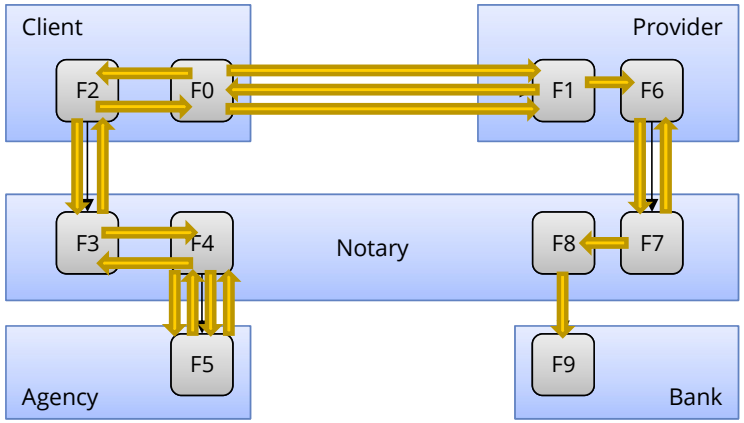
Loop, but only 1 party sends

A choreography is correct if **all** of the following hold:

1. Only 2 parties involved
2. No parallelism! (sorry…)
3. Each choice should be made by one party only
4. Both parties send at least one message
5. In any loop, both parties should at least send one message

Utrecht University

Define a Choreography for F0 and F1

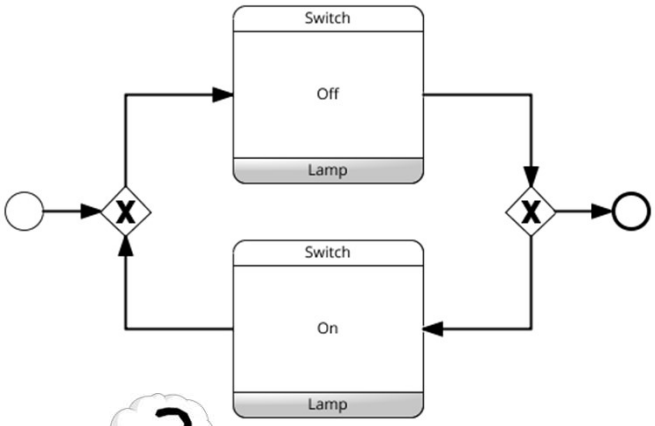


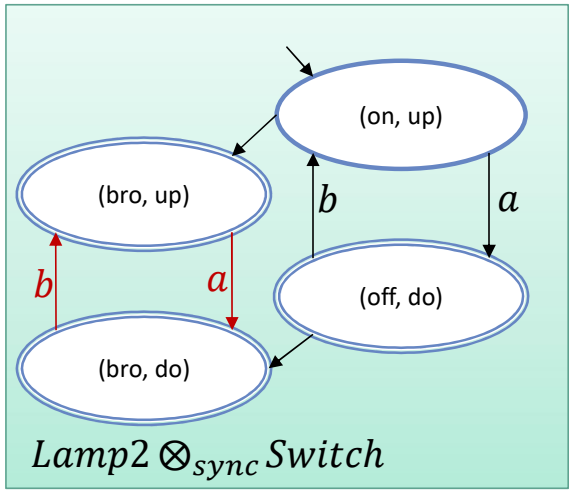
- The client asks for permission. Upon receiving the permission, (s)he asks for a service of the provider, and either accepts or denies the offer received by the provider. Upon accepting, the provider offers the service, and regularly sends an invoice, which needs to be signed and returned by the client. Once denied, either (s)he stops, or (s)he asks for a better offer.
  In case the client does not receive permission, (s)he tries again.

Key:
F0: Request service
F1: Handle service request
F2: Request approval
F3: Receive approval request
F4: Validate client
F5: Do credibility check
F6: Request payment
F7: Check payment request
F8: Send payment
F9: Make payment

Utrecht University

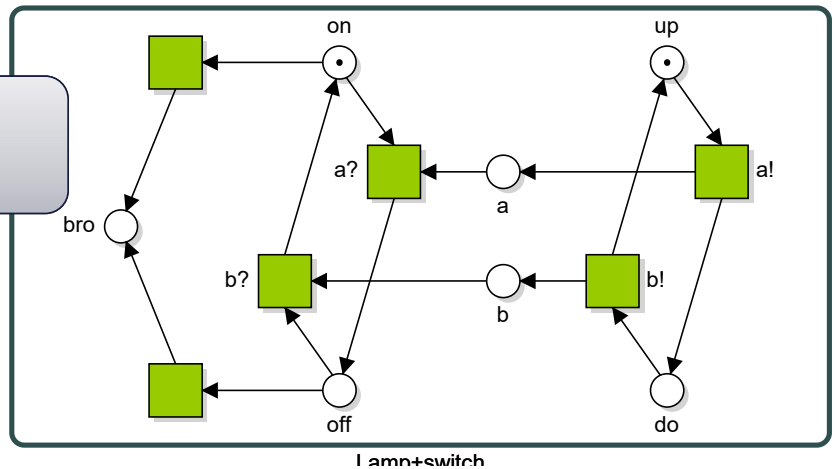
Realizability: asynchronous vs. synchronous





Which semantics do you choose? Why?

Realizable:
Synchronous : **yes**
Asynchronous : **no**



Utrecht University

Towards a construction method…

Research in progress

Utrecht University

How to model the containers?

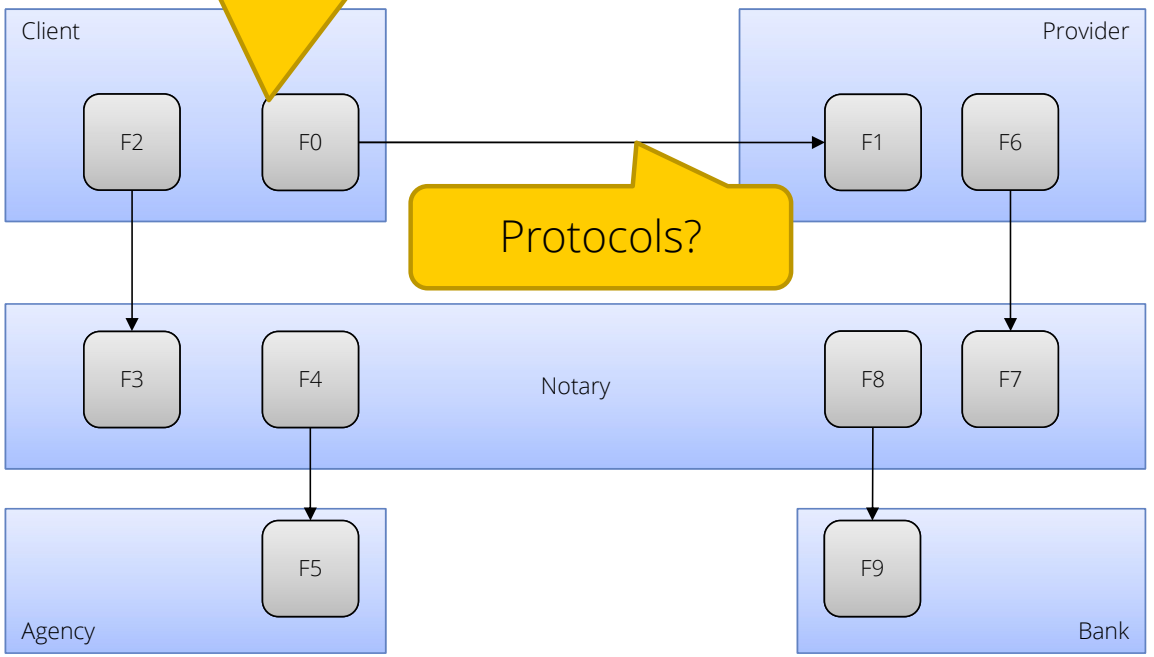
Allowed orders?

Internal dependencies?

Types of dependencies:
- Exclusion / choice
- Parallelism
- Loops

State diagrams?
- No parallelism

Petri nets?
- Feature == transition
- Places denote dependencies



Key:

F0: Request service
F1: Handle service request
F2: Request approval
F3: Receive approval request
F4: Validate client
F5: Do credibility check
F6: Request payment
F7: Check payment request
F8: Send payment
F9: Make payment

Utrecht University

How to model the containers?

Types of dependencies:
- Exclusion / choice
- Parallelism
- Loops

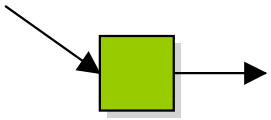
State diagrams?
- No parallelism

Petri nets?
- Feature == transition
- Places denote dependencies

Key:

F0: Request service
F1: Handle service request
F2: Request approval
F3: Receive approval request
F4: Validate client
F5: Do credibility check
F6: Request payment
F7: Check payment request
F8: Send payment
F9: Make payment

Function:



Dependency

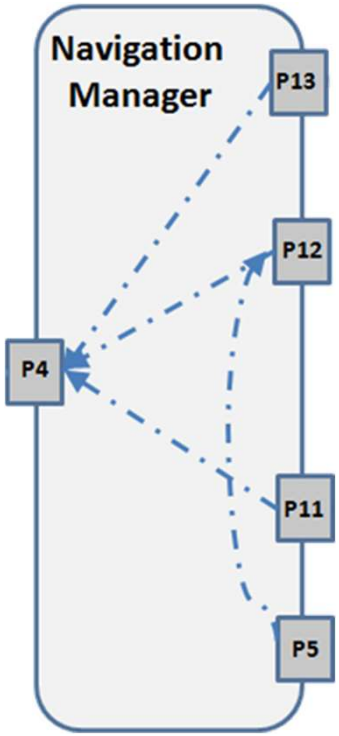
Via places

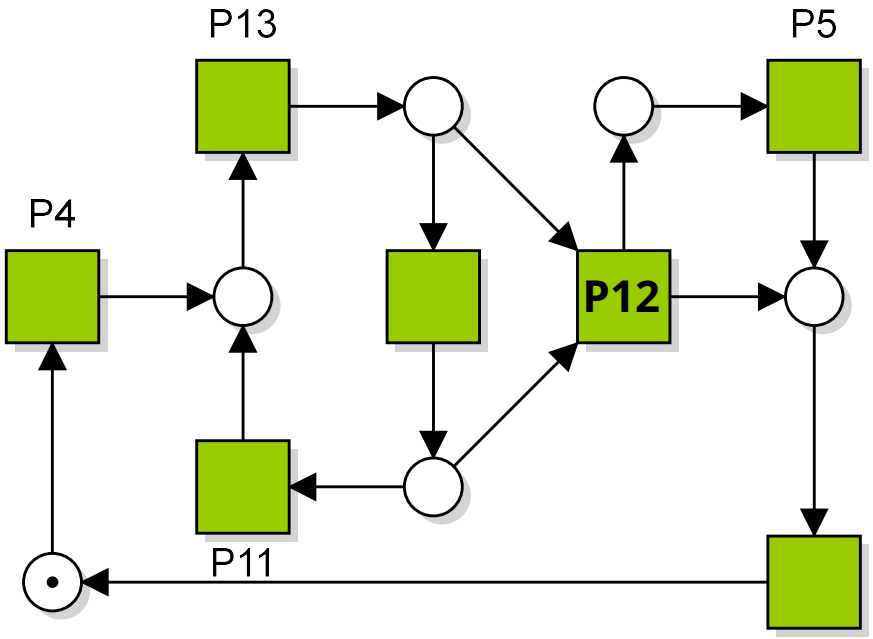
Allowed order of functions?

Token game in Petri nets

Utrecht University

When is an internal specification correct?

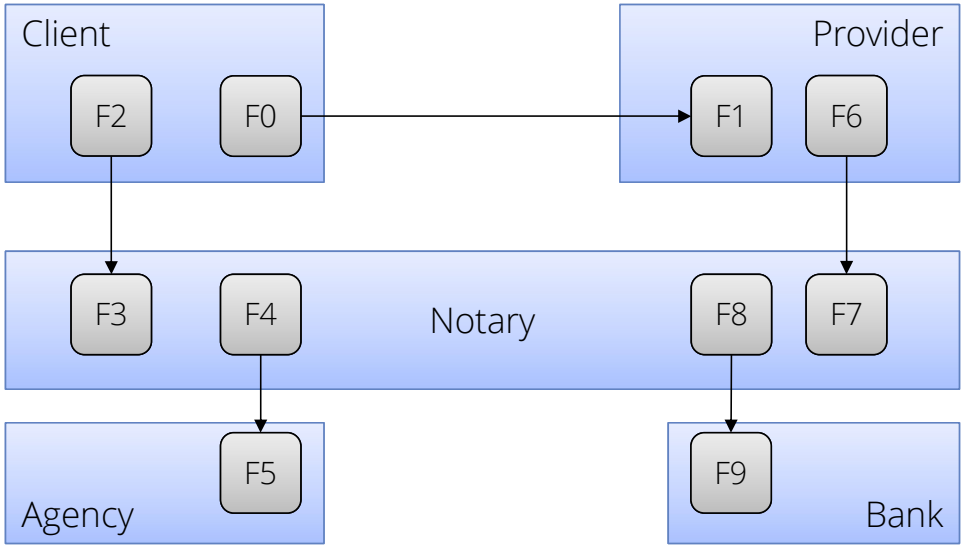




Every function should always eventually be enabled?

Utrecht University

How to model the containers?



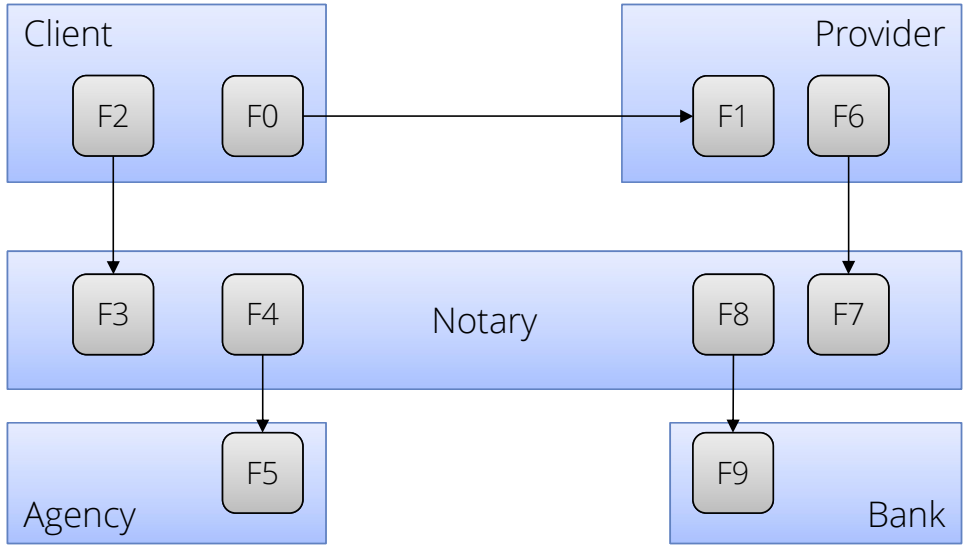
Key:
F0: Request service
F1: Handle service request
F2: Request approval
F3: Receive approval request
F4: Validate client
F5: Do credibility check
F6: Request payment
F7: Check payment request
F8: Send payment
F9: Make payment

Questions:

- ctions depend on another function?
- ctions may be called on their own?

Utrecht University

How to model the containers?

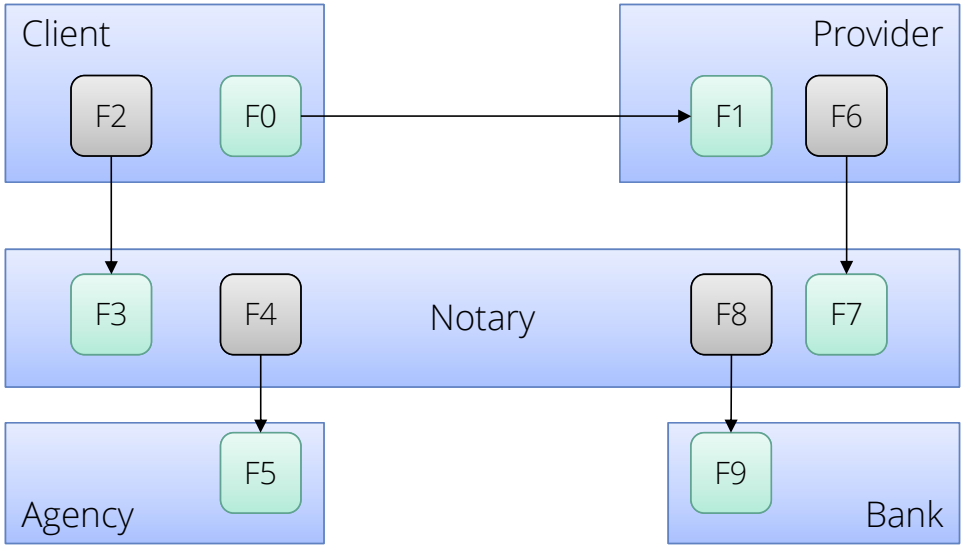


Questions:
1. Which functions depend on another function?
2. Which functions may be called on their own?

A function F is **independent** iff there are no functions that depend on F within the same container

Utrecht University

How to model the containers?

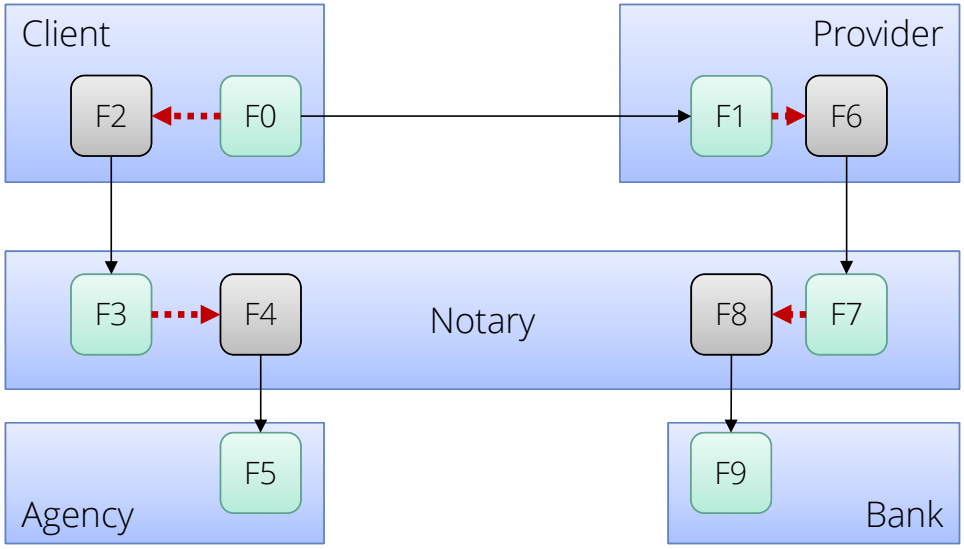


Questions:
1. Which functions depend on another function?
2. Which functions may be called on their own?

A function F is **independent** iff there are no functions that depend on F within the same container

Utrecht University

How to model the containers?

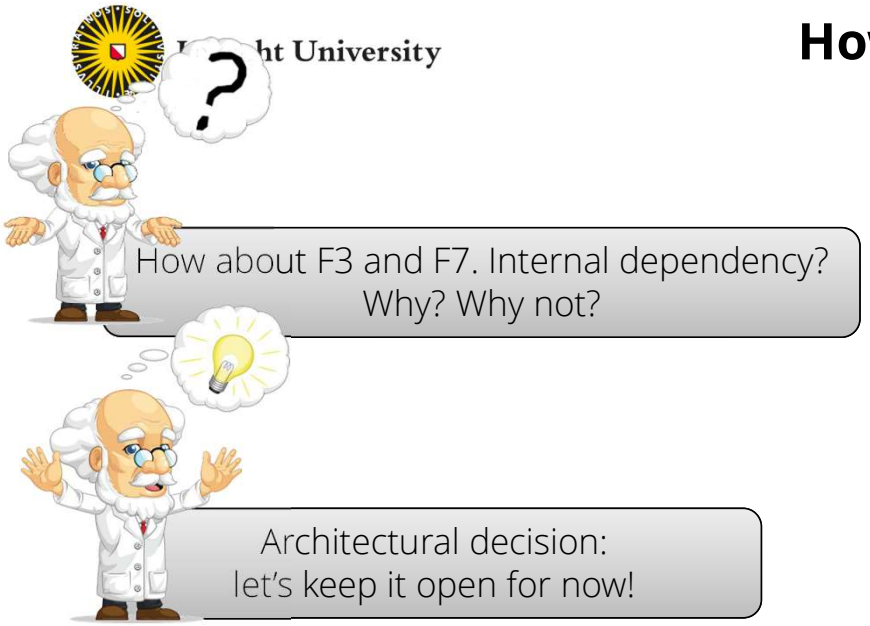


Two types of arrows in the logical model:

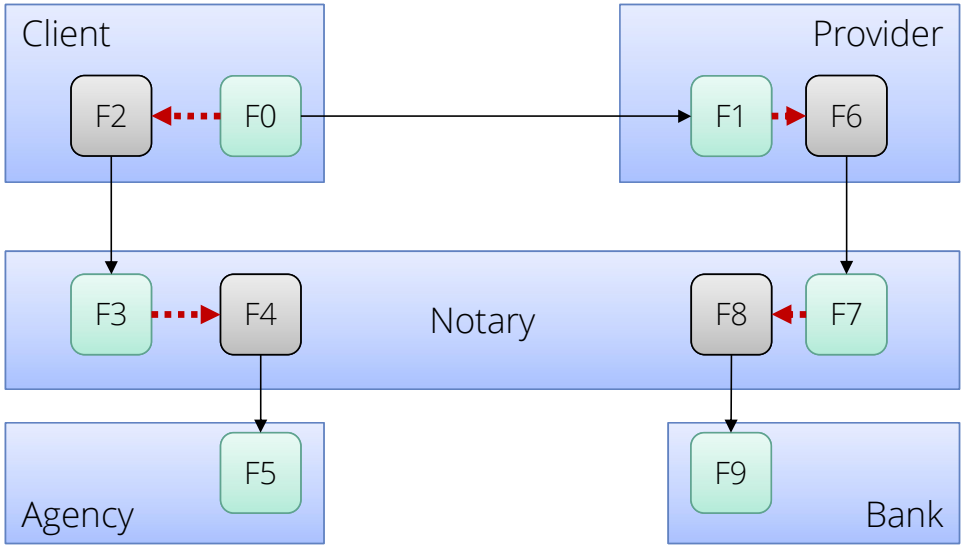
1. Function calls: between functions of **different** containers

2. Dependencies: "requires relation": A → B iff "A requires B to complete its function"

We use a dashed arrow for the requires relation!



How to model the containers?

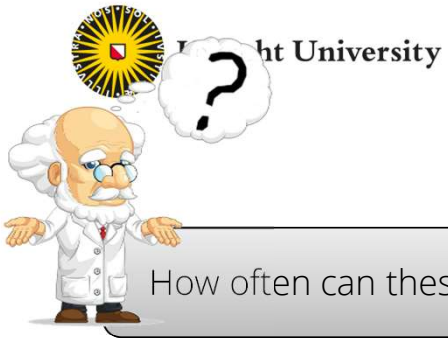


Two types of arrows in the logical model:

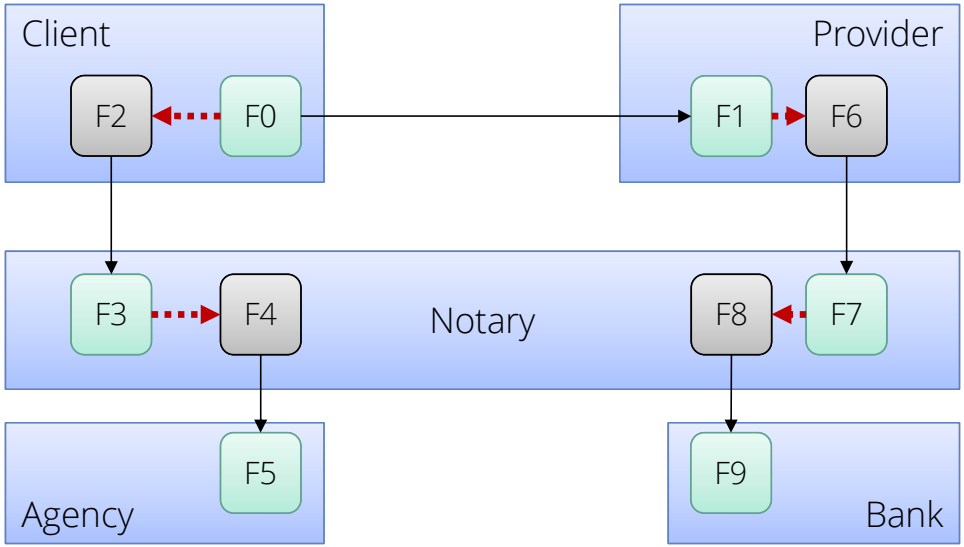
1. Function calls: between functions of **different** containers

2. Dependencies: "requires relation": A → B iff "A requires B to complete its function"

We use a dashed arrow for the requires relation!



How to model the containers?



Two types of arrows in the logical model:

1. Function calls: between functions of **different** containers

2. Dependencies: "requires relation": A → B iff "A requires B to complete its function"

We use a dashed arrow for the requires relation!

Utrecht University

Building a container net

- Each **independent** feature becomes a transition

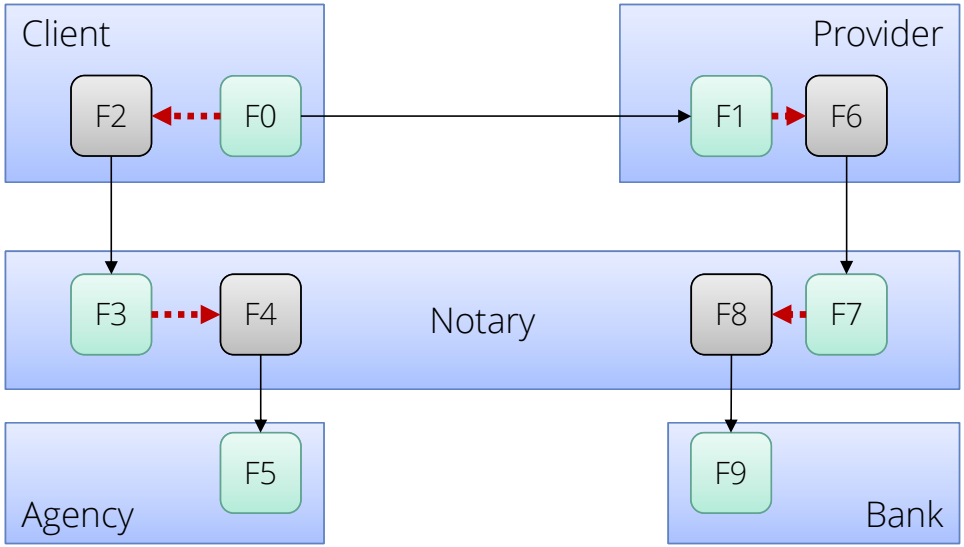
- Questions your net should answer:

  Is it always possible to "restart" the module?
  Can the feature only be called once?

Refinement rules aid in building **correct** Petri nets!

Utrecht University

Transformation of LM into Petri nets



Rules of the transformation:

1. Dependencies between independent functions as **container nets**.

Utrecht University

Building a container net

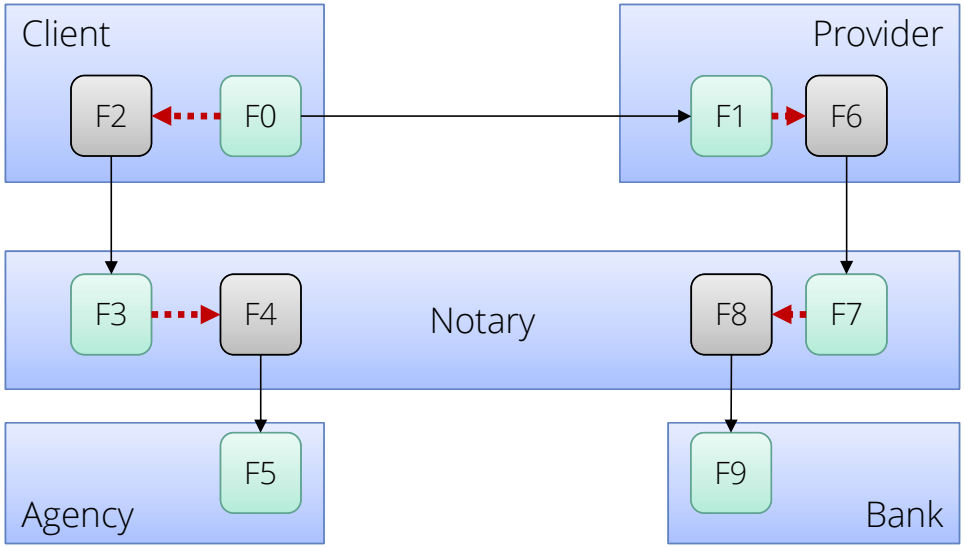
- Each **independent** function becomes a transition
- Each container net starts with the following net:

- Stick to the Self-loop transition rule!

  Even the Jackson rules may introduce dead- and livelocks!

Utrecht University

Transformation of LM into Petri nets

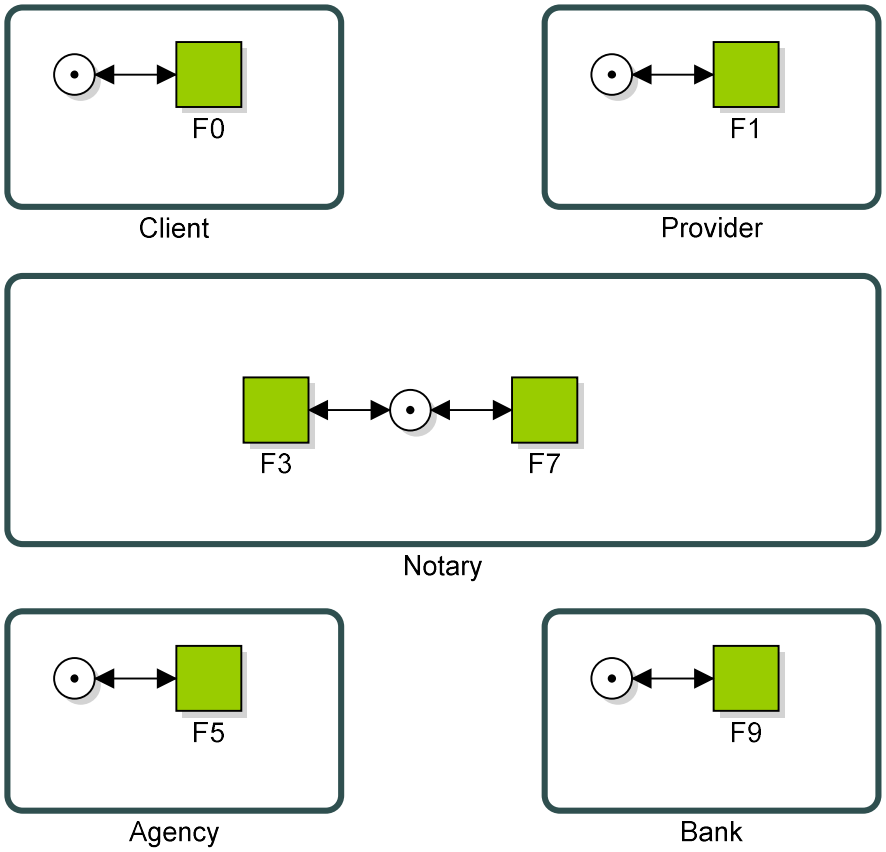


Rules of the transformation:

1. Dependencies between independent functions as **container nets**.

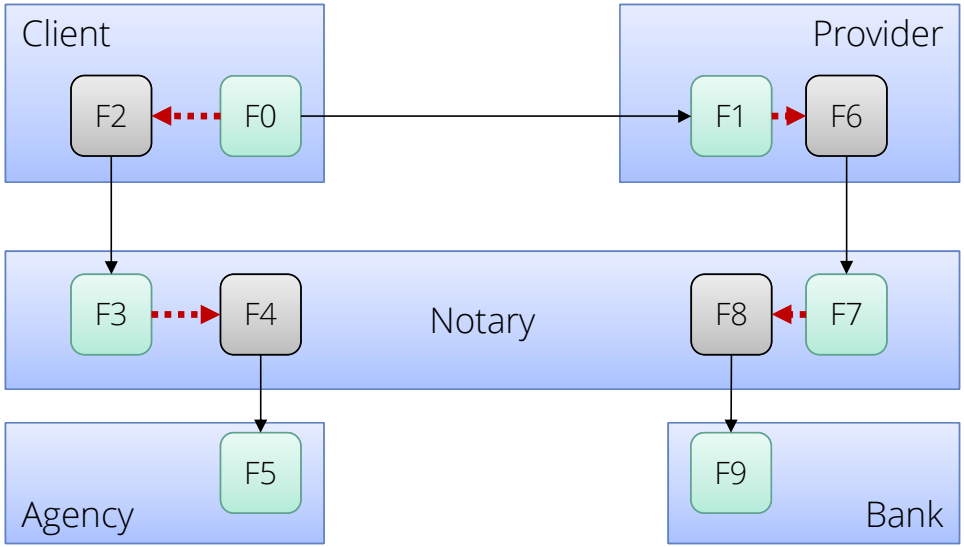
Utrecht University

Step 1: create a “Flower model” with all independent functions



Utrecht University

Transformation of LM into Petri nets

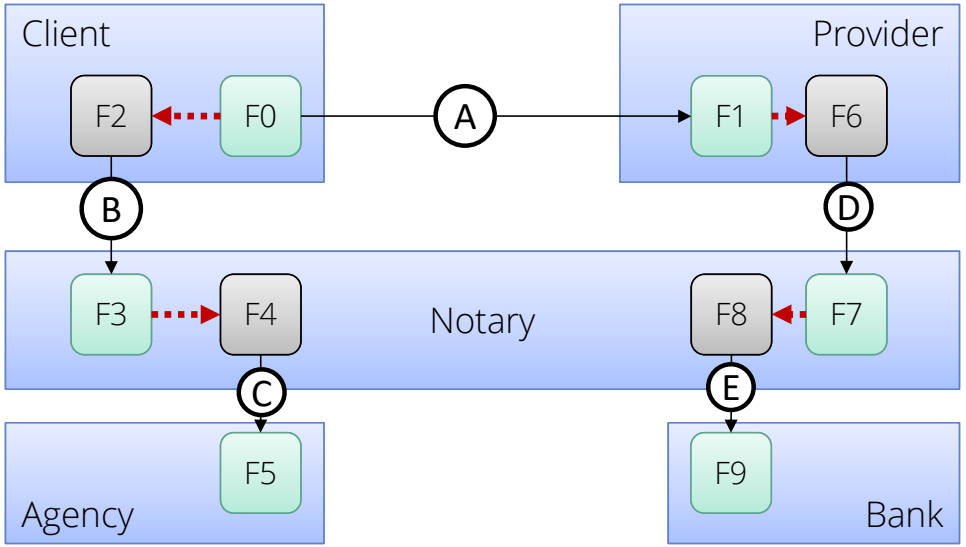


Rules of the transformation:

1. Dependencies between independent functions as **container nets**.
2. Function calls are **protocol choreographies**.

Utrecht University

Transformation of LM into Petri nets

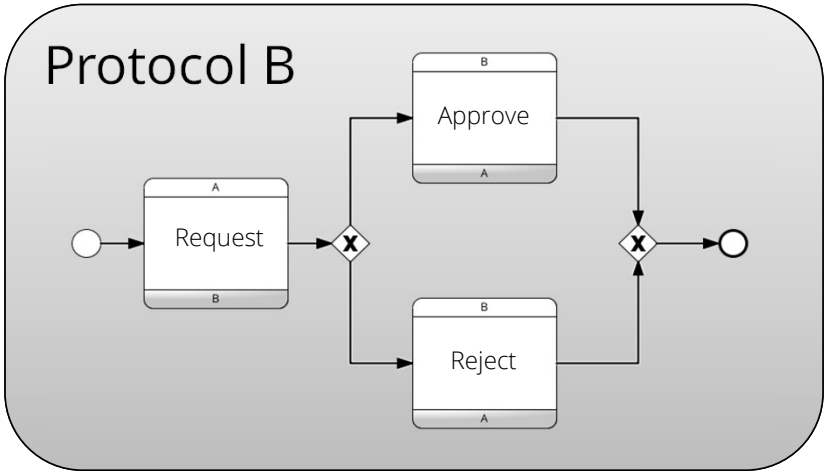


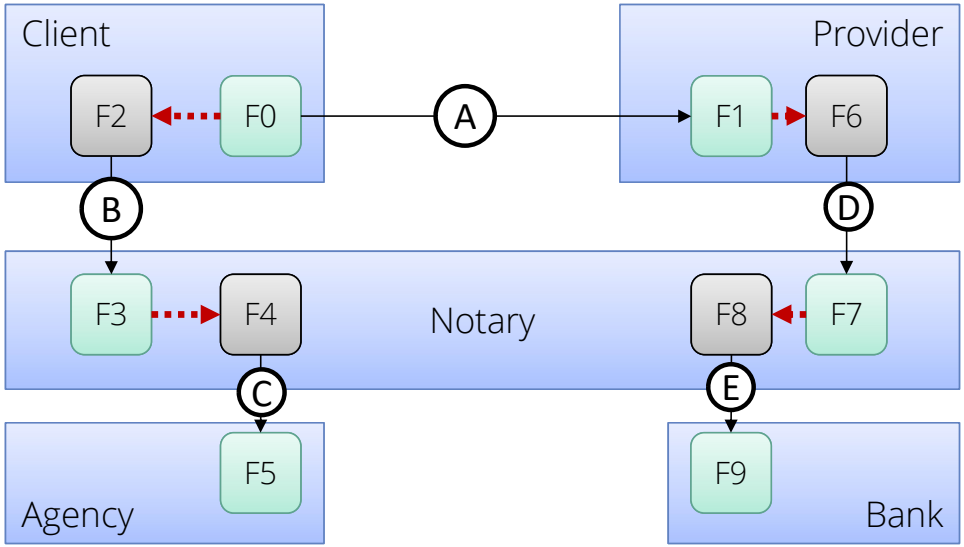
Rules of the transformation:

1. Dependencies between independent functions as **container nets**.
2. Function calls are **protocol choreographies**.

Utrecht University

Transformation of LM into Petri nets



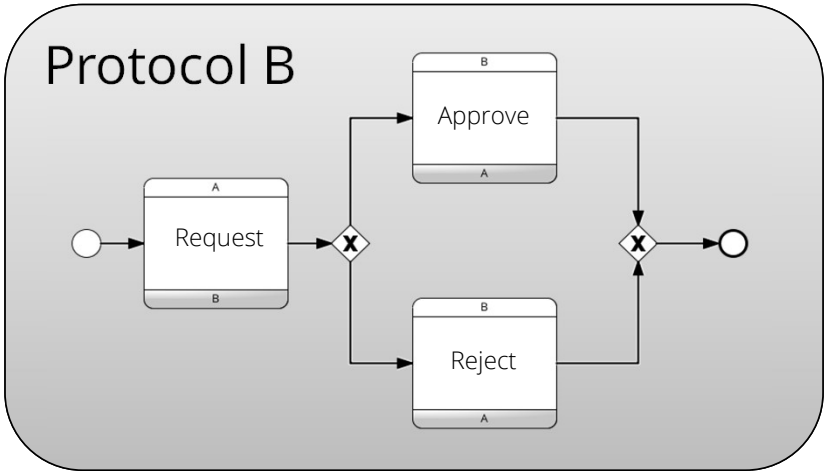


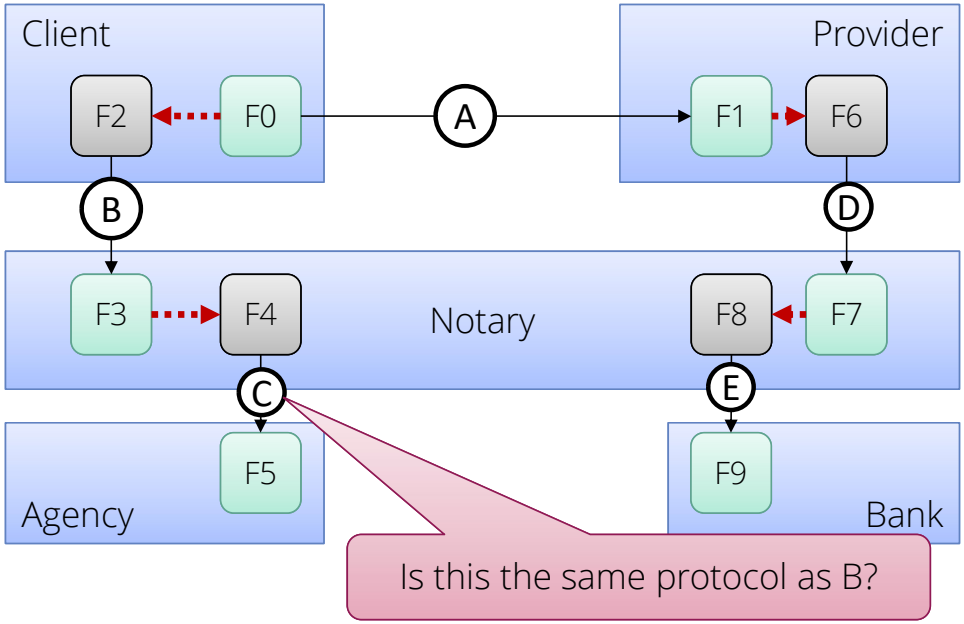
Rules of the transformation:

1. Dependencies between independent functions as **container nets**.
2. Function calls are **protocol choreographies**.

Utrecht University

Transformation of LM into Petri nets



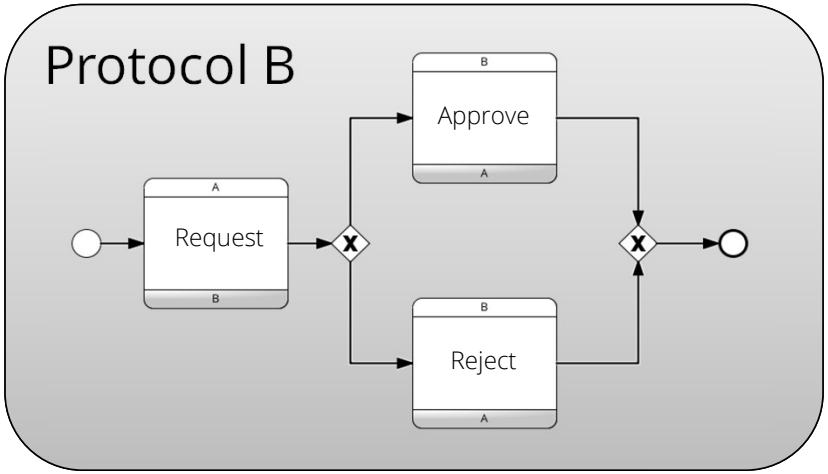


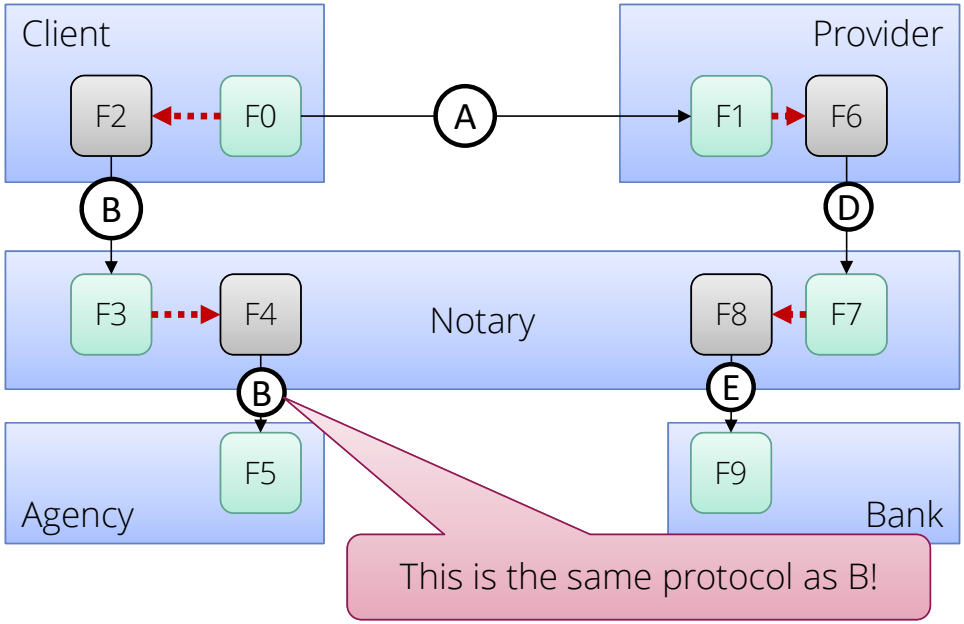
Rules of the transformation:

1. Dependencies between independent functions as **container nets**.
2. Function calls are **protocol choreographies**.

Utrecht University

Transformation of LM into Petri nets



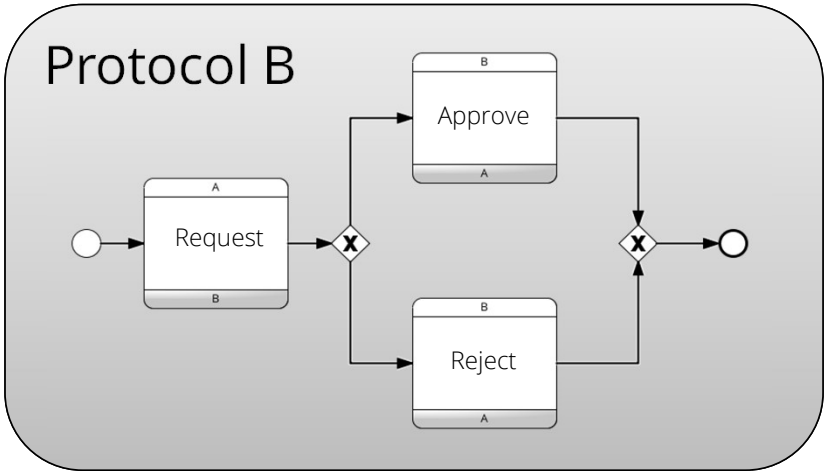


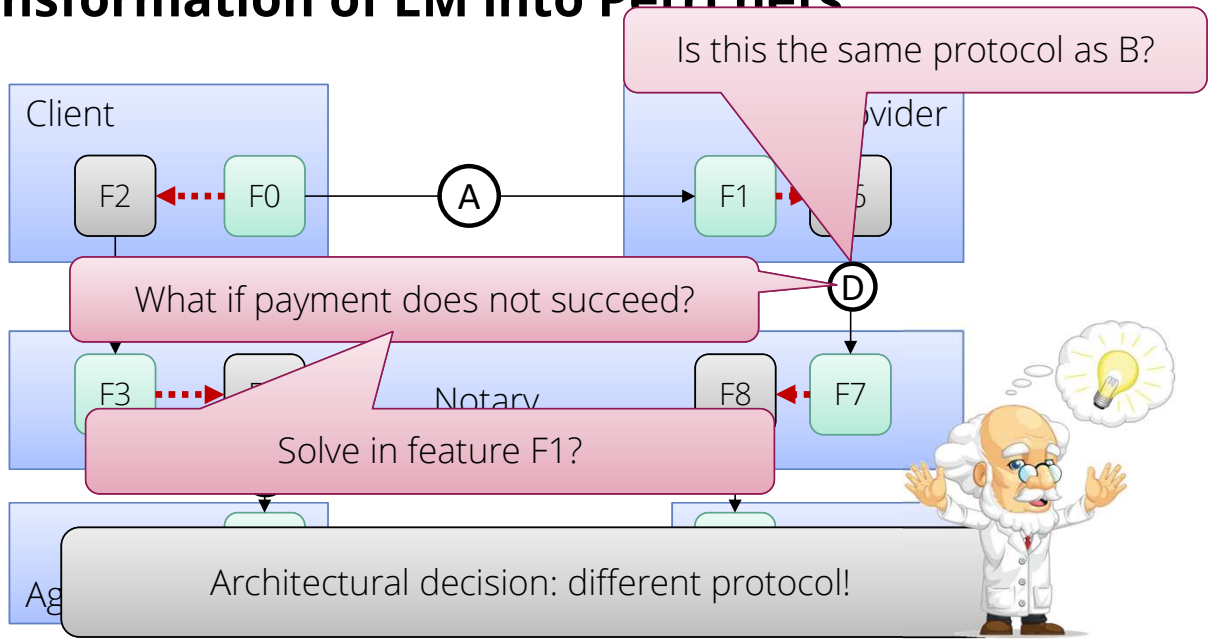
Rules of the transformation:

1. Dependencies between independent functions as **container nets**.
2. Function calls are **protocol choreographies**.

Utrecht University

Transformation of LM into Petri nets



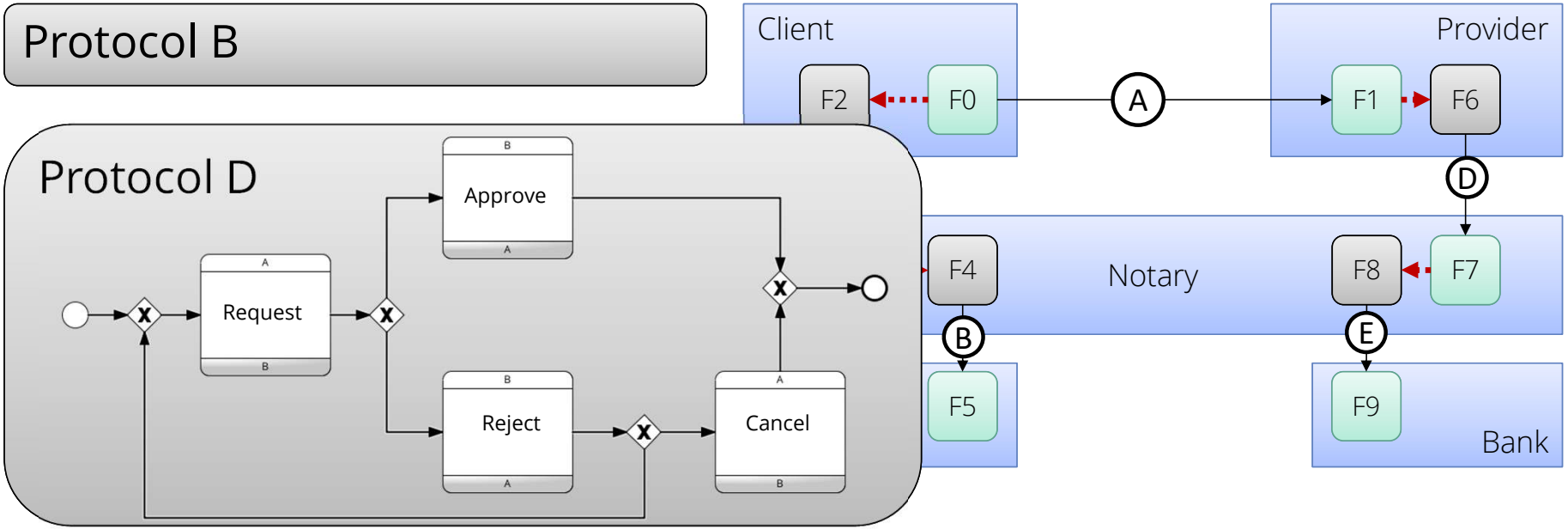


Rules of the transformation:

1. Dependencies between independent functions as **container nets**.
2. Function calls are **protocol choreographies**.

Utrecht University

Transformation of LM into Petri nets



Rules of the transformation:

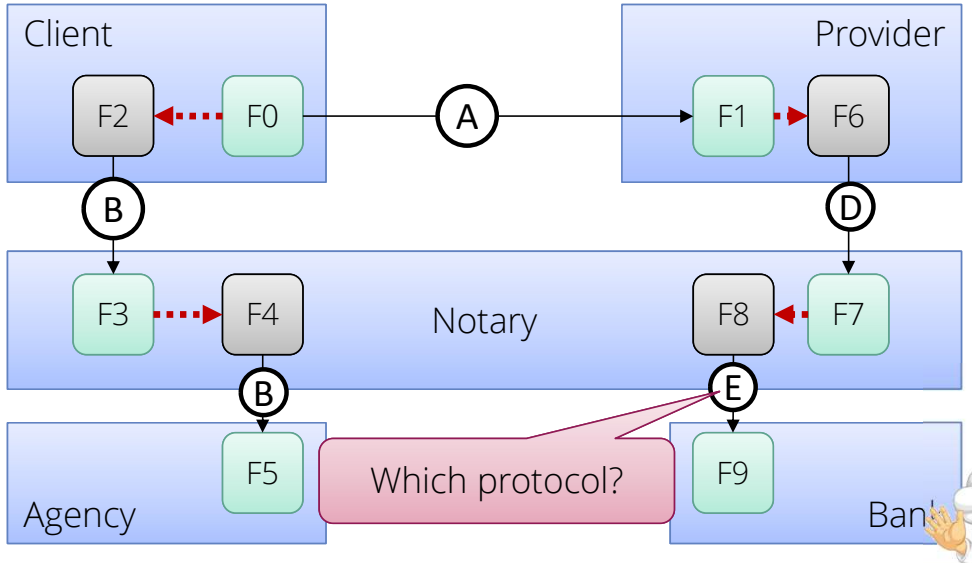
1. Dependencies between independent functions as **container nets**.
2. Function calls are **protocol choreographies**.

Utrecht University

Transformation of LM into Petri nets

Protocol B

Protocol D



Architectural decision: Remove bank from the system!

Rules of the transformation:

1. Dependencies between independent functions as **container nets**.
2. Function calls are **protocol choreographies**.

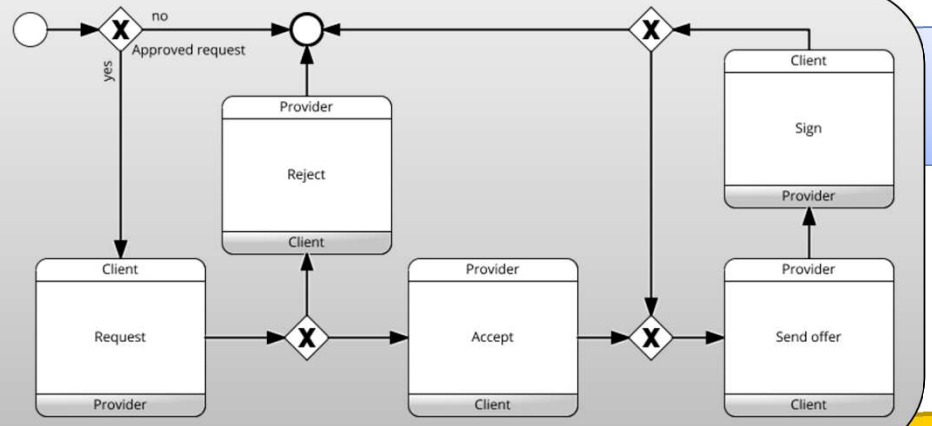
Utrecht University

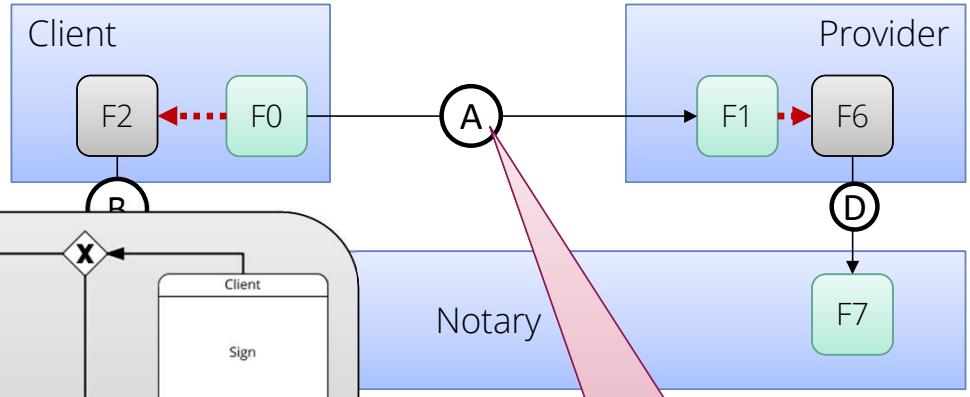
Transformation of LM into Petri nets

Protocol B

Protocol D

Protocol A





Which protocol?

What if function F2 fails?

Rules of the transformation:

1. Dependencies between independent functions as **container nets**.
2. Function calls are **protocol choreographies**.

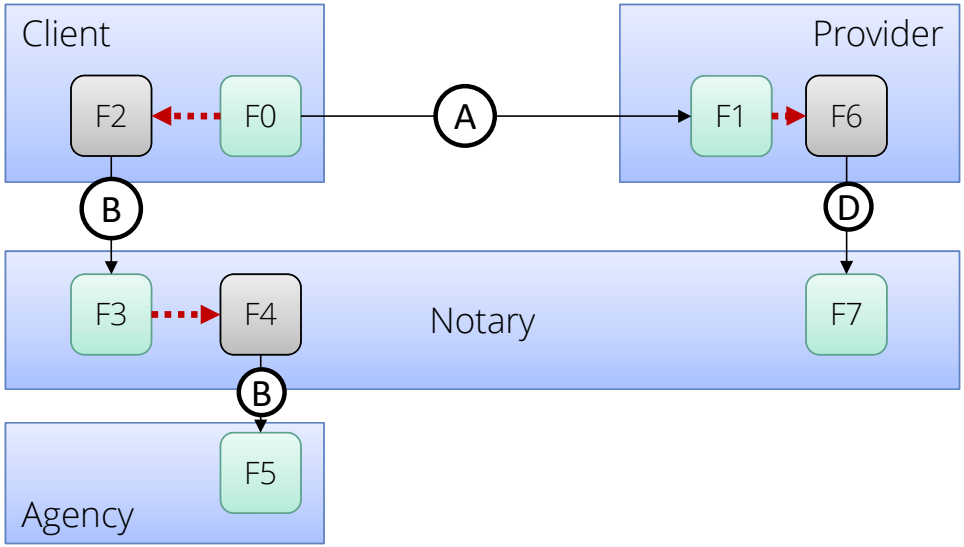
Utrecht University

Transformation of LM into Petri nets

Protocol B

Protocol D

Protocol A



Rules of the transformation:

1. Dependencies between independent functions as **container nets**.
2. Function calls are **protocol choreographies**.
3. Transform each protocol choreography into a “protocol Petri net”
4. Refinement step: refine each function with its protocol part.

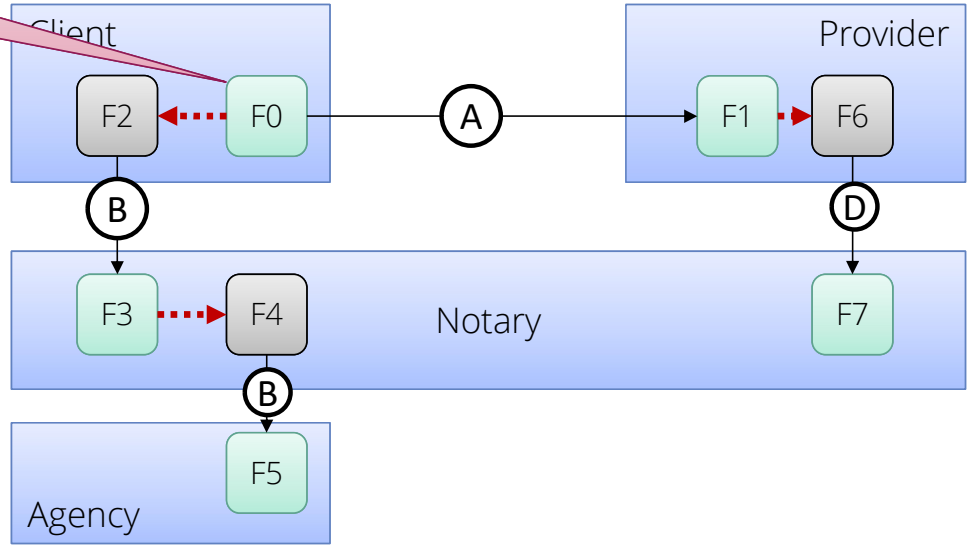
ansformation of LM into Petri nets

Select which place is refined in F0 for F2

Protocol B

Protocol D

Protocol A

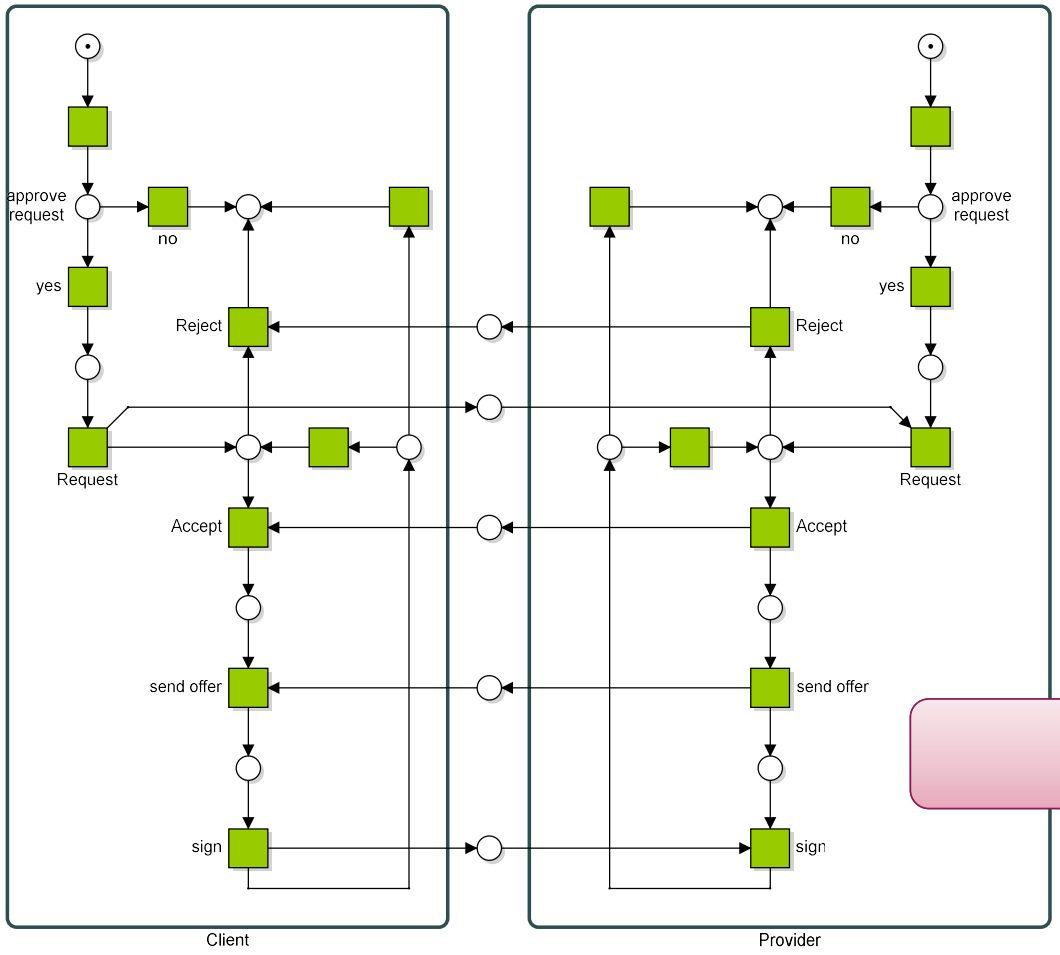


Rules of the transformation:

1. Dependencies between independent functions as **container nets**.
2. Function calls are **protocol choreographies**.
3. Transform each protocol choreography into a “protocol Petri net”
4. Refinement step: refine each function with its protocol part.
   For dependent functions: select the place in the protocol to refine

Utrecht University

Protocol net for Protocol A



Not a correct protocol! Back to the drawing board!

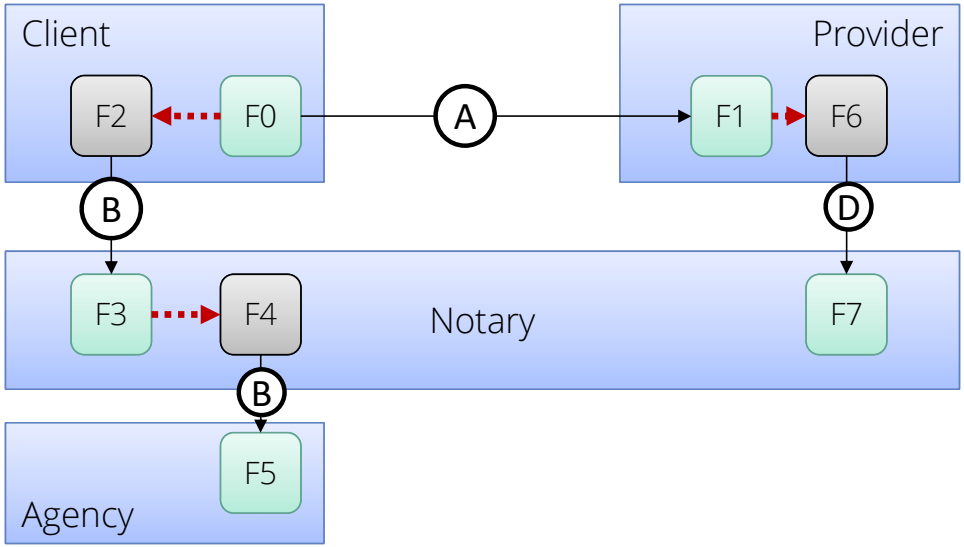
Utrecht University

Transformation of LM into Petri nets

Protocol B

Protocol D

Protocol A



Rules of the transformation:

1. Dependencies between independent functions as **container nets**.
2. Function calls are **protocol choreographies**.
3. Transform each protocol choreography into a “protocol Petri net”
4. Refinement step: refine each function with its protocol part.
   For dependent functions: select the place in the protocol to refine

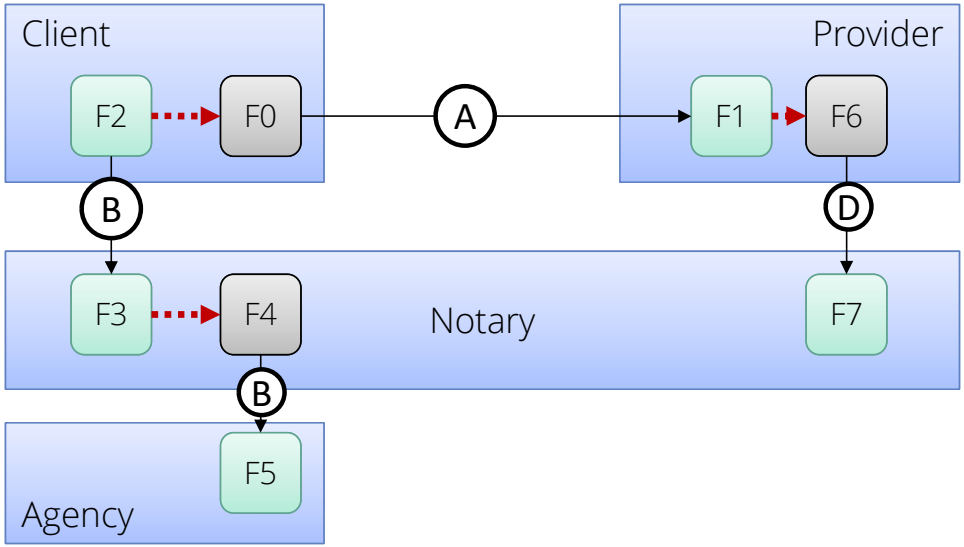
Utrecht University

Transformation of LM into Petri nets

Protocol B

Protocol D

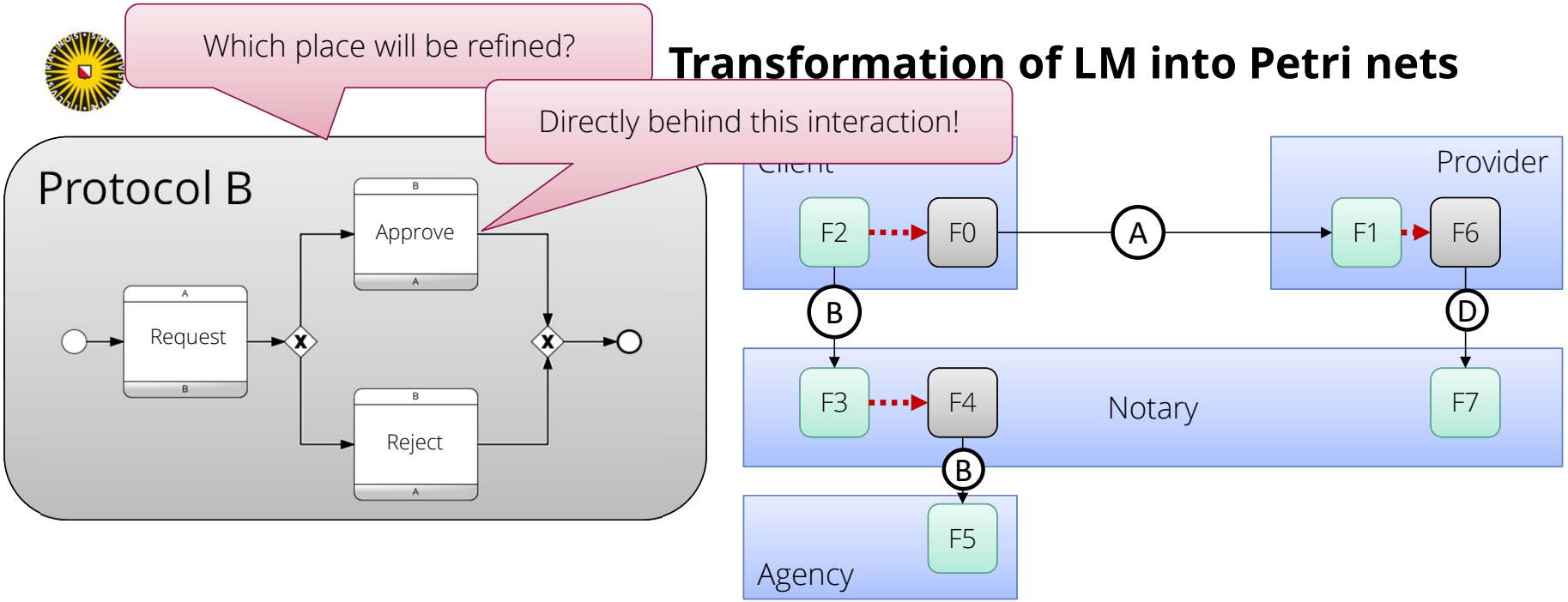
Protocol A



Rules of the transformation:

1. Dependencies between independent functions as **container nets**.
2. Function calls are **protocol choreographies**.
3. Transform each protocol choreography into a “protocol Petri net”
4. Refinement step: refine each function with its protocol part.
   For dependent functions: select the place in the protocol to refine

Transformation of LM into Petri nets



Rules of the transformation:

1. Dependencies between independent functions as **container nets**.
2. Function calls are **protocol choreographies**.
3. Transform each protocol choreography into a “protocol Petri net”
4. Refinement step: refine each function with its protocol part.
   For dependent functions: select the place in the protocol to refine

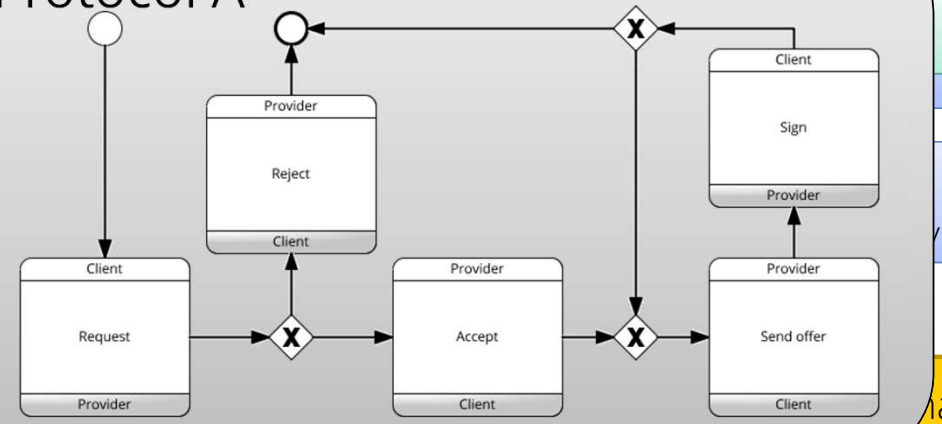
Utrecht University

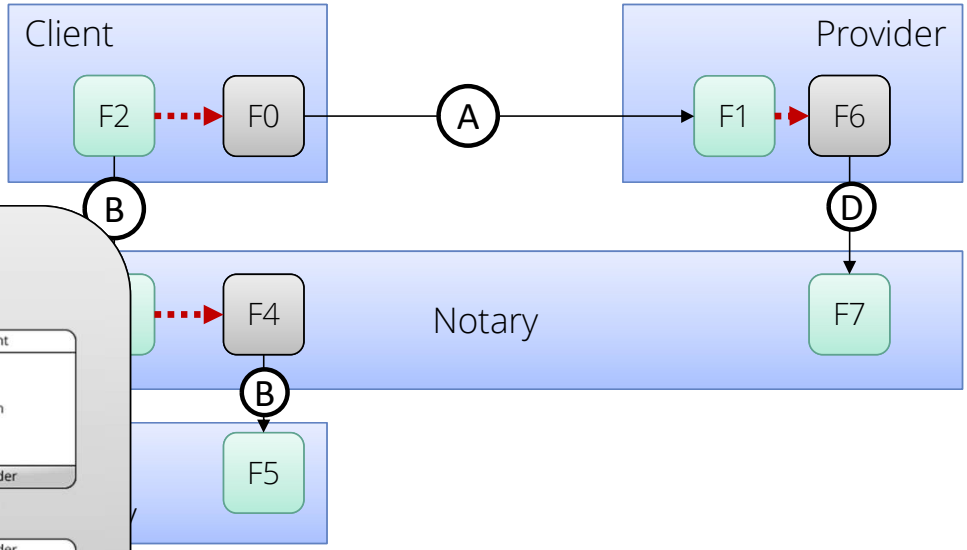
Transformation of LM into Petri nets

Protocol B

Protocol D

Protocol A





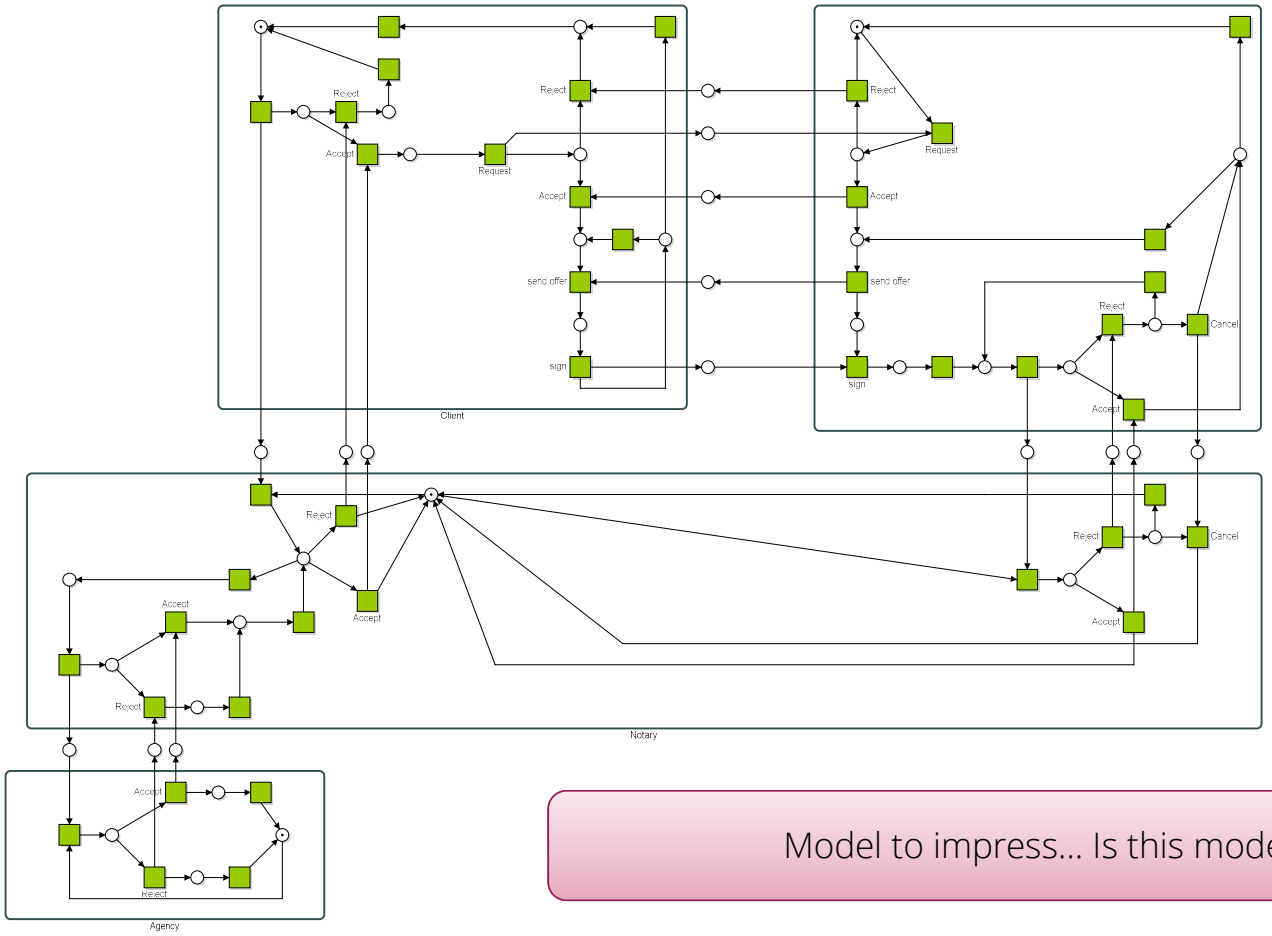
ıation:

etween independent functions as **container nets**.

2. Function calls are **protocol choreographies**.
3. Transform each protocol choreography into a “protocol Petri net”
4. Refinement step: refine each function with its protocol part.
   For dependent functions: select the place in the protocol to refine

Utrecht University

Resulting net



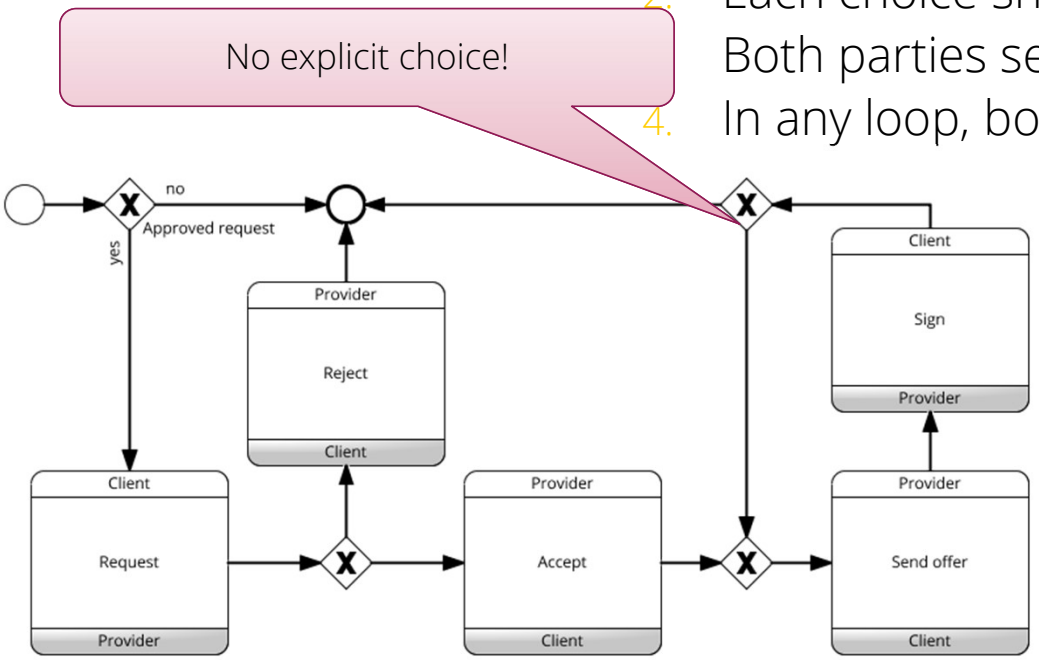
Model to impress... Is this model correct?

Utrecht University

Rules on choreographies
A sufficient condition

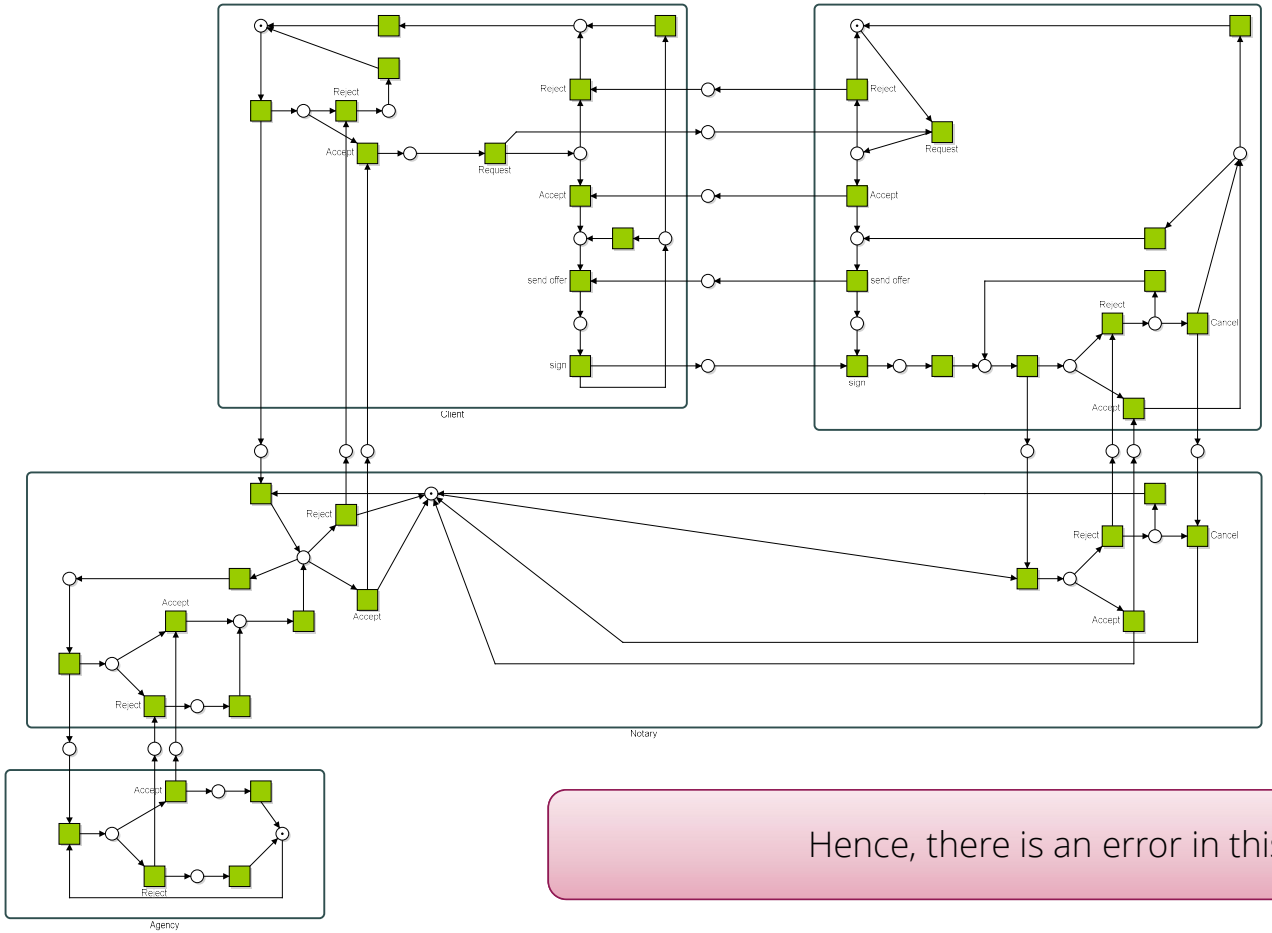
A choreography is correct if **all** of the following hold:

1. No parallelism! (sorry…)
2. Each choice should be made by one party only
3. Both parties send at least one message
4. In any loop, both parties should at least send one message



Utrecht University

Resulting net



Hence, there is an error in this model!

Utrecht University

Is the model correct after repairing protocol A?

When is the composition correct?

- Protocols between two containers only
- Protocol choreography with before mentioned rules

- Acyclic high level picture (i.e., LM) should be acyclic!

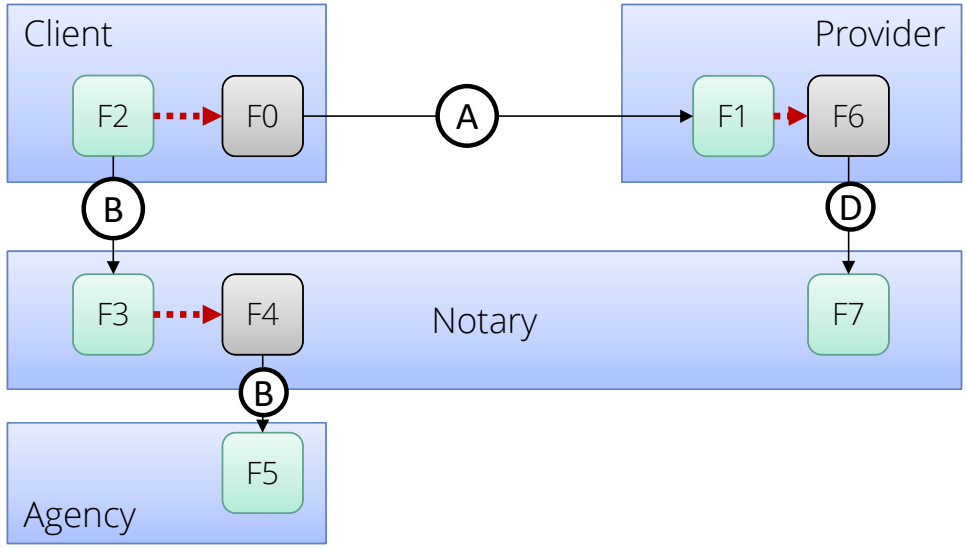
Utrecht University

Transformation of LM into Petri nets

Protocol B

Protocol D

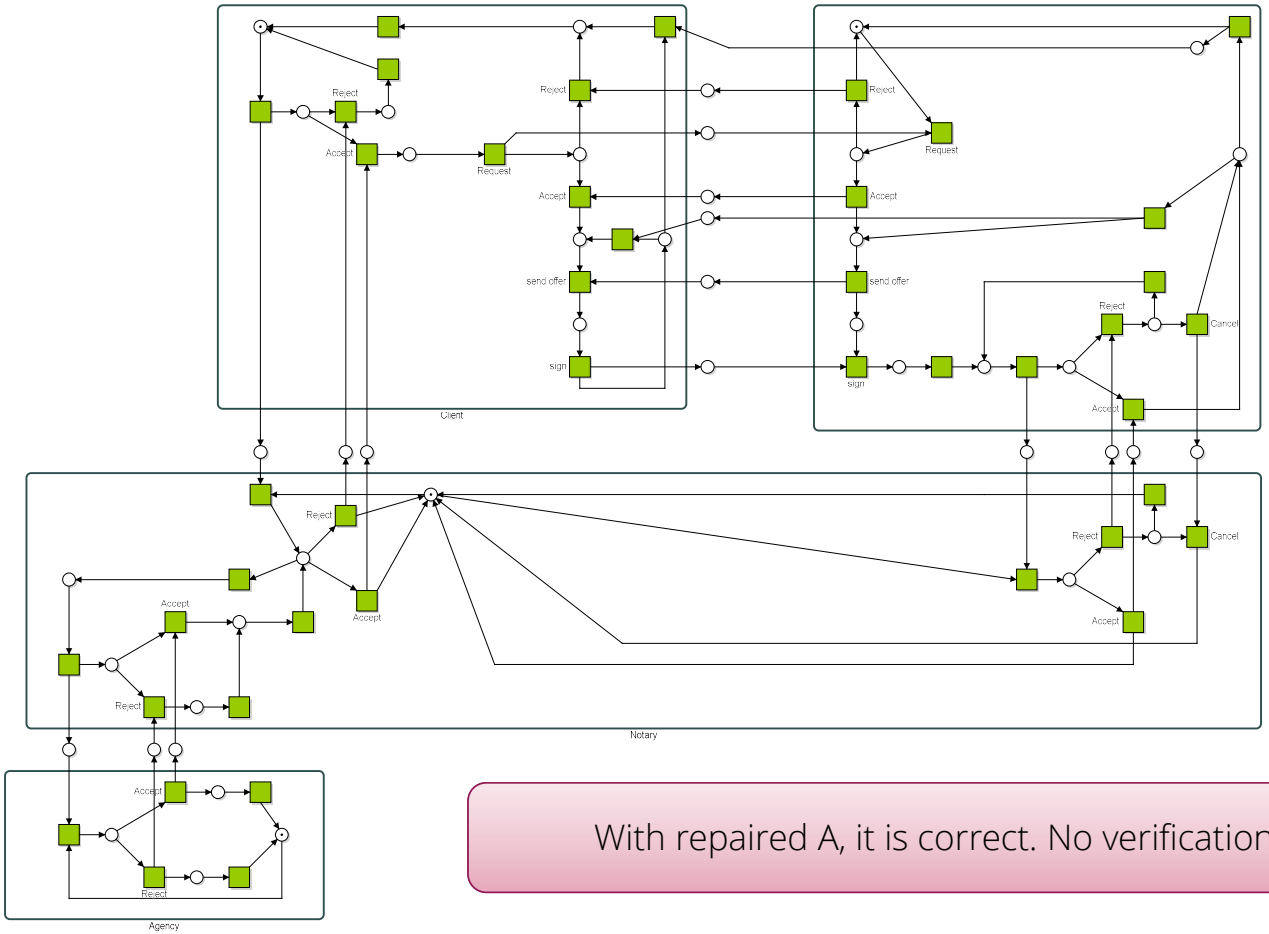
Protocol A



This logical model is acyclic

Utrecht University

Resulting net



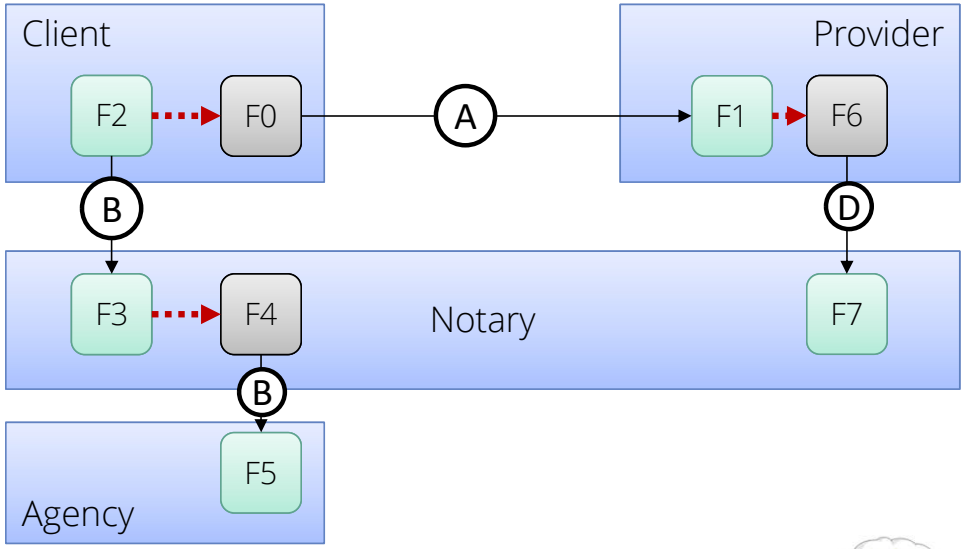
With repaired A, it is correct. No verification required ☺

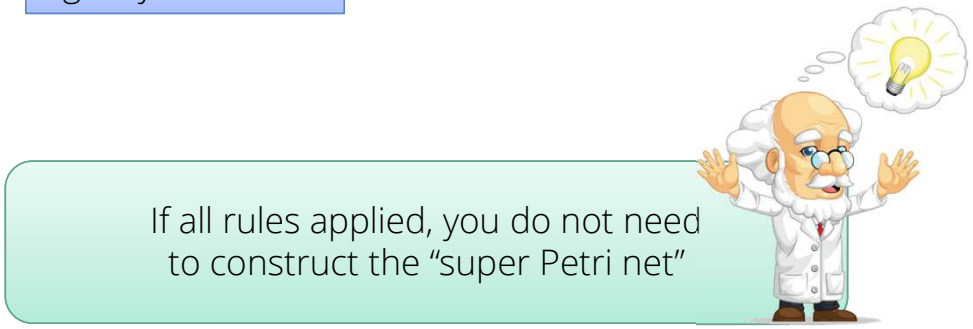
Utrecht University

Dynamics in Architecture Documentation

Documentation

- **Logical model**
- **Container nets**
- **Protocol A**
- **Protocol B**
- **Protocol D**





Utrecht University

About the debates…

Utrecht University

Architecture debate

- Evaluation of your architecture, serves as feedback!
- A debate lasts 30 minutes:
  - **Team A presents for 15 minutes**
  - **Team B&C are contesting (provide arguments against)**
  - **Team D&E are defending (provide arguments in favor)**
  - **Team F takes notes for team A**
- Discussion on pros and cons
  - **Proper trade-offs?**
  - **How do the trade-offs influence the architecture?**
  - **What are the consequences of the choices?**
  - **How can the architecture be improved?**

Utrecht University

DISCLAIMER

The information in this presentation has been compiled with the utmost care,
but no rights can be derived from its contents.