2. Modelling Dynamic Behavior with Petri Nets

Lecturer: Dr. Sebastian Götz

Prof. Dr. U. Aßmann
Technische Universität Dresden
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie
http://st.inf.tu-dresden.de/teaching/swt2
WS 2017, 17.10.2017

1. Basics
   1. Elementary Nets
   2. Special Nets
   3. Colored Petri Nets
2. Patterns in Petri Nets
3. Application to modelling

Obligatory Readings

- Balzert et al. (german)
  - Chapter 10.4 (p. 303ff)

- Ghezzi et al. (english)
  - Chapter 5.5.4 (p. 185ff)

- http://www.scholarpedia.org/article/Petri_net

Secondary Literature

- W.M.P. van der Aalst and A.H.M. ter Hofstede. **Verification of workflow task structures: A petri-net-based approach**. Information Systems, 25(1): 43-69, 2000.
- Kurt Jensen, Lars Michael Kristensen and Lisa Wells. **Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems**. Software Tools for Technology Transfer (STTT). Vol. 9, Number 3-4, pp. 213-254, 2007.
- J. B. Jörgensen. **Colored Petri Nets in UML-based Software Development – Designing Middleware for Pervasive Healthcare**. www.pervasive.dk/publications/files/CPN02.pdf
- Web portal "Petri Net World" http://www.informatik.uni-hamburg.de/TGI/PetriNets

Further Literature

- K. Jensen and L. M. Kristensen. **Colored Petri Nets**. Springer, 2009. (http://cs.au.dk/~cpnbook/)
- T. Murata. **Petri Nets: properties, analysis, applications**. IEEE volume 77, No 4, 1989.
- W. Reisig. **Elements of Distributed Algorithms – Modelling and Analysis with Petri Nets.** Springer. 1998.
- W. Reisig, G. Rozenberg. **Lectures on Petri Nets I+II**, Lecture Notes in Computer Science, 1491+1492, Springer.
- J. Peterson. **Petri Nets**. ACM Computing Surveys, Vol 9, No 3, Sept 1977

Goals

- Understand Untyped (Page/Transition nets) and Colored Petri nets (CPN)
- Understand that PN/CPN are a verifiable and automated technology for safety-critical systems
- Understand why PN are a good modeling language for parallel systems simulating the real world
- PN have subclasses corresponding to finite automata and data-flow graphs
- PN can be refined, then reducible graphs result

The Initial Problem

You work for PowerPlant Inc. Your boss comes in and says:

"Our government wants a new EPR reactor, similarly, in the way Finland has it."

How can we produce a verified control software?

We need a good modelling language!

How do we produce software for safety-critical systems?

Projects with Safety-Critical, Parallel Embedded Software

Aerospace

- The WITAS UAV unmanned autonomously flying helicopter from Linköping http://www.ida.liu.se/~marwz/papers/ICAPS06_System_Demo.pdf

Automotive

- Prometheus: driving in car queues on the motorway http://www.springerlink.com/content/j06n312r36805683/

Trains

- www.railcab.de Autonomous rail cabs
- The Copenhagen metro (fully autonomous)
  - Inauguration seminar http://www.cowi.com.pl/SiteCollectionDocuments/cowi/en/menu/02.%20Services/03.%20Transport/5.%20Tunnels/Other%20file%20types/Copenhagen%20Metro%20Inauguration%20Seminar.pdf

3.1 Basics of PN

Petri Net Classes

- Predicate/Transition Nets: simple tokens, no hierarchy.
- Place-Transition Nets: multiple tokens
- High Level Nets: structured tokens, hierarchy
- There are many other variants, e.g., with timing constraints

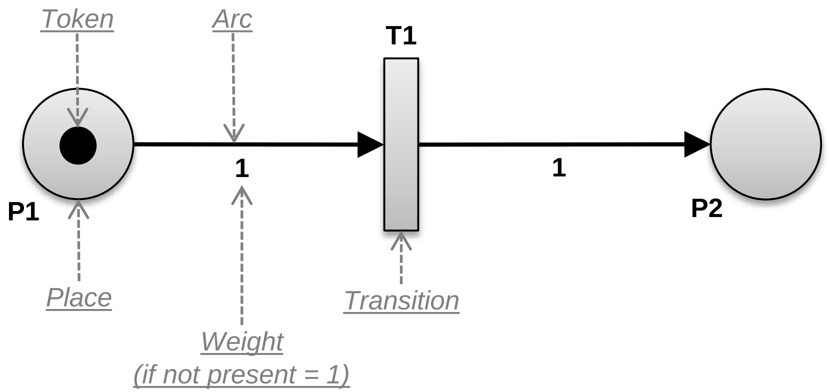
Petri Nets

Model introduced by Carl Adam Petri in 1962,
C.A. Petri. Ph.D. Thesis: ”Communication with Automata”.

- Over many years developed within GMD (now Fraunhofer, FhG)
- PNs specify diagrammatically:
  - Infinite state systems, regular and non-decidable
  - Concurrency (parallelism) with conflict/non-deterministic choice
  - Distributed memory (“places” can be distributed)
- Modeling of parallelism and synchronization
- Behavioral modeling, state modeling etc.

Integer Place/Transition Nets

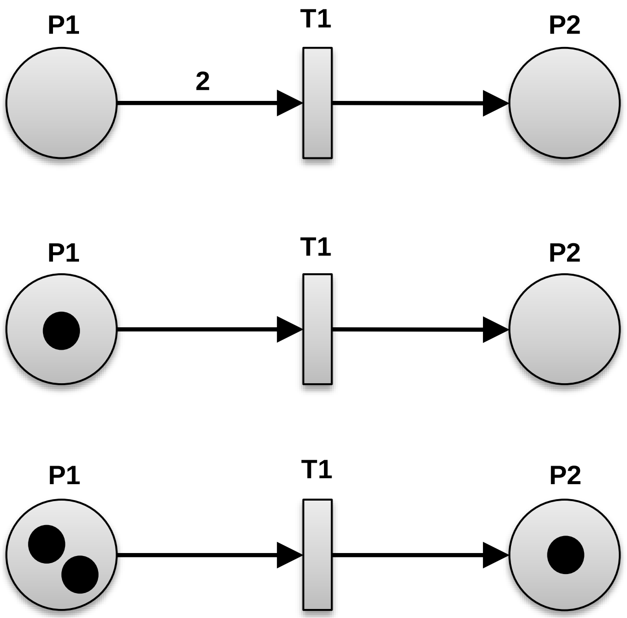
- Tupel (P,T,F,W,m_0)
  - **P** = Places $P \cap T = \emptyset$
  - **T** = Transistions
  - **F** = Flow Relations $F \subseteq (P \times T) \cup (T \times P)$
  - **W** = (Relation) Weight $W: F \rightarrow \mathbb{N}_0$ wobei
    $\mathrm{W(p,t)} = 0 \equiv (p,t) \notin F, p \in P \; und \; t \in T$ und
    $\mathrm{W(t,p)} = 0 \equiv (t,p) \notin F, p \in P \; und \; t \in T$
  - $\mathbf{m_0}$ = Start Marking $m_0: P \rightarrow \mathbb{N}_0$




```
P = {P1, P2}
T = {T1}
F = {(P1,T1), (T1,P2)}
W = f(x) = 1
m0 = {P1}
```

Integer Place/Transition Nets

- A **marking** $m(p) \rightarrow \mathbb{N}_0,\ p \in P$ assigns a non-negative Integer to places
  - Number of tokens in a place
- A **weight** $W(f) \rightarrow \mathbb{N}_0,\ f \in F$ assigns a non-negative Integer to arcs
  - How many tokens can they carry



$m(P1) \rightarrow 0$
$m(P2) \rightarrow 0$

$m(P1) \rightarrow 1$
$m(P2) \rightarrow 0$

$m(P1) \rightarrow 2$
$m(P2) \rightarrow 1$

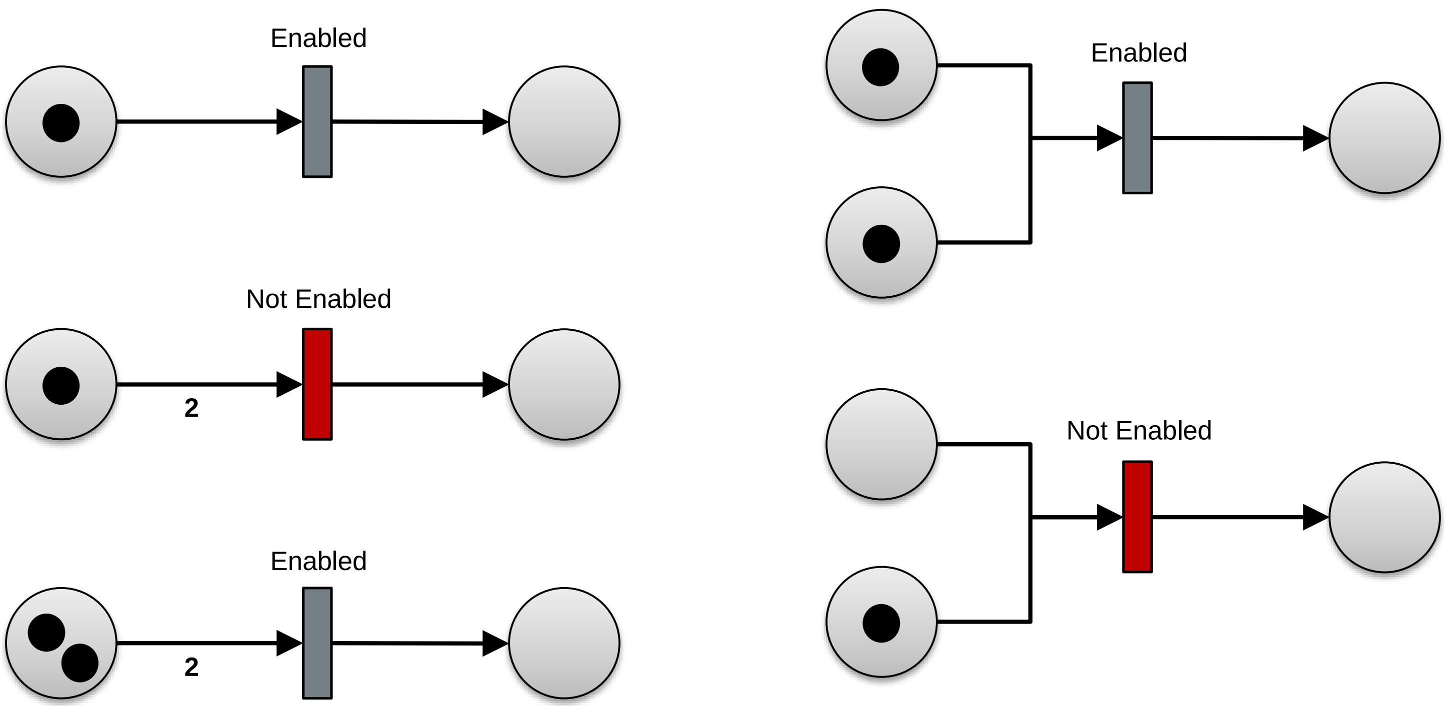
Integer Place/Transition Nets

- Transition $t \in T$ is **enabled** when

  $m(p) - W(p,t) > 0, \forall p \in P$

  - For all incoming arcs, the places must contain at least n tokens

    → n = the weight of the incoming arc



Integer Place/Transition Nets

- When a transition is Enabled, it may or may not fire
- When a transition $t \in T$ **fires**

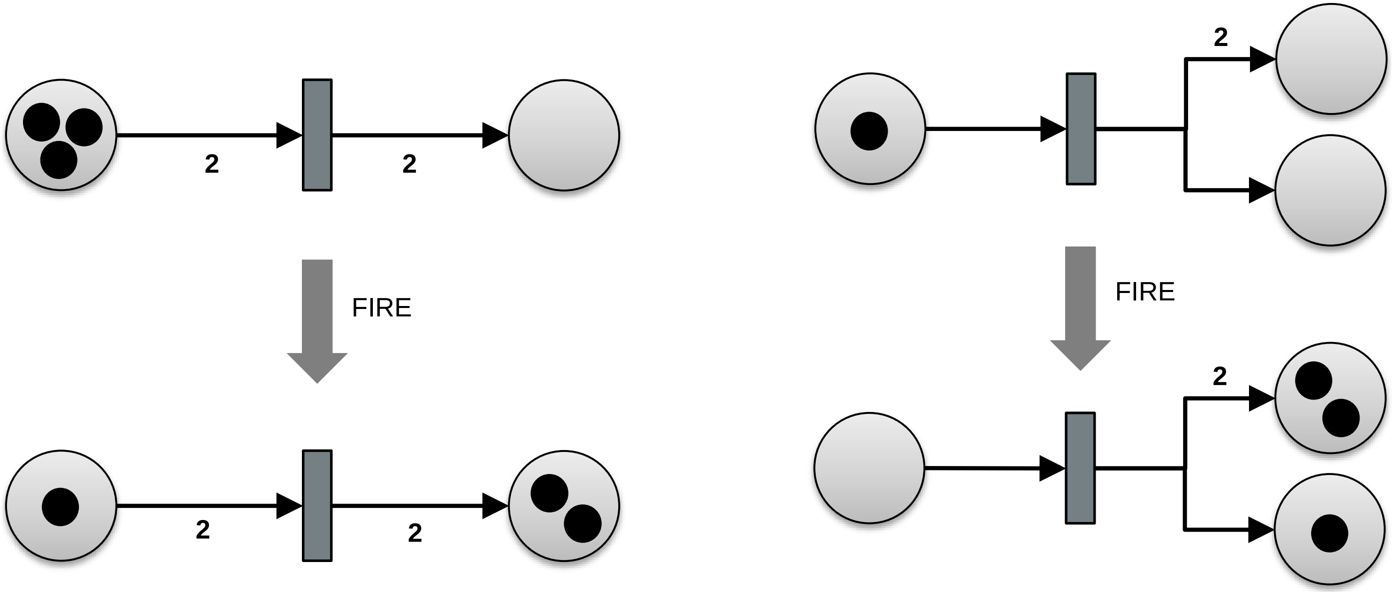
$$m(p) = m(p) - W(p,t), \forall p \in P$$

  - N Tokens are removed from all incoming places

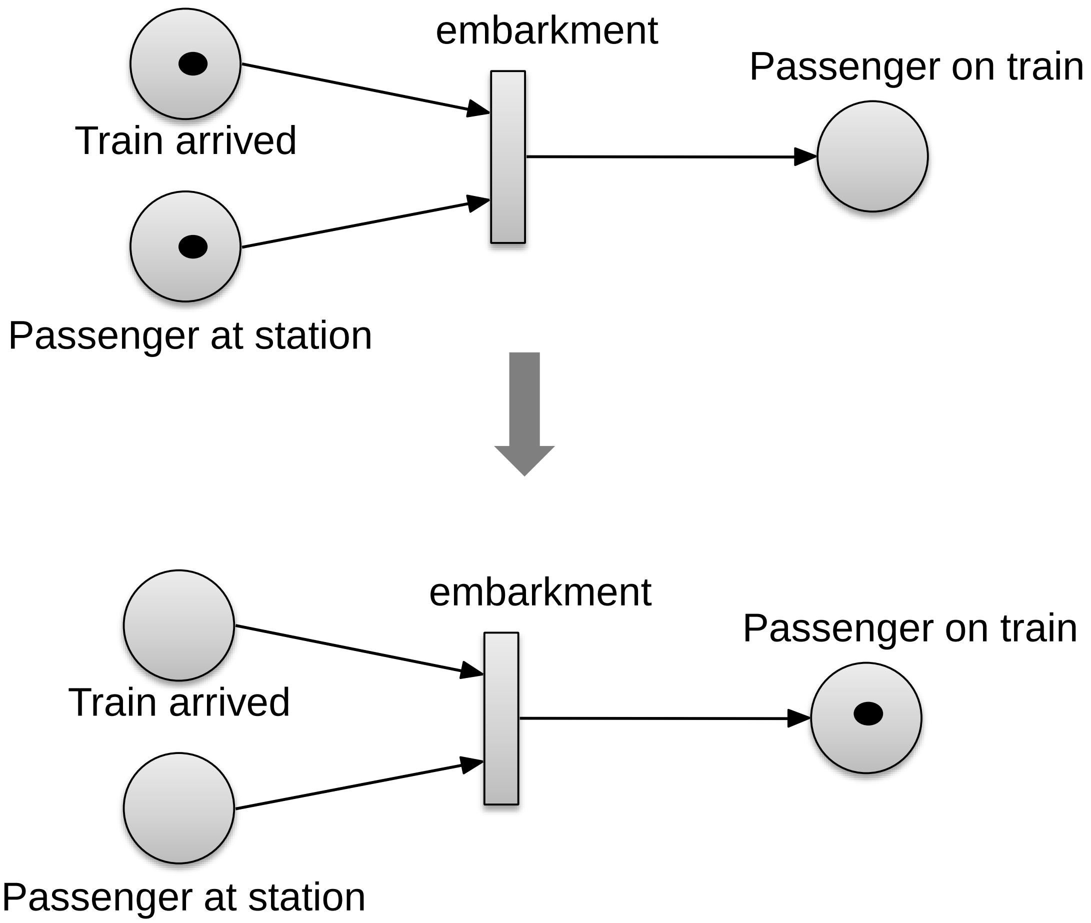
$$m(p) = m(p) + W(t,p), \forall p \in P$$

  - M Tokens are added to all outgoing places

- The state (marking) of the Petri Net is changed



Ex.: Department of a Train



Elementary Nets: Predicate/Transition Nets

- A **Petri Net (PN)** is a directed, bipartite graph over two kinds of *nodes*
  - 1. Places (circles)
  - 2. Transitions (bars or boxes)

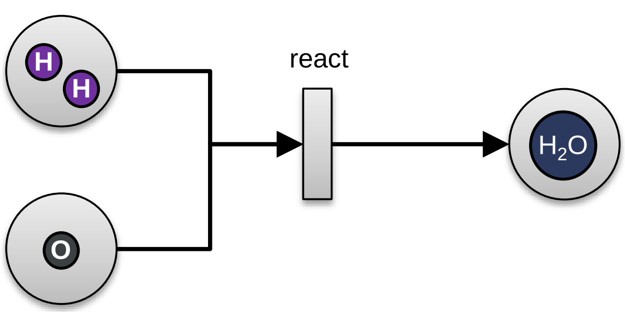
- A **Integer PN** is a directed, weighted, bipartite graph with integer tokens
  - Places may contain several tokens
  - Places may contain a capacity (bound=k)
  - k tokens in a place indicate that k items are available

Integer Place/Transitions-Nets

- An ***Elementary PN*** (boolean net, predicate/transition or condition/event nets)
  - Boolean tokens
    One token per place (bound of place = 1)
  - Arcs have no weights
  - Presence of a token = condition or predicate is true
  - *Firing* of a transition = from the input the output predicates are concluded
  - Thus elementary PN can represent simple forms of logic

High-Level Nets

- A **High-Level PN** (Colored PN, CPN) allows for *typed places* and *typed arcs*
  - For types, any DDL can be used (e.g., UML-CD)
- High-level nets are modular
  - Places and transitions can be refined
  - A Colored Petri Net is a reducible graph
- The upper layers of a reducible CPN are called *channel agency nets*
  - Places are interpreted as channels between components



Application Areas of Petri Nets

- Reliable software (quality-aware software)
  - PetriNets can be checked on deadlocks, liveness, fairness, bounded resources
- Safety-critical software that require proofs
  - Control software in embedded systems or power plants
- Hardware synthesis
  - Software/Hardware co-design
- User interface software
  - Users and system can be modeled as parallel components

Application Area I: Behavior Specifications in UML

- Instead of describing the behavior of a class with a statechart, a CPN can be used
  - Statecharts, data flow diagrams, activity diagrams are subsets of CPNs
- CPN have several advantages:
  - They model **parallel** systems (with a fixed net) naturally
  - They are compact and **modular**, they can be reducible
  - They are suitable for **aspect-oriented** composition, in particular of parallel protocols
  - They can be used to **generate code**, also for complete applications
- Informal: for CPN, the following features can be proven
  - **Liveness**: The net can fire at least n times
  - **Fairness**: All parts of the net are equally “loaded” with activity
  - **K-boundedness**: The number of tokens, a place can contain, are bound by k
  - **Deadlock**: The net cannot proceed but did not terminate correctly
  - **Deadlock-freeness**: The net contains no deadlocks

Application Area II: Contract checking (Protocol Checking) for Components

- Petri Nets describe behavior of components (dynamic semantics)
  - They can be used to check whether components fit to each other
- Problem: General fit of components is undecidable
  - The protocol of a component must be described with a decidable language
  - Due to complexity, context-free or -sensitive protocol languages are required
- Algorithm:
  - Describe the behavior of two components with two CPN
  - Link their ports
  - Check on *liveness* of the unified CPN
  - If the unified net is not live, components will not fit to each other…
- Liveness and fairness are very important criteria in safety-critical systems

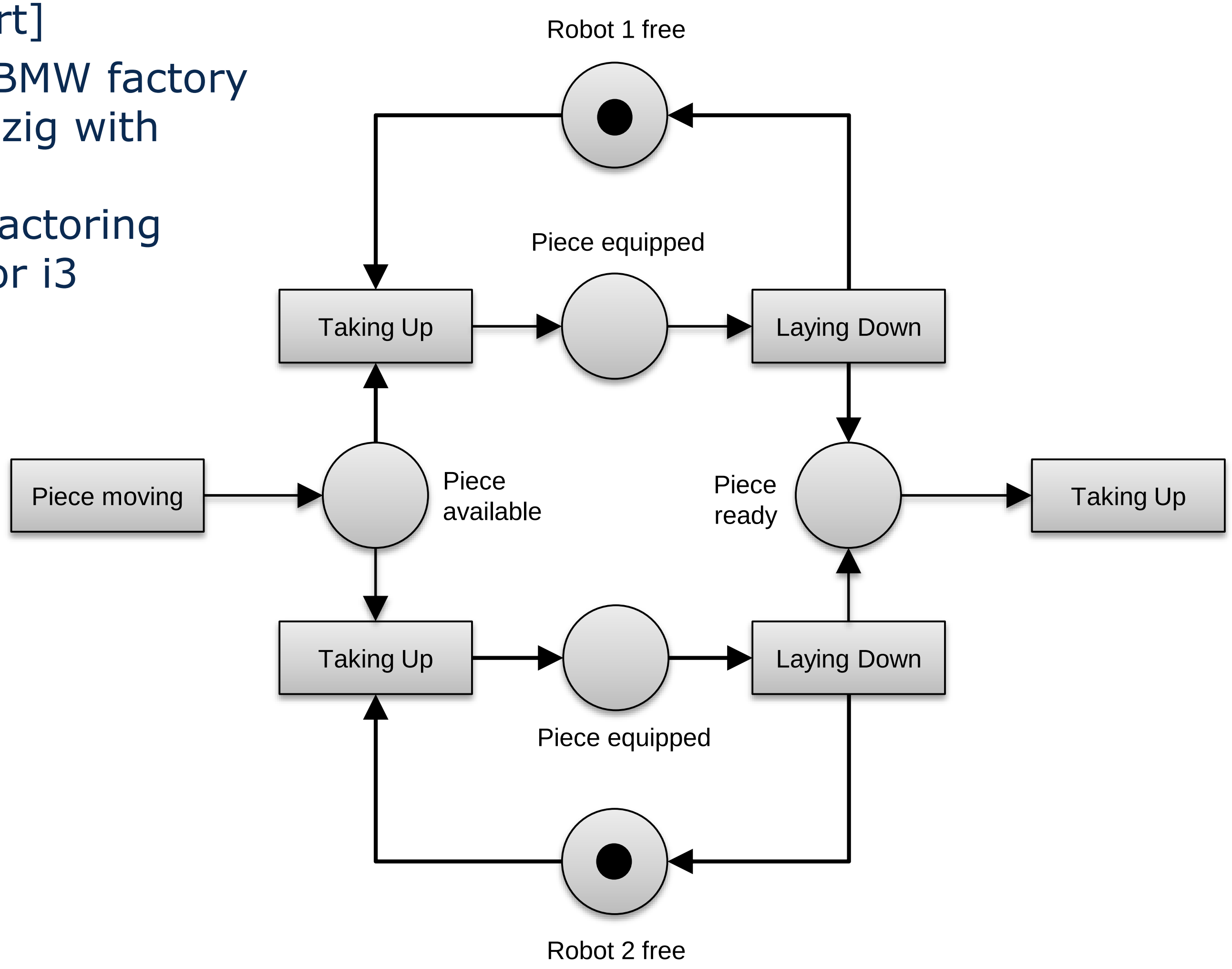
3.1.1 Elementary Nets (Predicate/Transition Nets)

Meaning of Places and Transitions in Elementary Nets

- Predicate/Transition (Condition/Event-, State/Transition) Nets:
  - Places represent conditions, states, or predicates
  - Transitions represent the firing of events:
    - if a transition has one input place,
      the event fires immediately if a token arrives in that place
    - If a transition has several input places,
      the event fires when all input places have tokens
- A transition has input and output places (pre- and postconditions)
  - The presence of a token in a place is interpreted as the condition is true

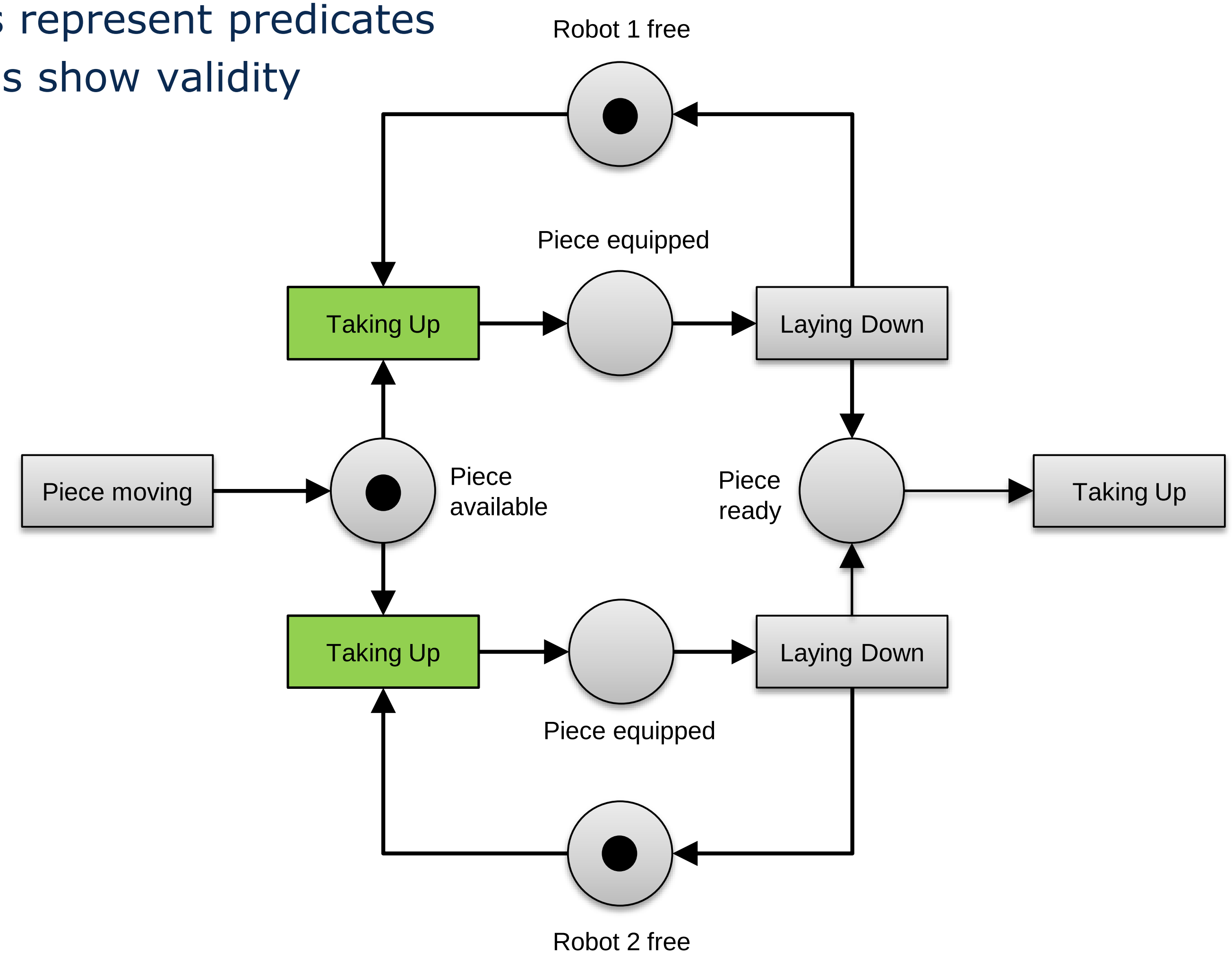
Example of 2 Robots as Predicate/Transition Net

- [Balzert]
- Cmp. BMW factory in Leipzig with robot manufactoring cells for i3

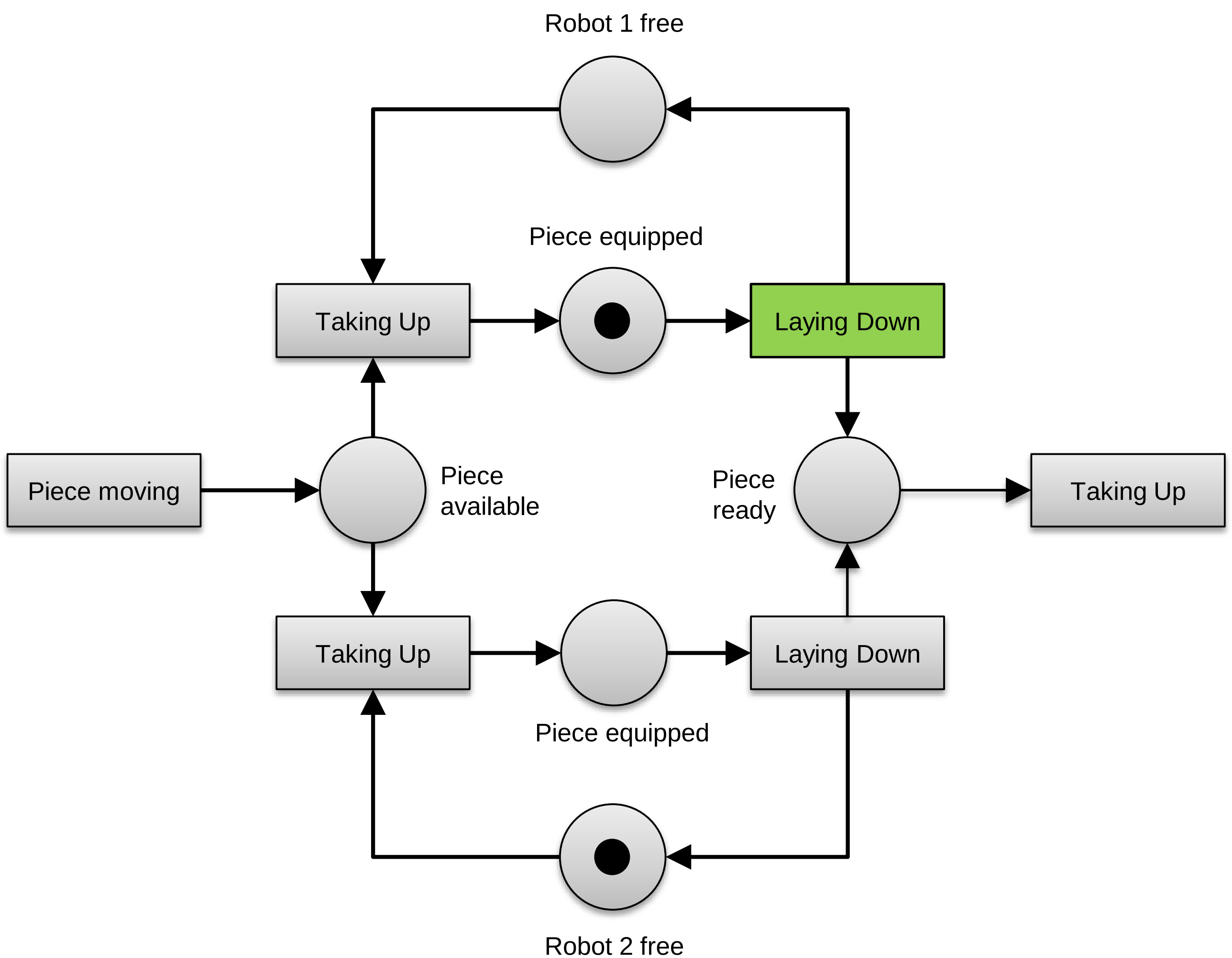


Example of 2 Robots as Predicate/Transition Net

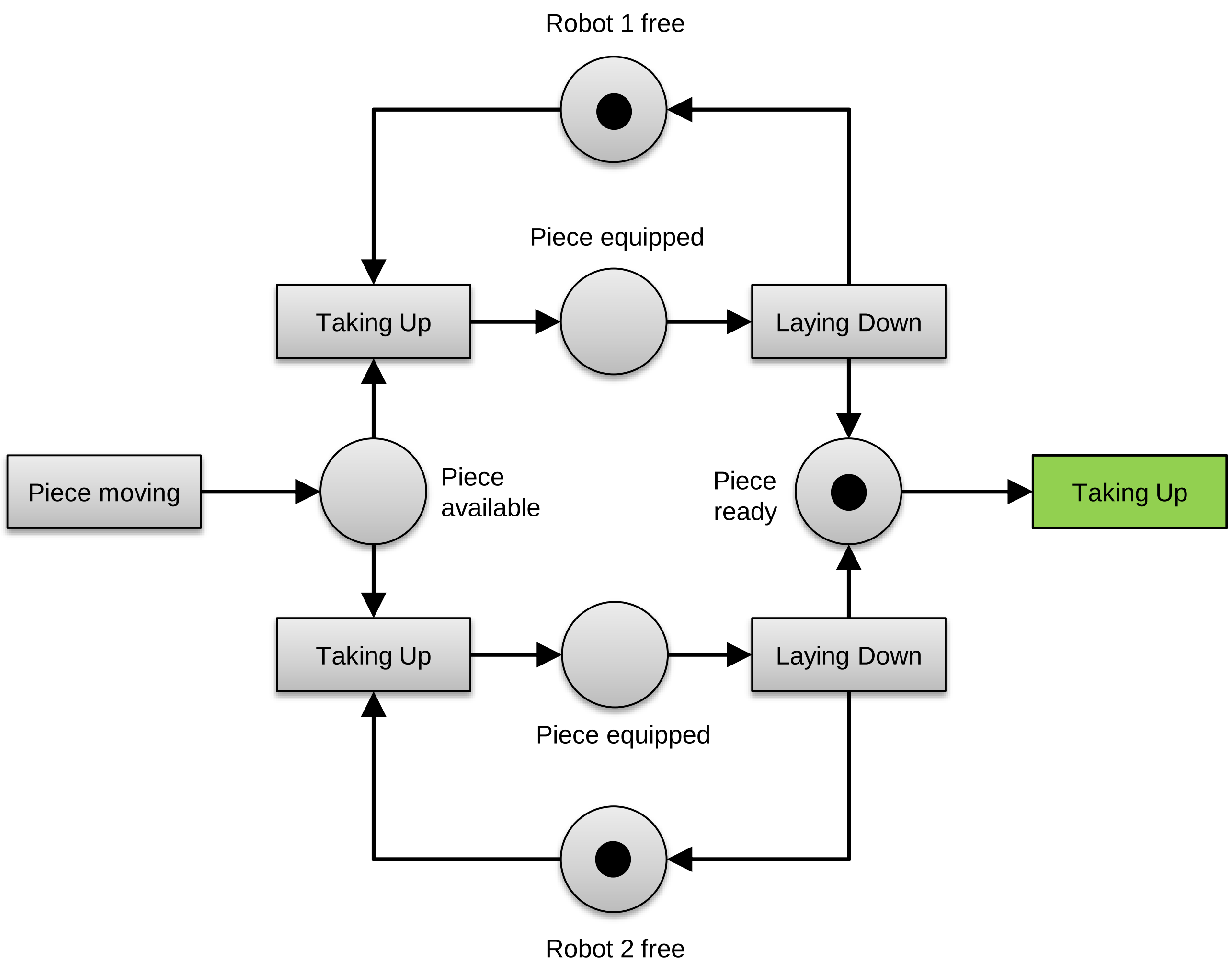
- Places represent predicates
- Tokens show validity



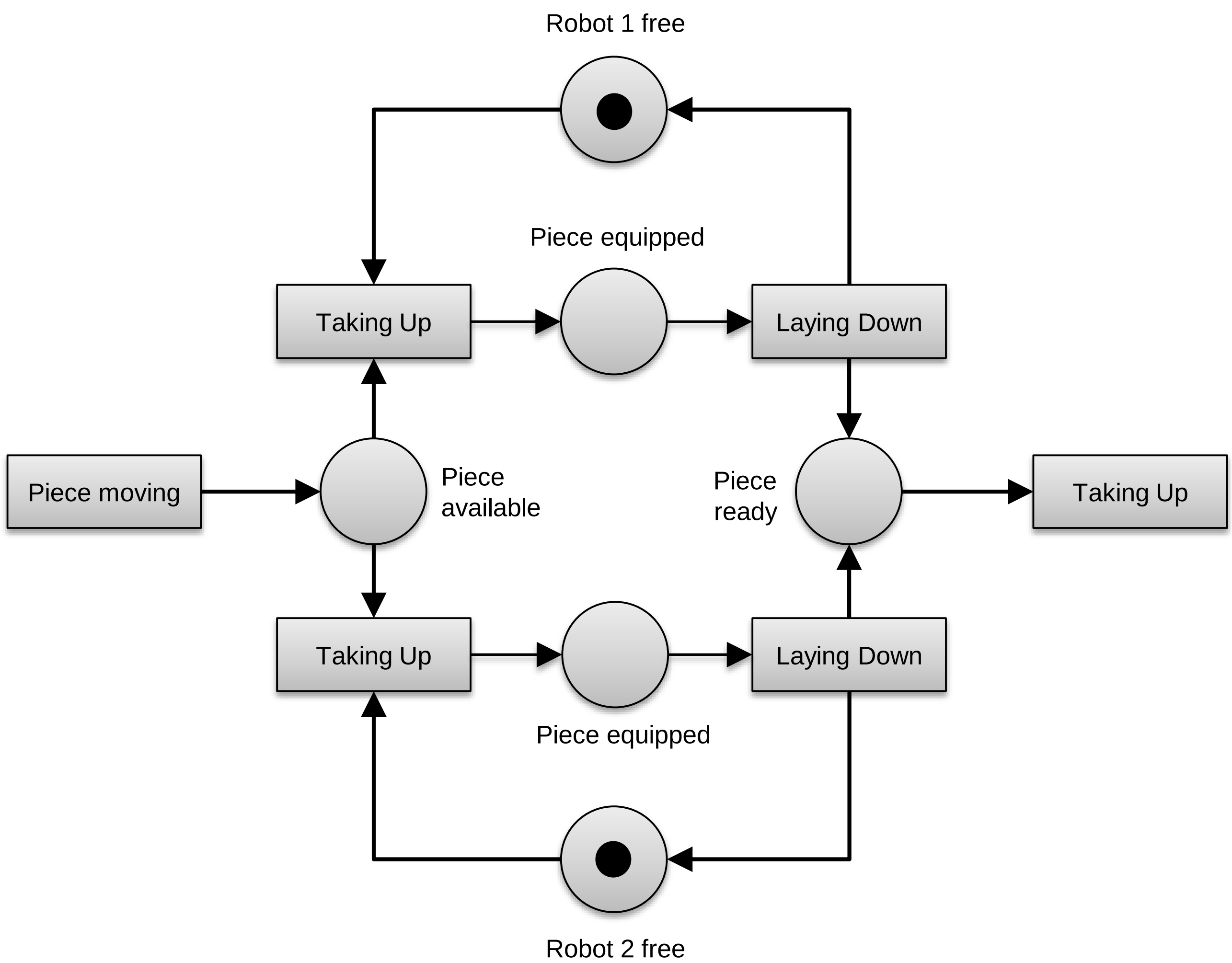
Example of 2 Robots as Predicate/Transition Net



Example of 2 Robots as Predicate/Transition Net



Example of 2 Robots as Predicate/Transition Net



Comparing PN to Automata

Petri Nets

- Tokens encode parallel “distributed” global state
- Can be switched “distributedly”

Automata

- Sequential
- One global state (one token)
- Can only be switched “centrally”

3.1.2 Special Nets (Special Syntactic forms of PN)

3.1.2.a Marked Graphs (MG) are DFD with Distributed Memory

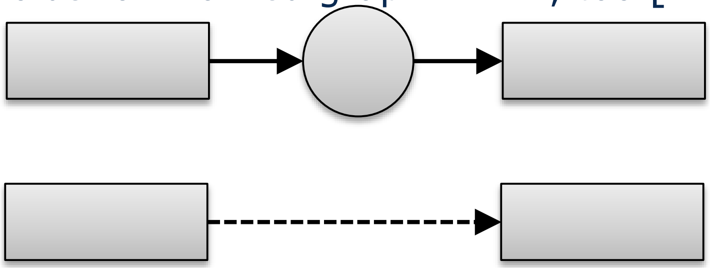
- A **Marked Graph** (MG) is a PN such that:

  1. Each place has only 1 incoming arc
  2. Each place has only 1 outgoing arc

  - Then the places can be abstracted (identified with one flow edge)
  - Transitions may split and join, however
  - No shared memories between transitions (distributed memory)

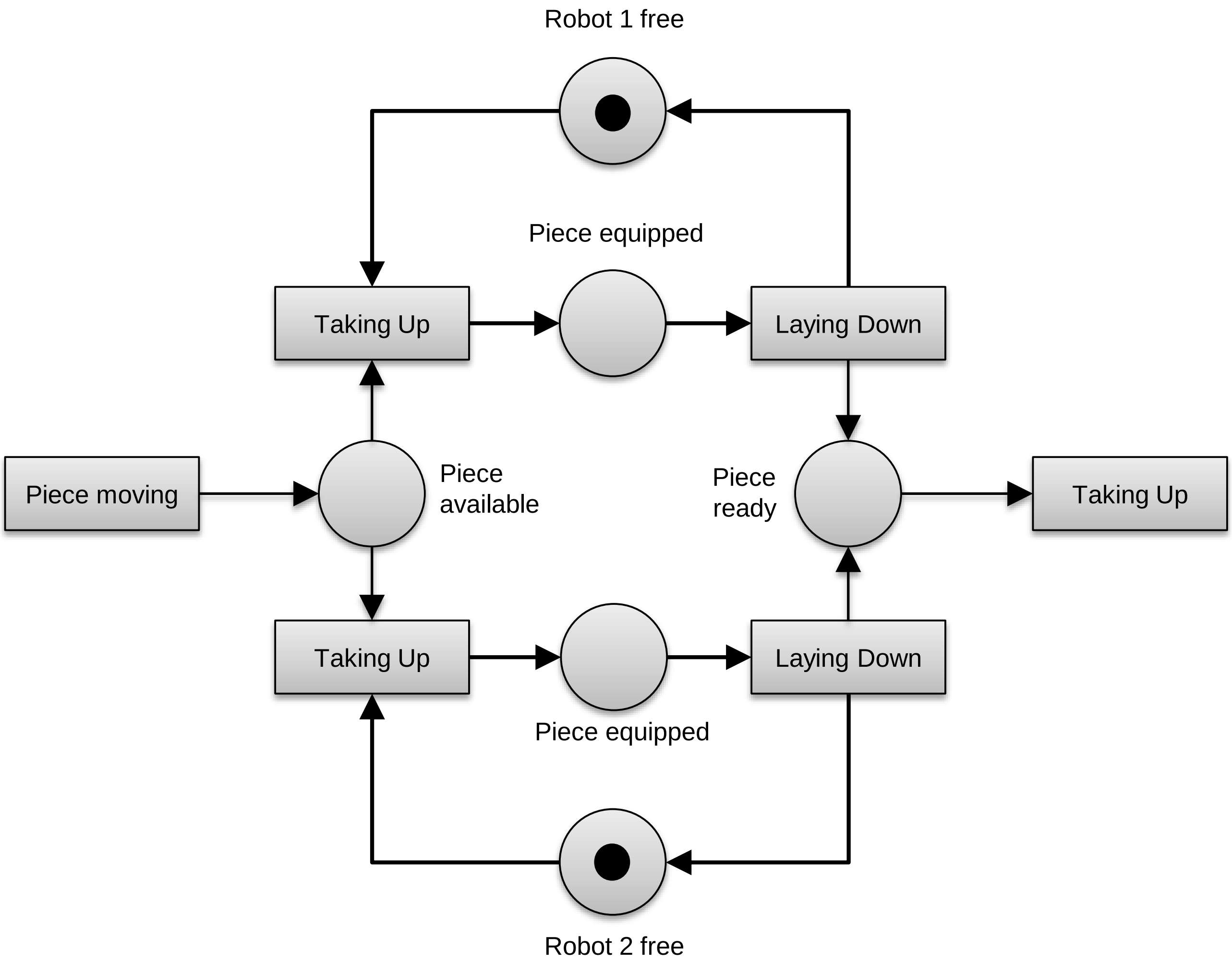
- Marked Graphs correspond to a special class of data-flow graphs **(Data flow diagrams with non-shared, distributed memory, dm-DFD)**
  - MG provide *deterministic parallelism without confusion*
  - Transitions correspond to processes in DFD, places to stores
  - States can be *merged* with the ingoing and outcoming arcs → DFD without stores
  - Restriction: Stores have only one producer and consumer
  - But activities can join and split

- All theory for CPN holds for marked graph - DFD, too [BrozaWeide]



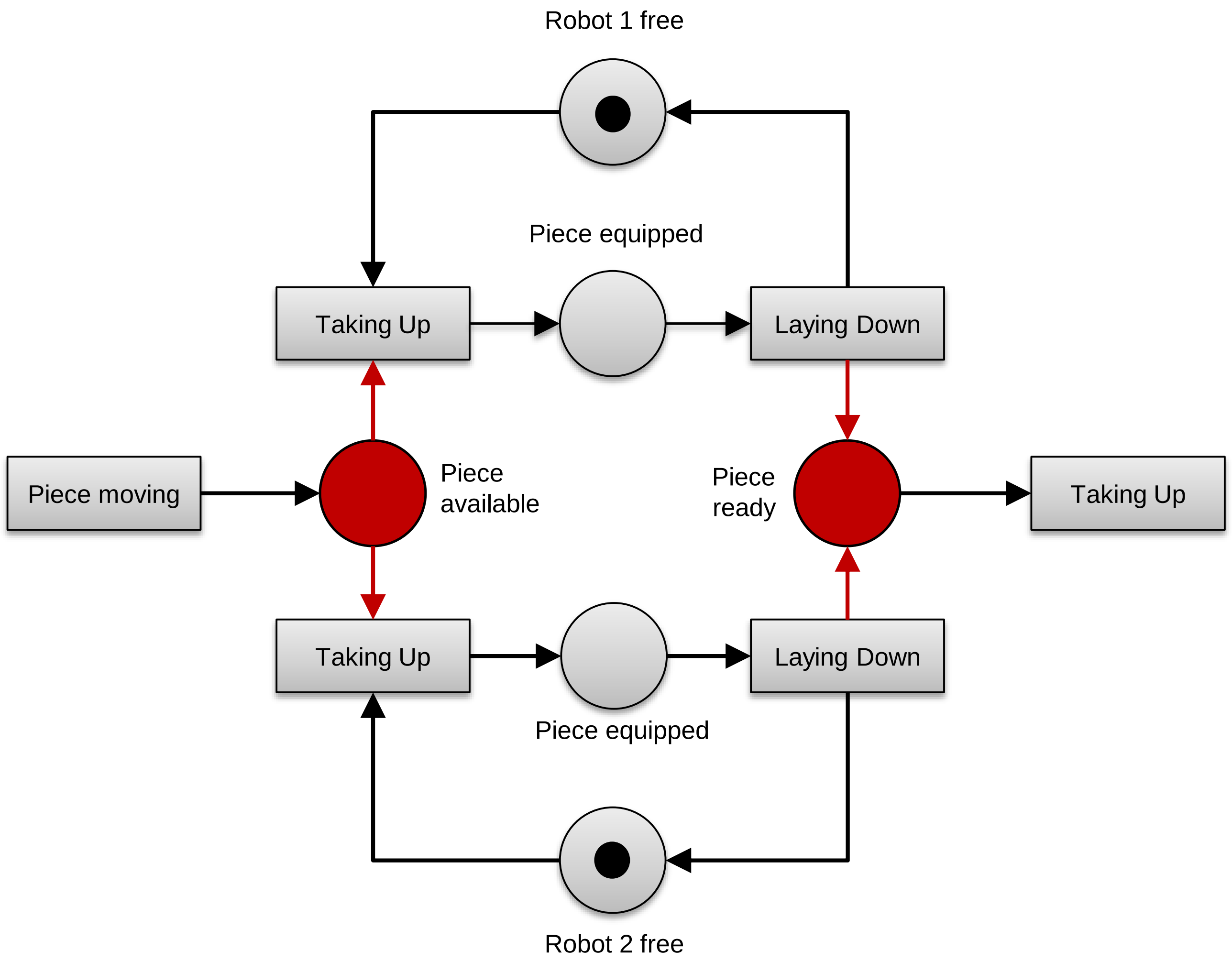
3.1.2.a Marked Graphs (MG)

- Is the production PN a MG ?



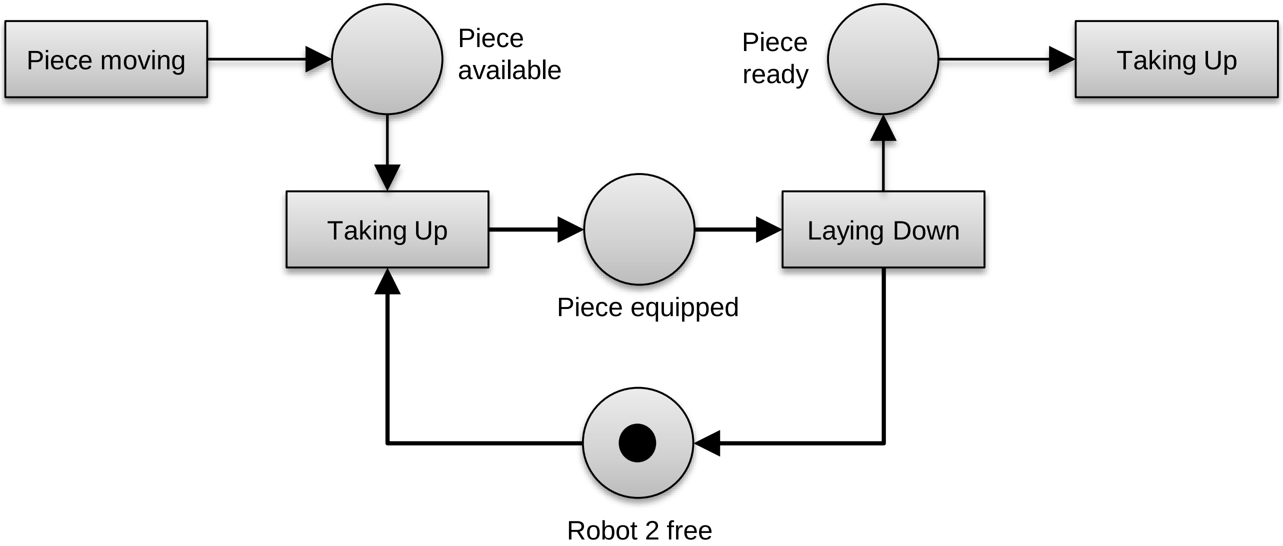
3.1.2.a Marked Graphs (MG)

- The production PN is no MG
  → Some places have more than 1 incoming/outgoing arc



3.1.2.a Marked Graphs (MG)

- However, the production robot PN is a MG

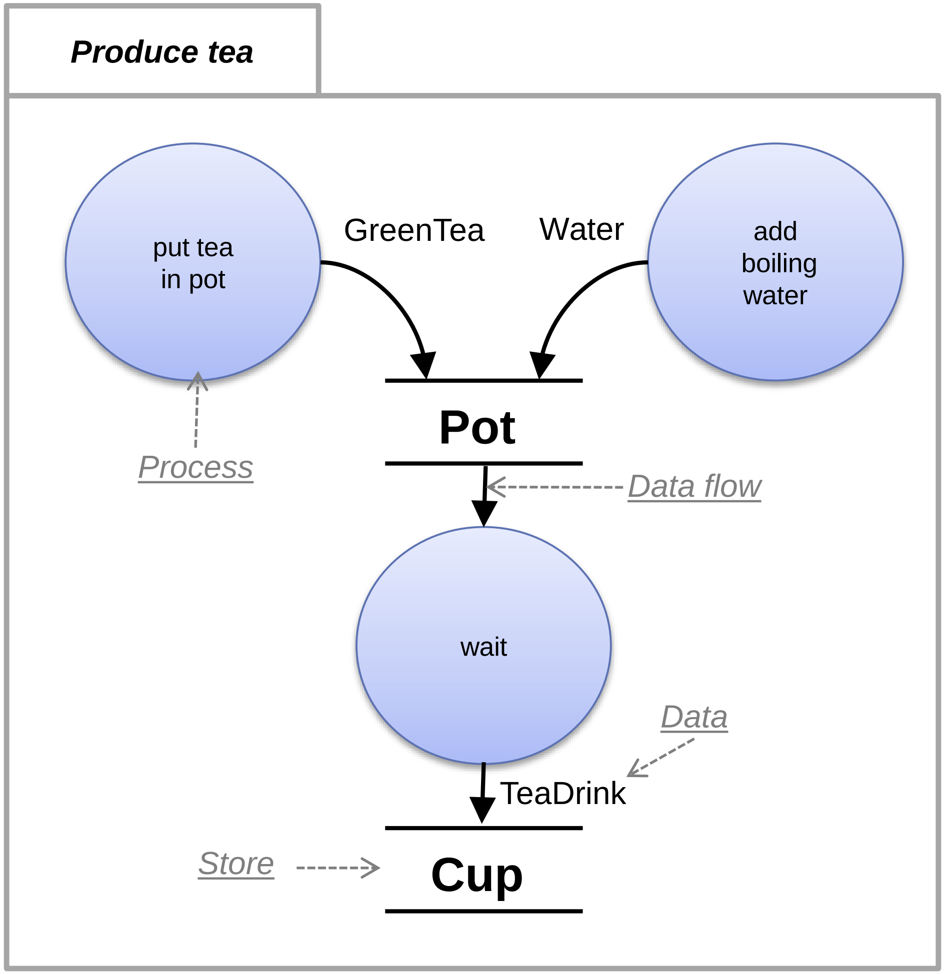


More General Data-Flow Diagrams

- General DFD without restriction can be modeled by PN, too.
  - However, places cannot be abstracted
  - They correspond to stores with 2 feeding or consuming processes
- Example: the full robot has places with 2 ingoing or outgoing edges,
  - They cannot be abstracted

For DFD, Many Notations Exist

- Notation from Structured Analysis [Balzert]



3.1.2.b State Machines are PN with Cardinality Restrictions

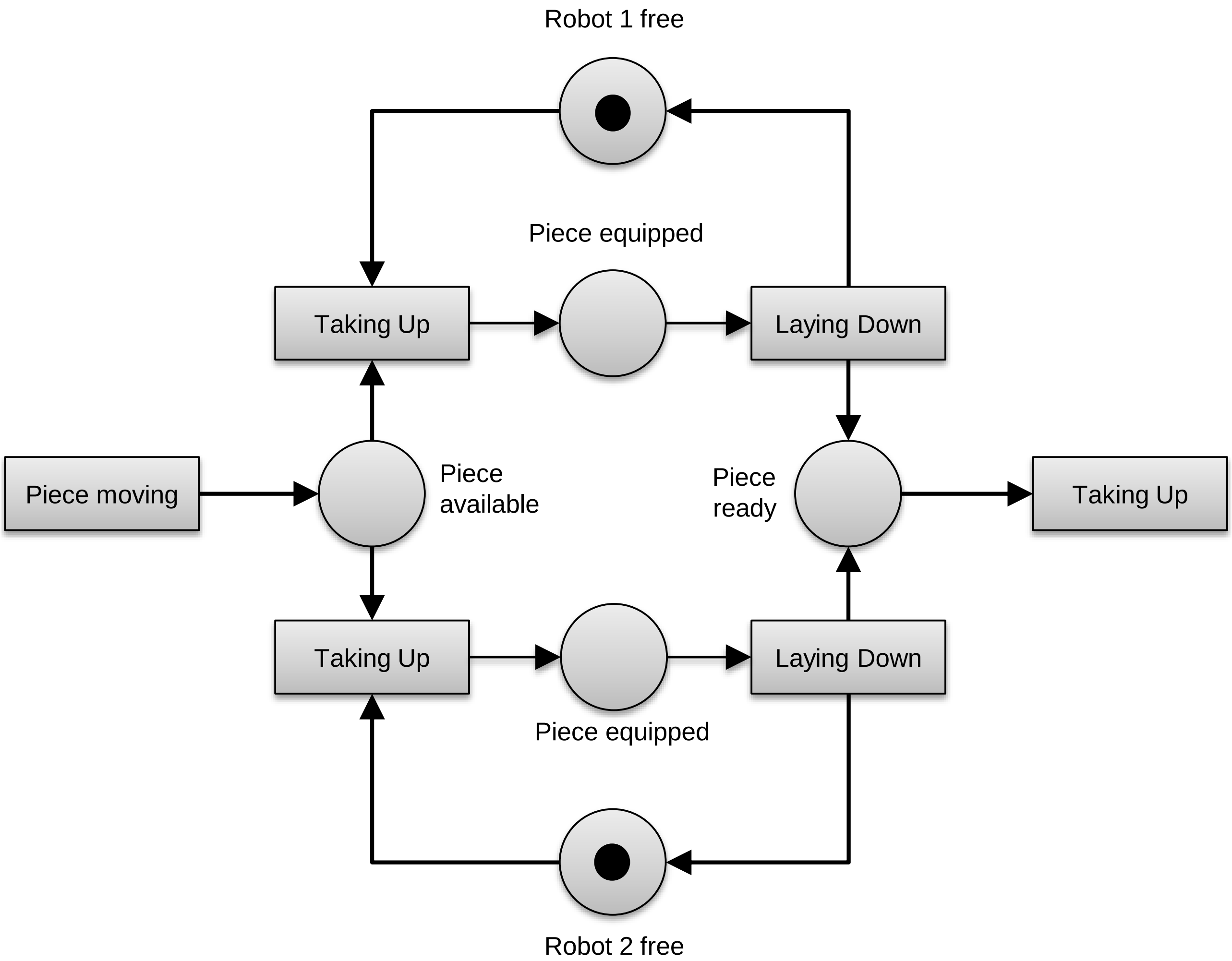
- A **Finite State Machine PN** is an elementary PN such that:

  1. Each transition has only 1 incoming arc
  2. Each transition has only 1 outgoing arc

  - Then, it is equivalent to a finite automaton or a *statechart*
  - From every class-statechart that specifies the behavior of a class, a State Machine can be produced easily
    - Flattening the nested states
  - Transitions correspond to transitions in statecharts, states to states
  - Transitions can be *merged* with the ingoing and outcoming arcs
  - In a FSM there is only one token
- All theory for CPN holds for Statecharts, too

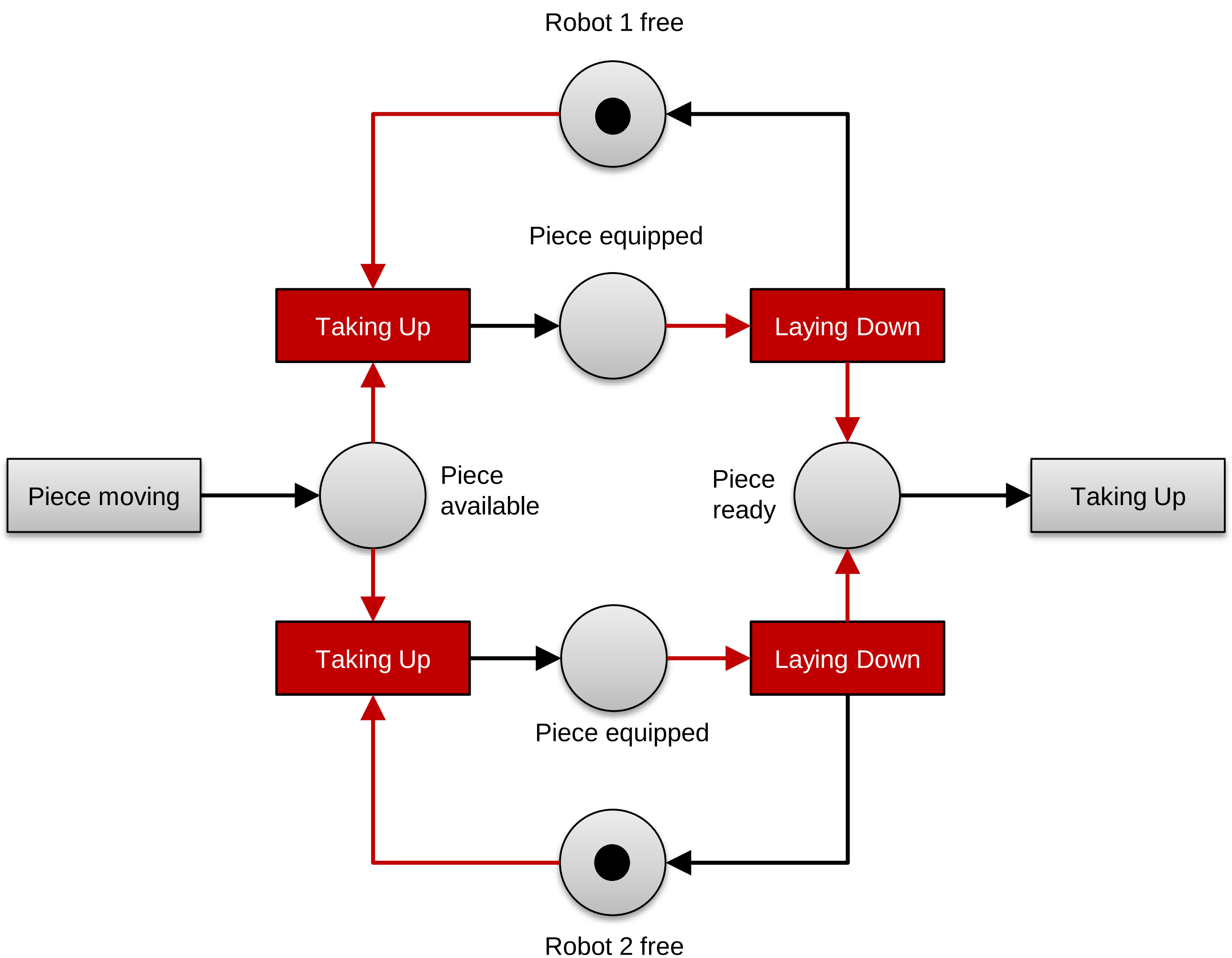
3.1.2.b State Machines

- Is the production PN a FSM ?



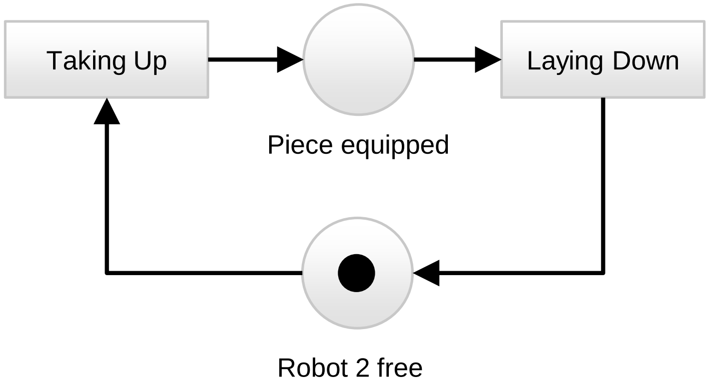
3.1.2.b State Machines

- The production PN is no FSM
  → Some transitions have more than 1 incoming/outgoing arc



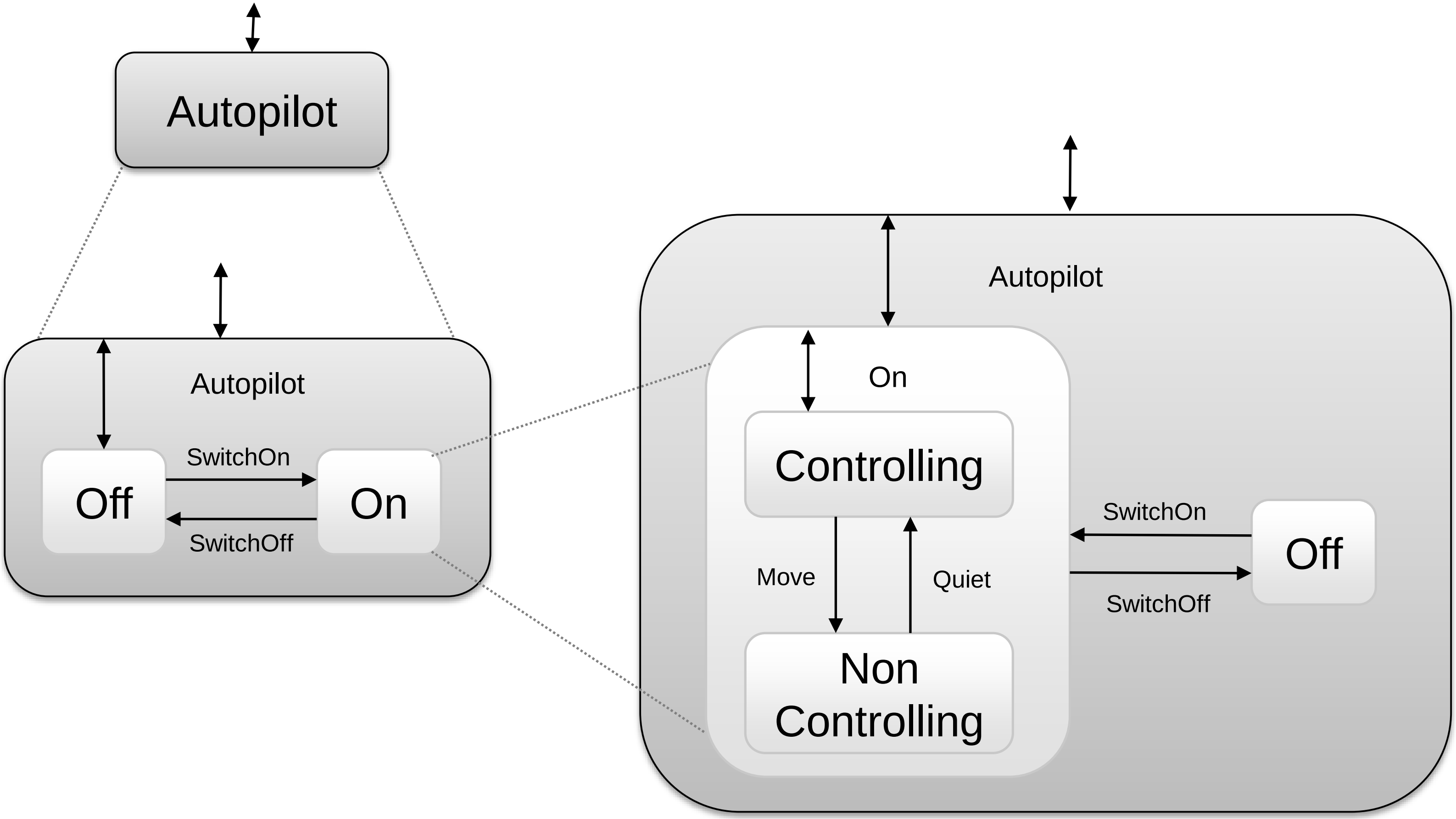
3.1.2.b State Machines

- One Robot is a FSM but not with incoming/outgoing arc



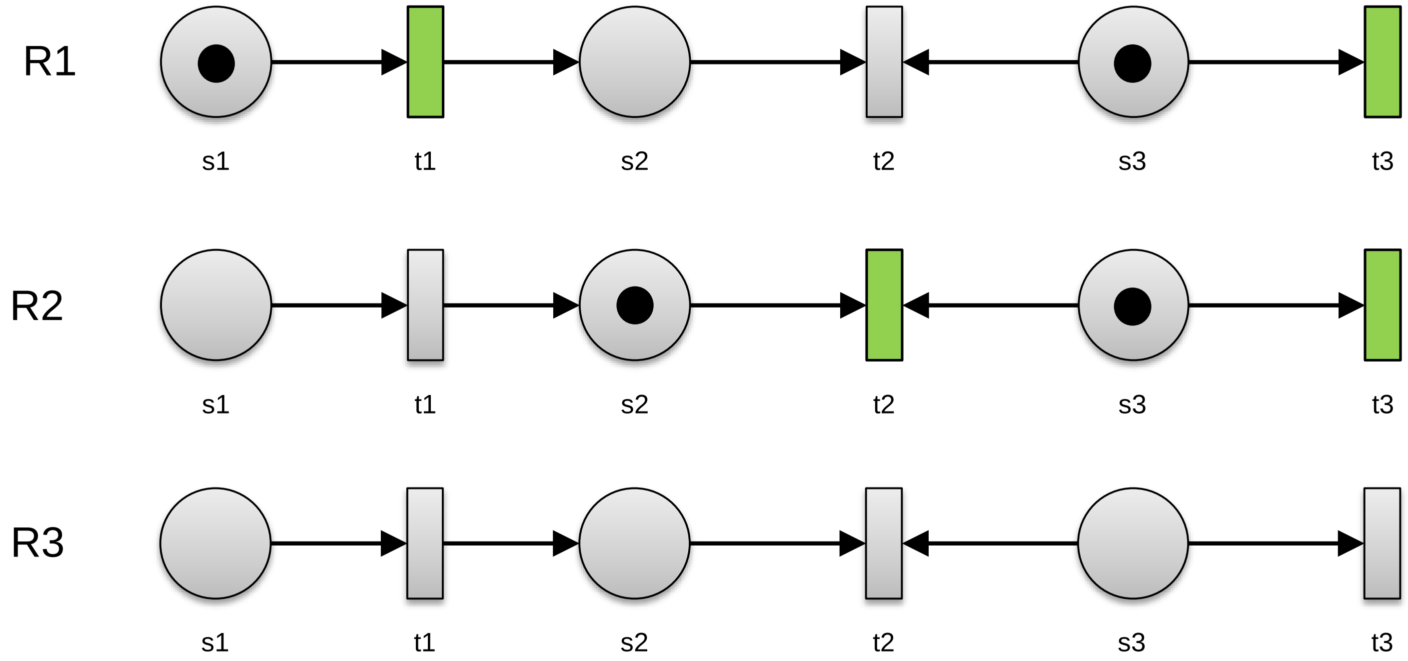
Hierarchical StateCharts from UML

- States can be nested in StateCharts
- This corresponds to hierarchical StateMachine-PN, in which states can be refined and nested



3.1.2.c Free-Choice Nets

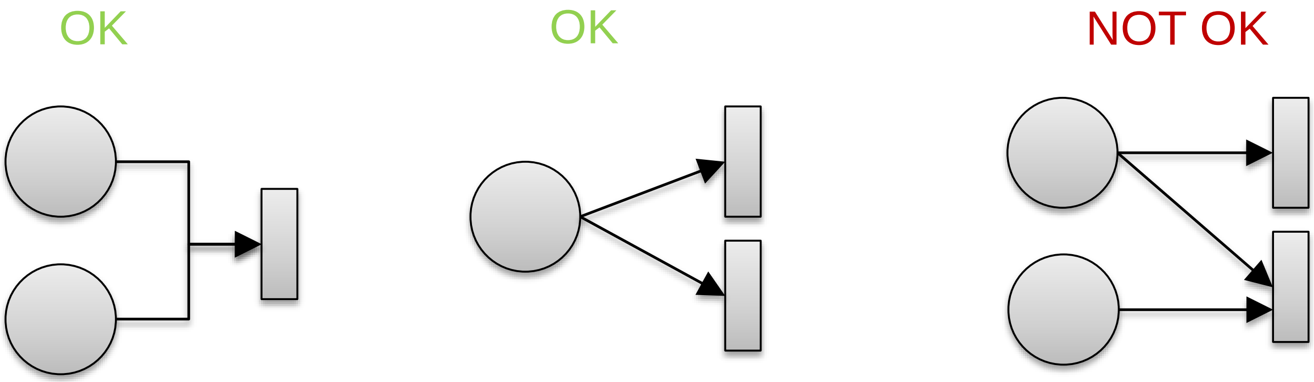
- Two transitions are in conflict if the firing of one transition deactivates another
  - R1: no conflicts (t1 and t3 activated) → in this example t1 fires
  - R2: t2 and t3 are in conflict → in this example t2 fires
  - R3: t3 is deactivated because of t2



3.1.2.c Free-Choice Nets

- **Free-Choice Petri Net** provides deterministic parallelism
  - Choice between transitions never influence the rest of the system („free choice")
  - Rule conflicts out
  - AND-splits and AND-joins

- Keep *places with more than one output transitions* away from *transitions with more than one input places* (forbidden are "side actions")
  - outdegree(place) → in(out(place)) = {place}



3.1.3 Colored Petri Nets as Example of High Level Nets

Modularity

Refinement

Reuse

Preparing “reducible graphs”

Colored Petri Nets, CPN

- Colored (Typed) Petri Nets (CPN) refine Petri nets:
  - Tokens are typed (colored)
  - Types are described by data structure language (e.g.,Java, ML, UML class diagrams, data dictionaries, grammars)
  - Concept of time can be added
- Full tool support
  - Fully automated code generation in Java and ML (in contrast to UML)
  - Possible to proof features about the PN
  - Net simulator allows for debugging
- Much better for safety-critical systems than UML, because proofs can be done

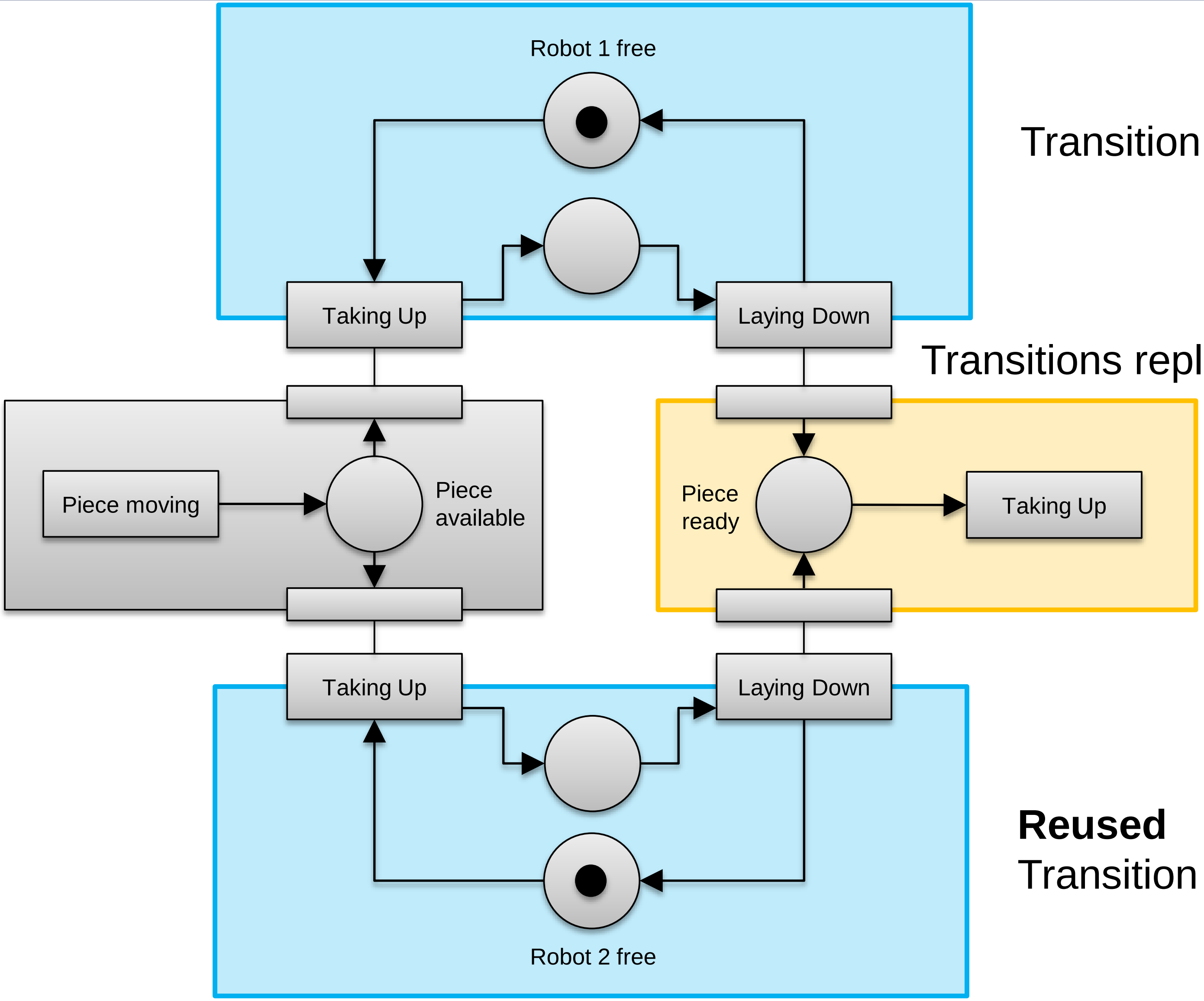
Annotations in CPN

- Places are annotated by
  - Token types
    `(STRING x STRING)`
  - Markings of objects and the cardinality in which they occur:
    `2'("Uwe","Assmann")`
- Edges are annotated by
  - Type variables which are unified by unification against the token objects
    `(X,Y)`
  - Guards
    `[ X == 10]`
  - If-Then-Else statements
    `if X < 20 then Y := 4 else Y := 7`
  - Switch statements
  - Boolean functions that test conditions

CPN are Modular

- A **subnet** is called a *page (module)*
  - Every page has ports
  - Ports mark in- and out-going transitions/places
- **Transition page**: interface contains transitions (transition ports)
- **Place page** (state page): interface contains place (place ports)
- **Net class**: a named page that is a kind of ”template” or ”class”
  - It can be instantiated to a net ”object”
- Reuse of pages and templates possible
  - Libraries of CPN ”procedures” possible

Robots with Transition Pages, Coupled by Transition Ports

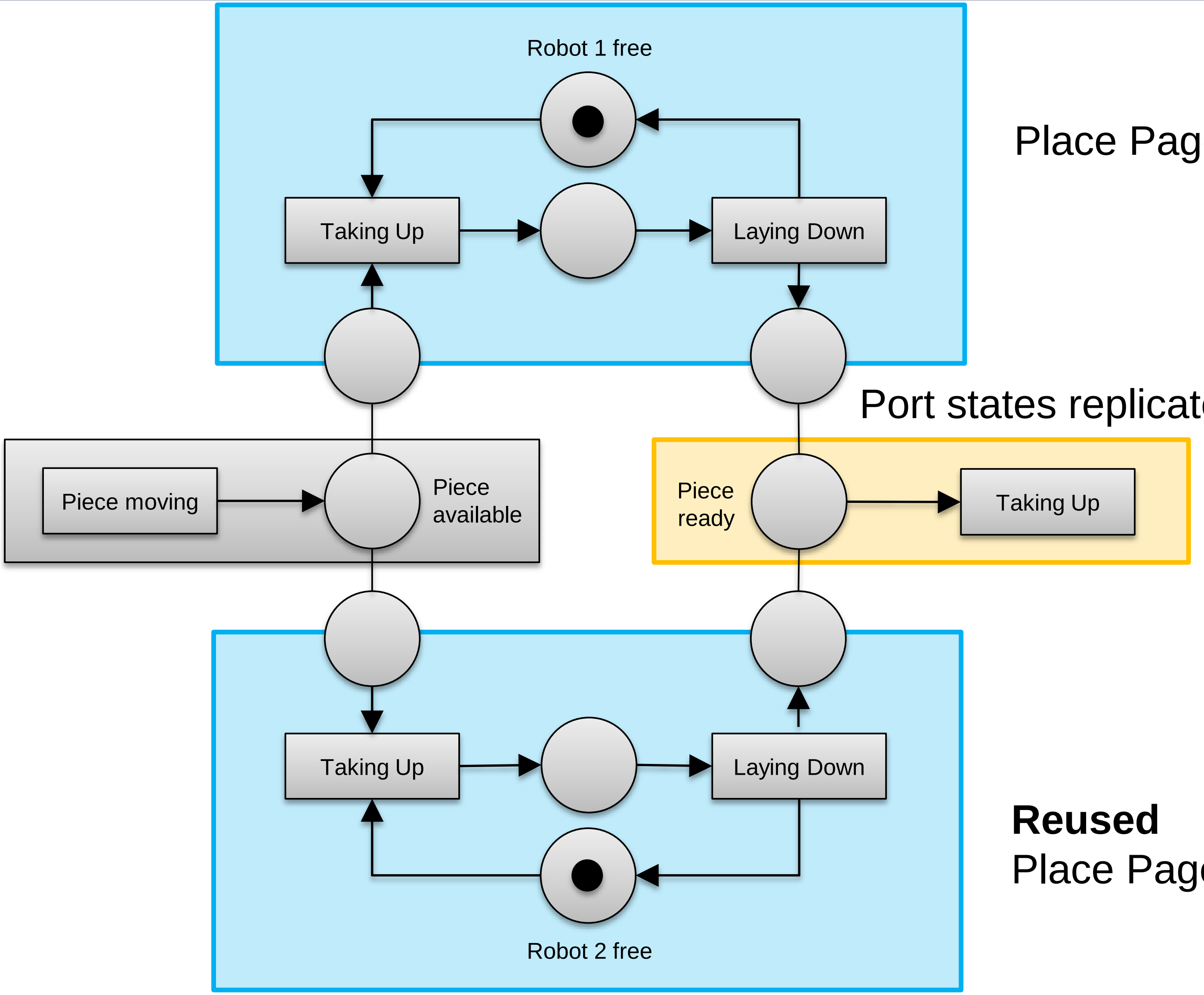


Transition Page

Transitions replicated

Reused
Transition Page

Robots with Place (State) Pages, Coupled by Replicated State Ports



Place Page

Port states replicated

Reused
Place Page

CPN are Hierarchical

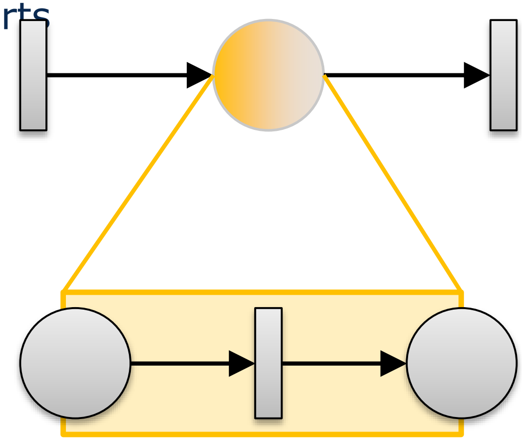
- Places and transitions may be hierarchically **refined**
  - Two pointwise refinement operations:
    - Replace a transition with a transition page
    - Replace a state with a state page
  - Refinement condition: Retain the embedding (embedding edges)

- CPN can be arranged as hierarchical graphs (reducible graphs, see later)
  - Large specifications possible, overview is still good
  - Subnet stemming from refinements are also place or transition pages

Point-wise Refinement Example

Pointwise refinement:

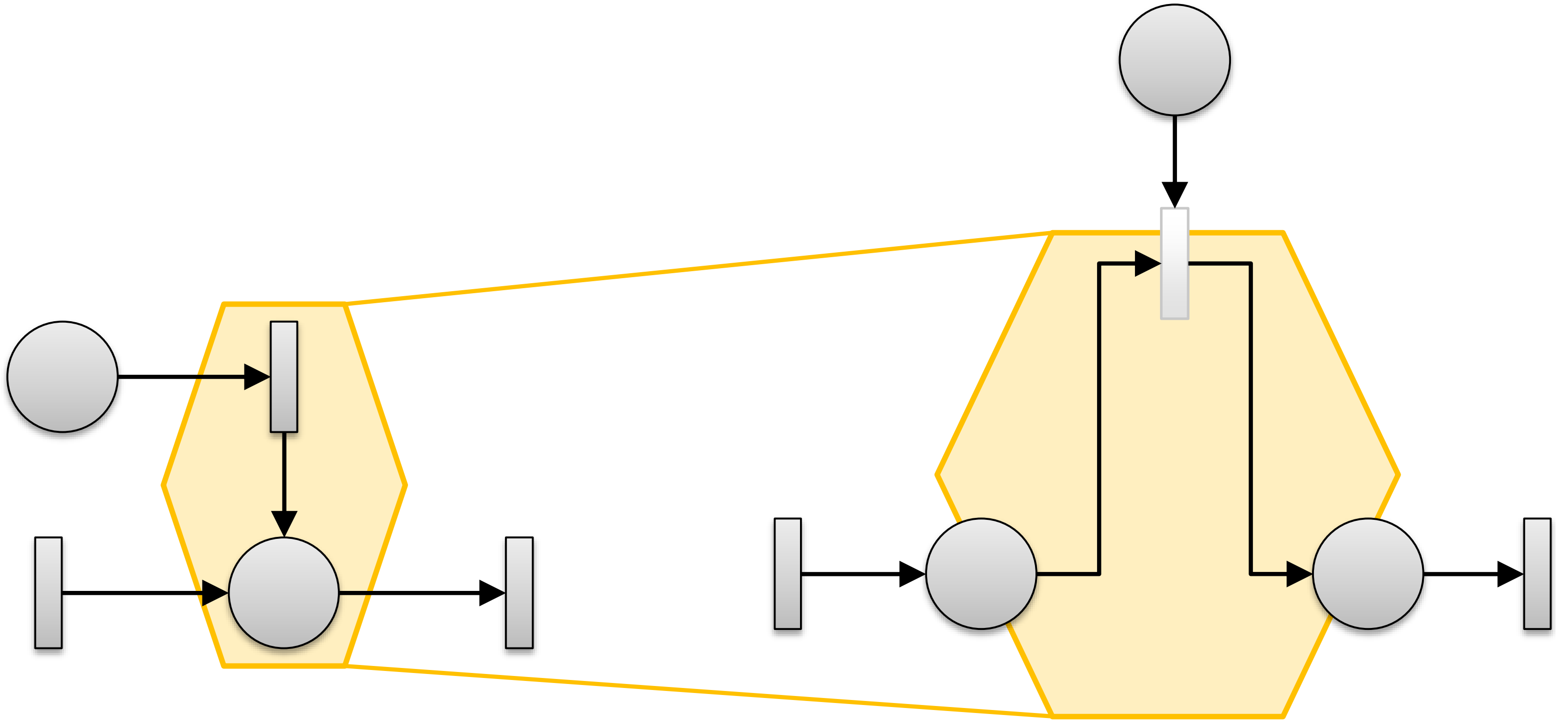
- *Transition refining page*: refines a transition, transition ports
- *Place refining page (state refining page)*:
  refines a place, place ports



Point-wise Refinement Example

Hyperedge refinement:

- Hyperedges and regions in PN can be refined



Modularity is Important for Scaling – Industrial Applications of CPN

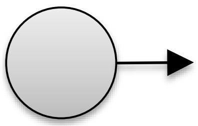
- Large systems are constructed as reducible specifications
  - They have 10-100 pages, up to 1000 transitions, 100 token types
- Example: ISDN Protocol specification
  - Some page templates have more than 100 uses
  - Corresponds to millions of places and transitions in the expanded, non-hierarchical net
  - Can be done in several person weeks

3.2 Patterns in and Transformations of Petri Nets

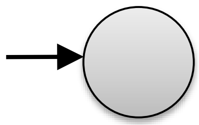
- Petri Nets have a real advantage when parallel processes and synchronization must be modelled
  - Many concepts can be expressed as *PN patterns* or with *PN complex operators*

- Analyzability: Petri Nets can be analyzed for patterns (by pattern matching)
- Transformation: Petri Nets can be simplified by automatic transformations

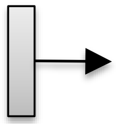
Simple PN Buffering Patterns



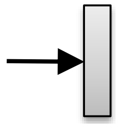
Reservoir Place
Does not generate objects



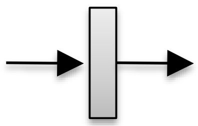
Archive
Stores objects.
Can be k-bounded



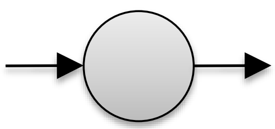
Permanently active transaction
Generates objects
(Object source, Event source)



Sink
Deletes/Destroys objects



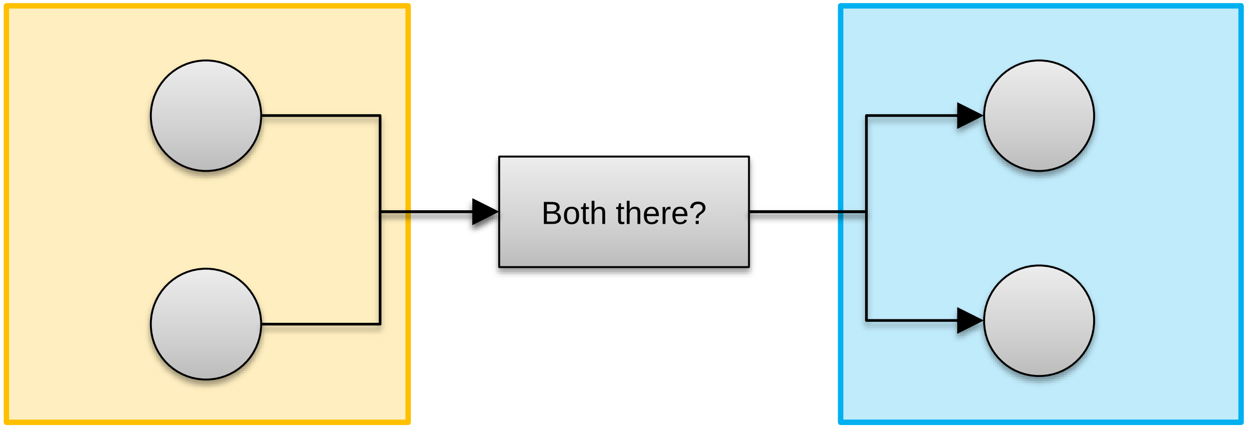
Process
Sequential



Intermediate Archive
Buffer

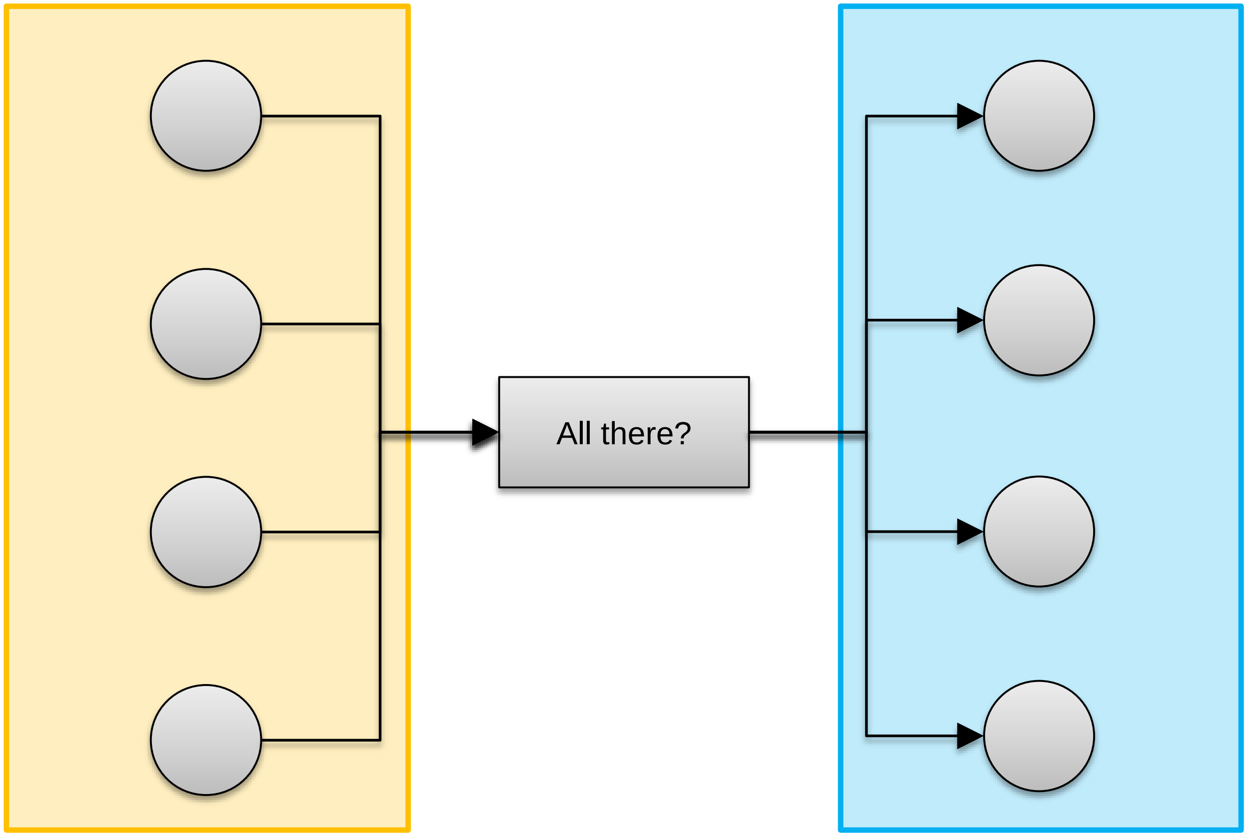
Patterns for Synchronization (Barrier)

- Coupling processes with parallel continuation



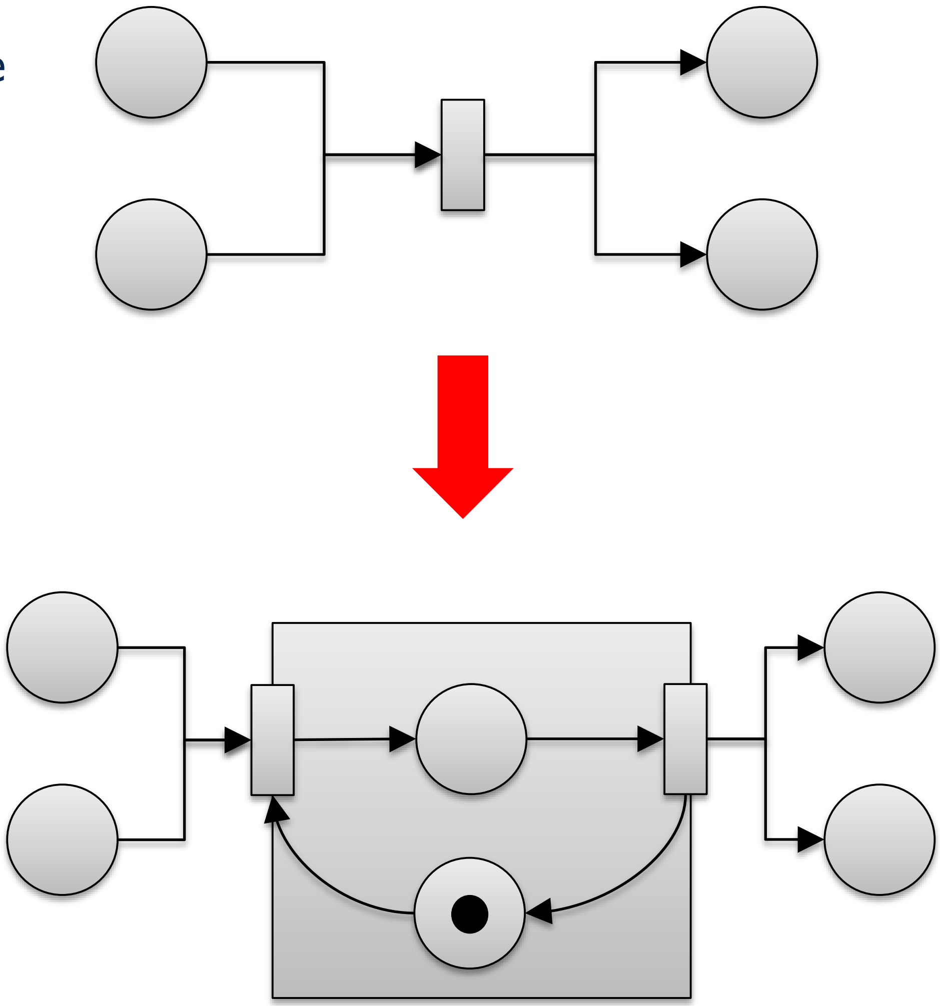
Patterns for Synchronization (n-Barrier)

- Bridges: Transitions between phases



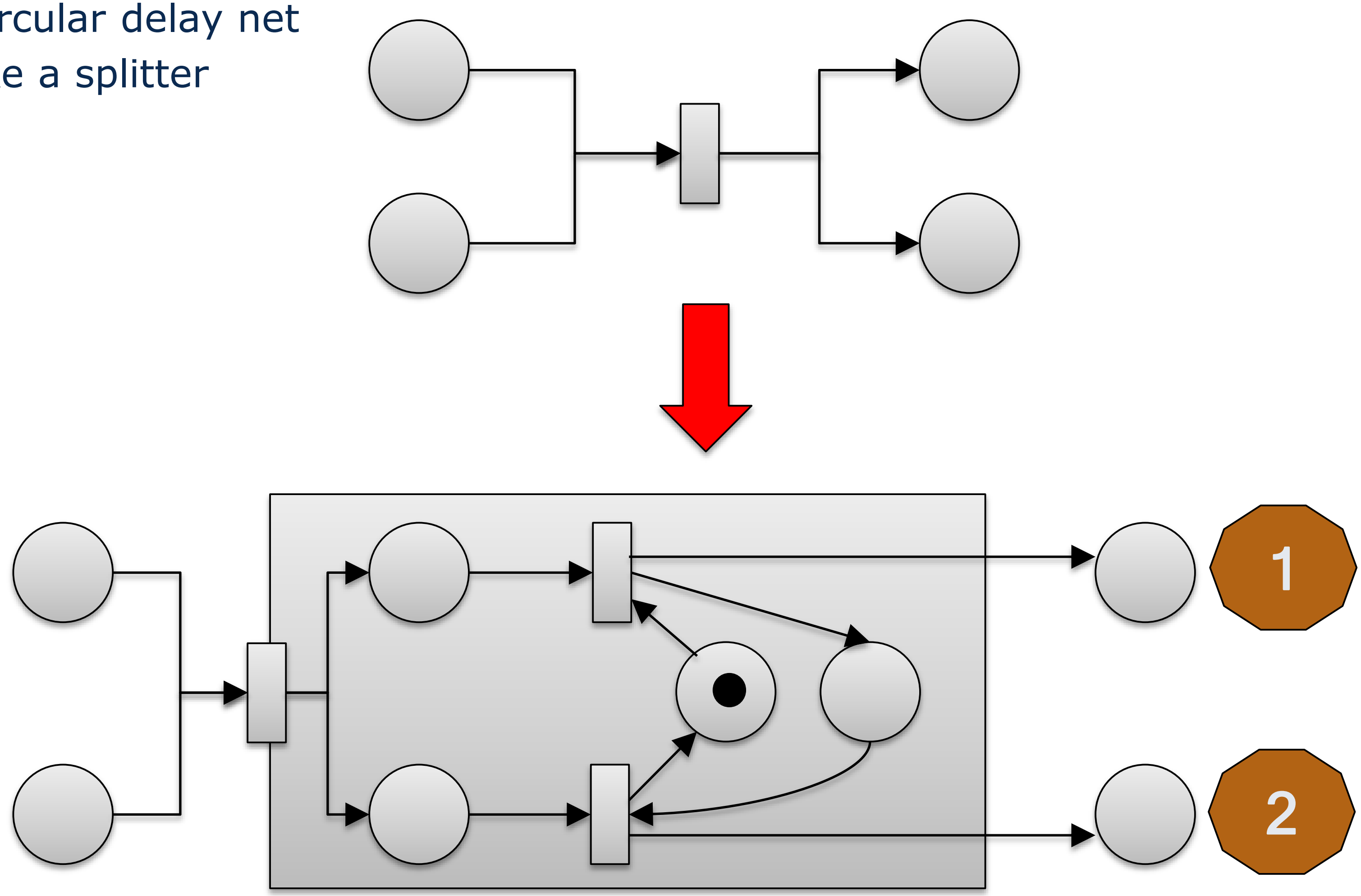
Adding Delays in Transitions by Feedback Loops

- Adding a **delay token**
- Behaves like a semaphore (lock – unlock critical region)



Adding Delays in Transitions by Feedback Loops

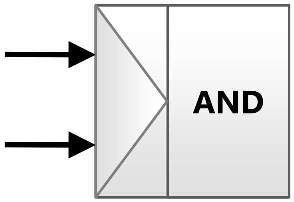
- Adding a circular delay net
- Behaves like a splitter



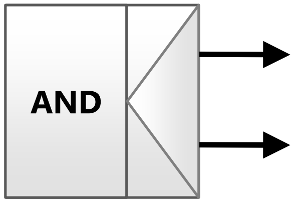
Simpler Specification with Special Operators (Transitions) in Workflow Nets

- In languages for Workflow nets, such as
  - ARIS workflow language
  - YAWL Yet another workflow language
  - BPMN Business Process Modeling Notation
  - BPEL Business Process Execution Language
- Specific transitions have been designed (specific operators) for simpler specification

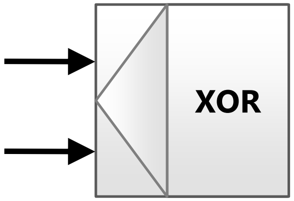
Complex Transition Operators in Workflow Nets: Join and Split Operators of YAWL



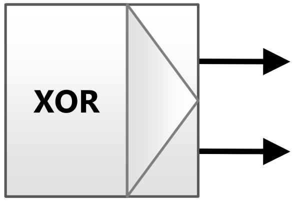
AND-Join
All ingoing places
are ready
(conjuctive input)



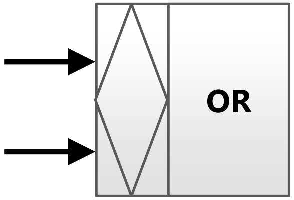
AND-Split
All outgoing places
are filled
(conjuctive output)



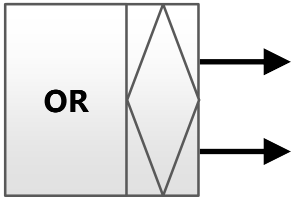
XOR-Join
Exactly one of n ingoing
places is ready
(disjunctive input)



XOR-Split
Exactly one of the outgoing
places are filled
(disjunctive output)



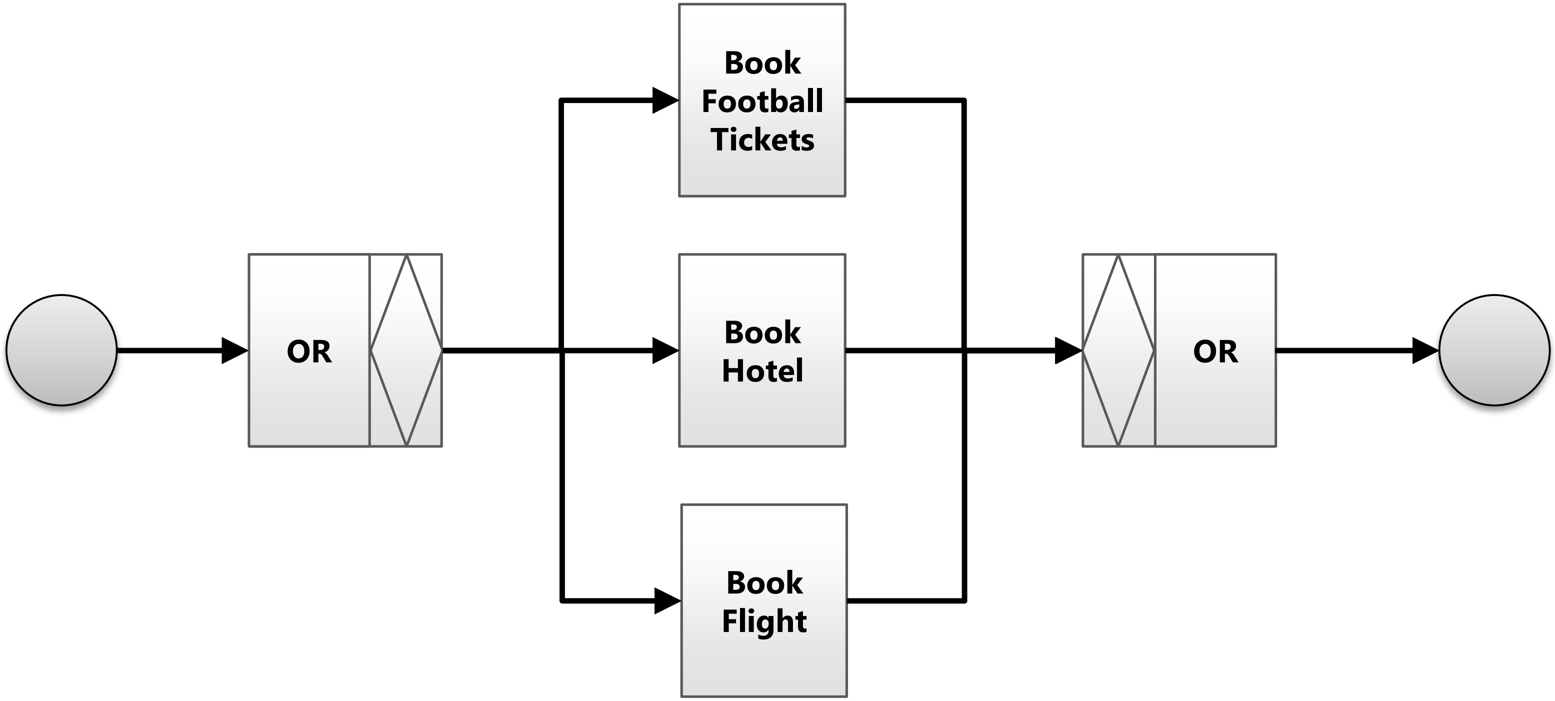
OR-Join
At least one of n
ingoing places is ready
(selective input)



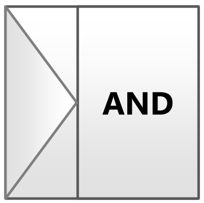
OR-Split (IOR-Split)
Some of the outgoing
places are filled
(selective output)

Simple YAWL example

- OR-Booking of travel activities
- Indeterministic choice possible

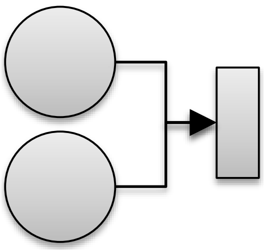


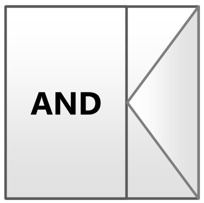
Parallelism Patterns – Transitional Operators



Joining Parallelism

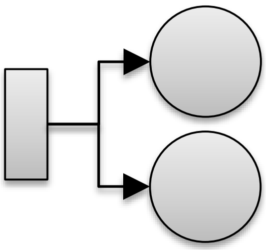
Synchronization Barrier
AND-Join

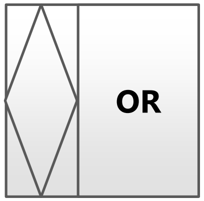




Replication and Distribution

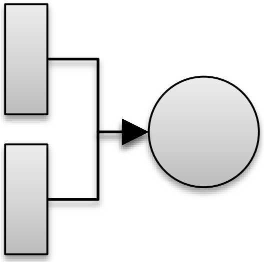
Forking
(AND-Split)

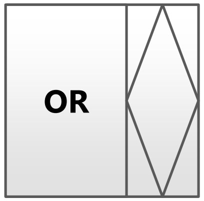




Collecting Objects

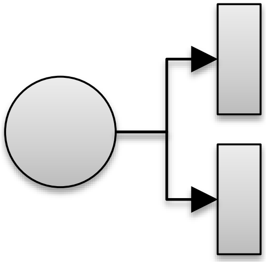
From parallel processes
OR-Join





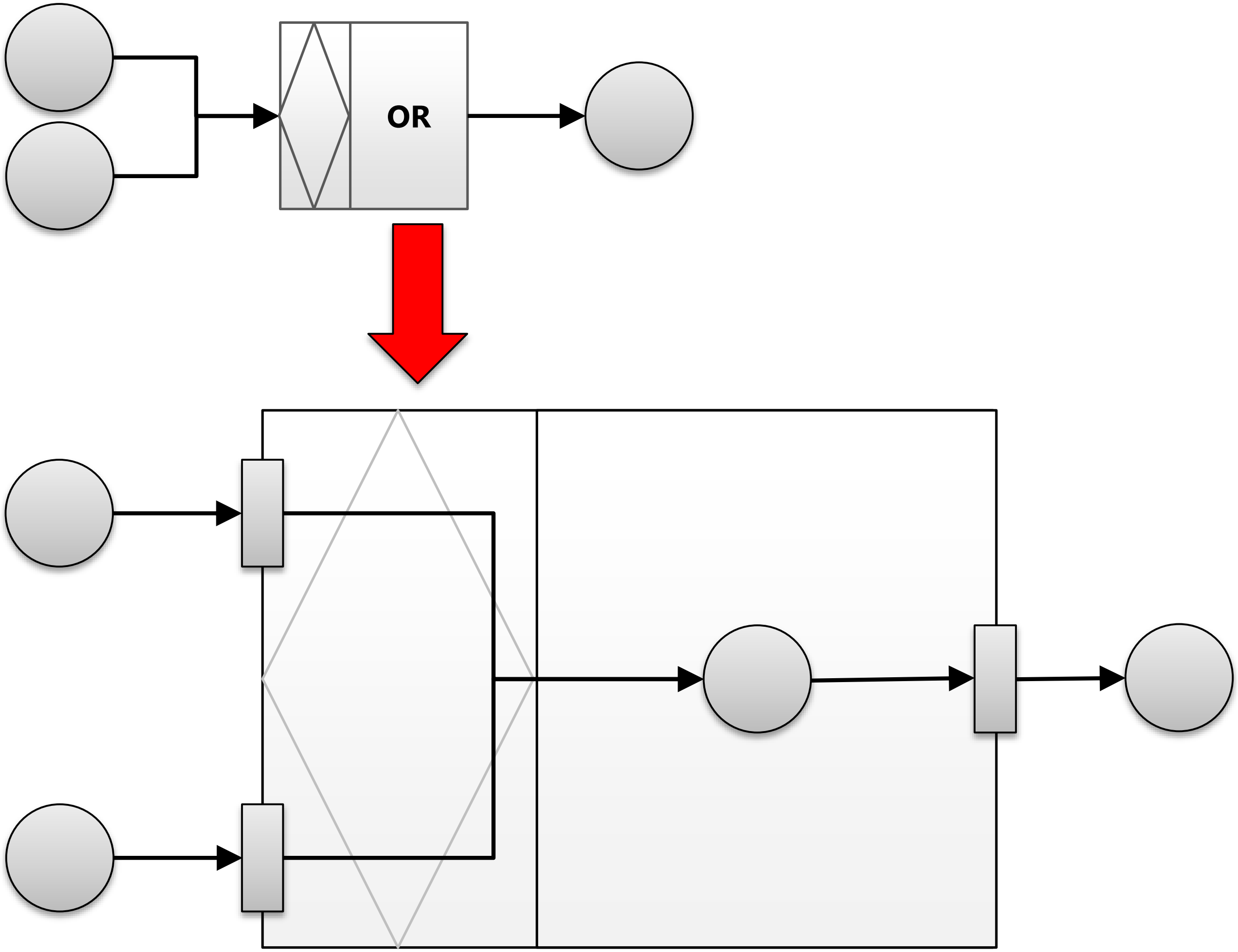
Decision

Indeterministically
(OR-Split)



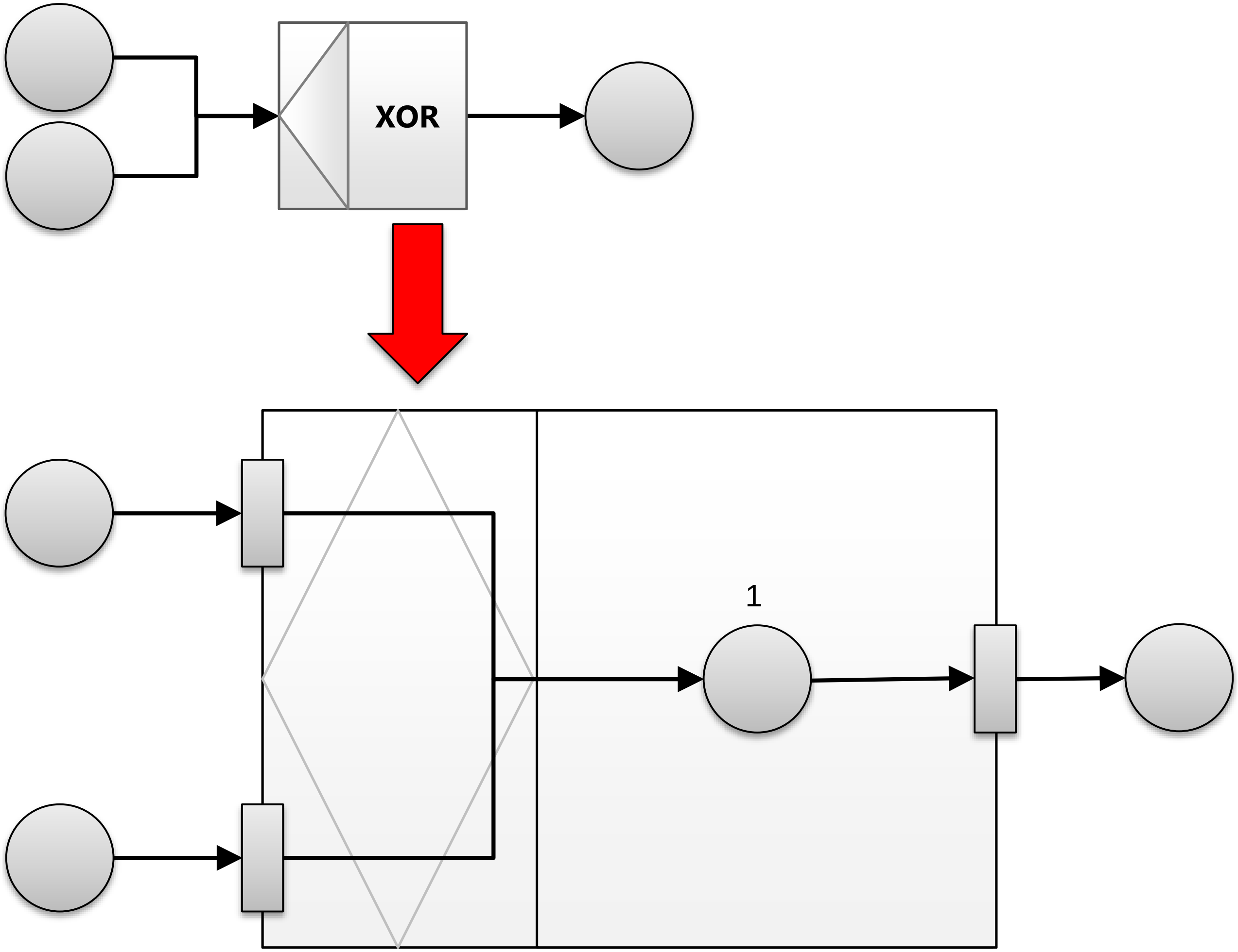
Example: Reduction Semantics of OR-Join Operator

- Complex operators refine to special pages with multiple transition ports

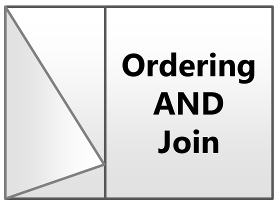


Example: Reduction Semantics of XOR-Join Operator

- XOR-Join with bound state (only 1 token can go into a place)

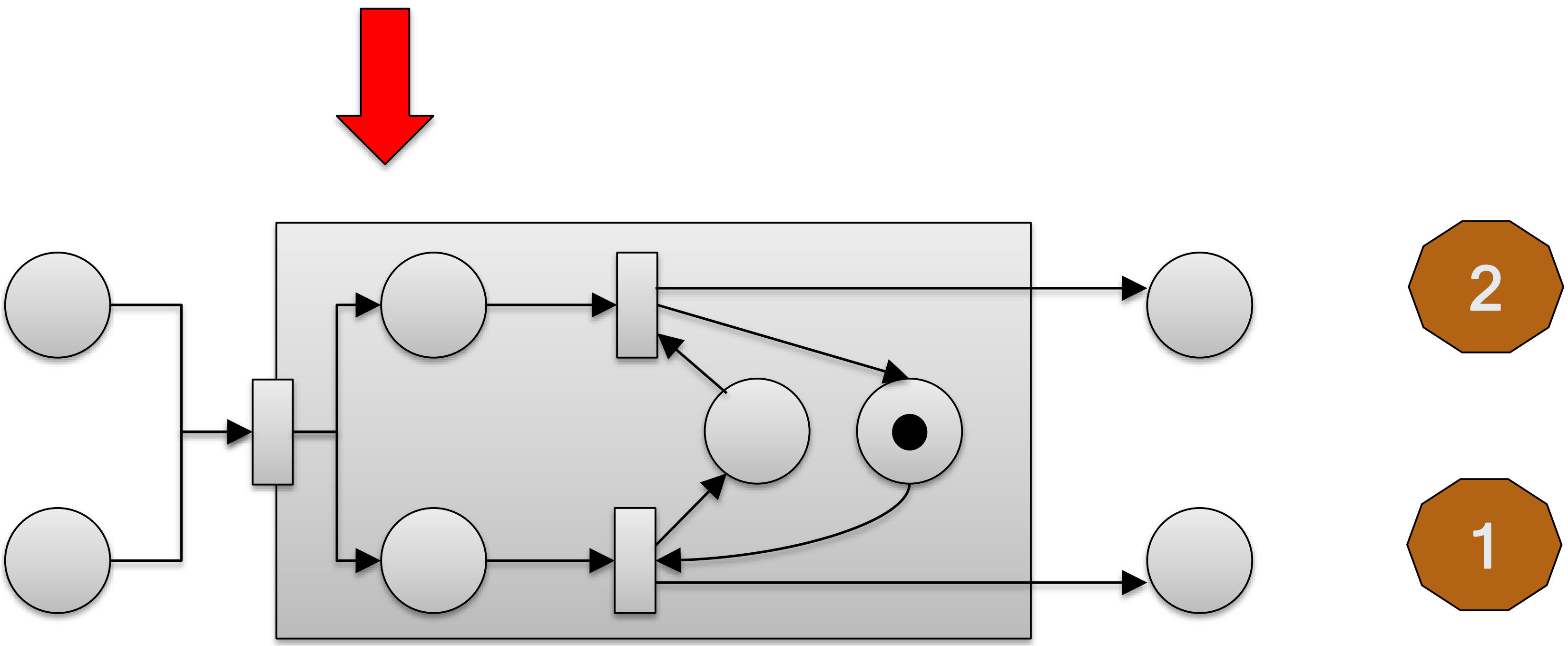


Parallelism Patterns – Transitional Operators (2)

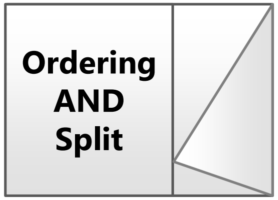


Ordering Synchronization Barrier

Ordering-AND-Join

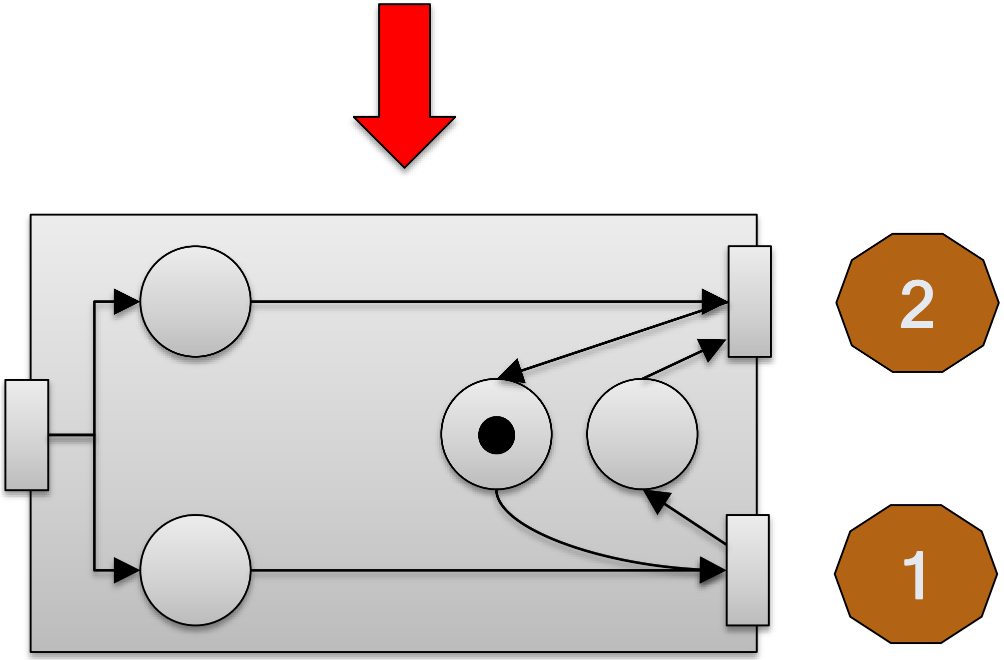


Parallelism Patterns – Transitional Operators (2)

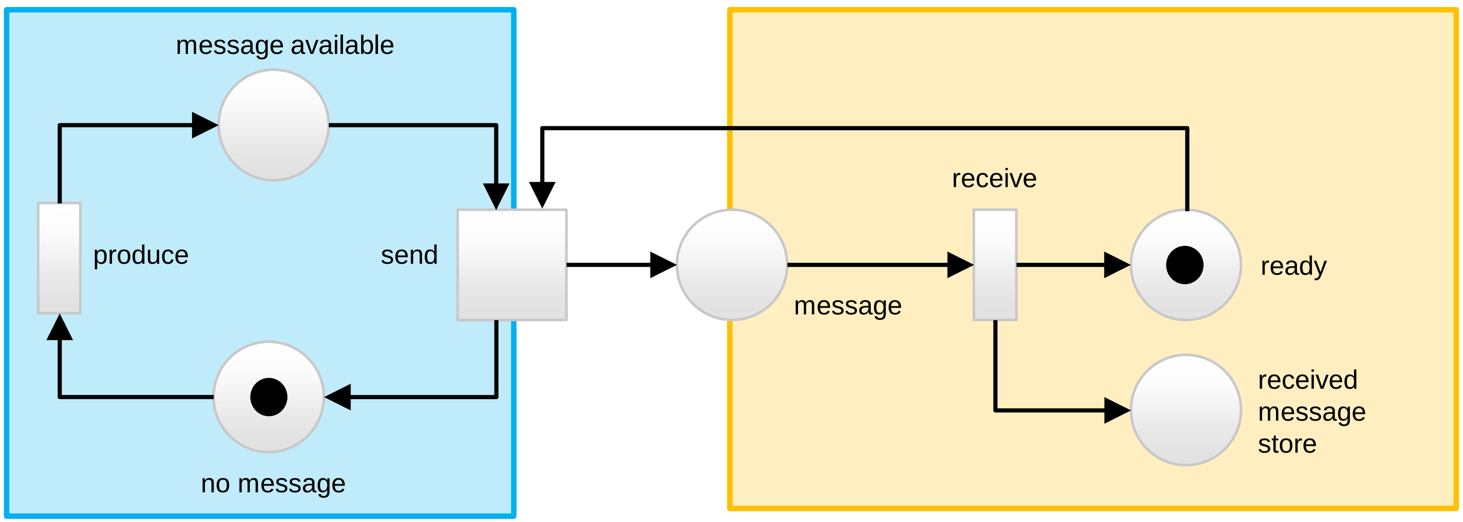


Output Ordering Generator

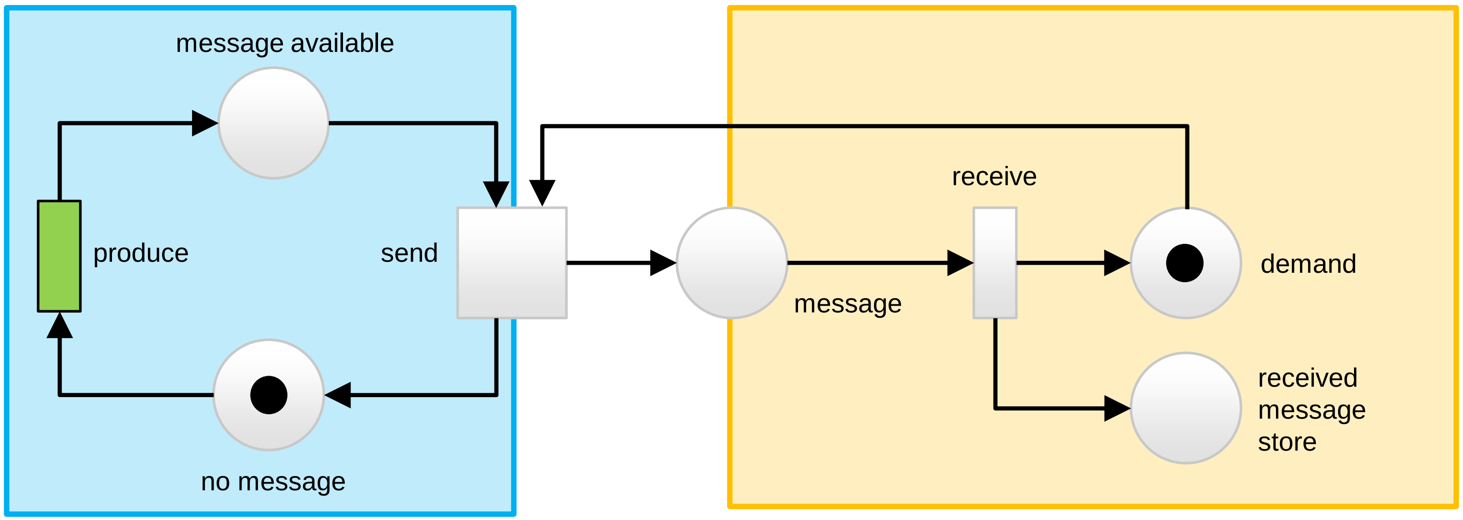
Ordering-AND-Split



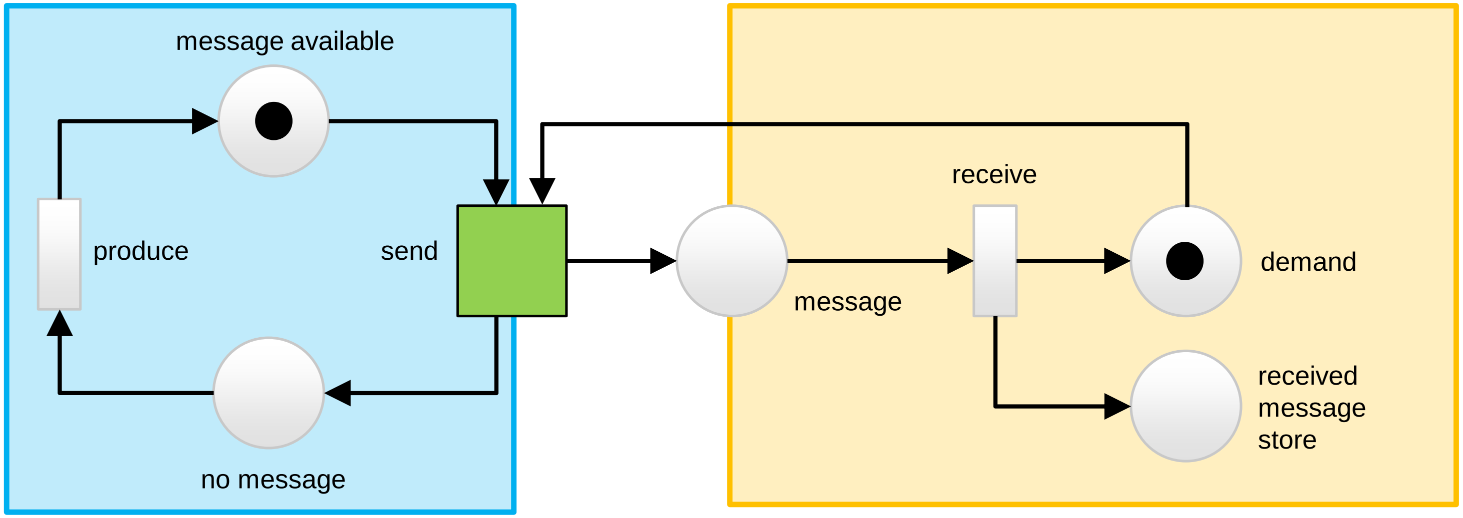
Patterns for Communication
Direct Producer-Consumer



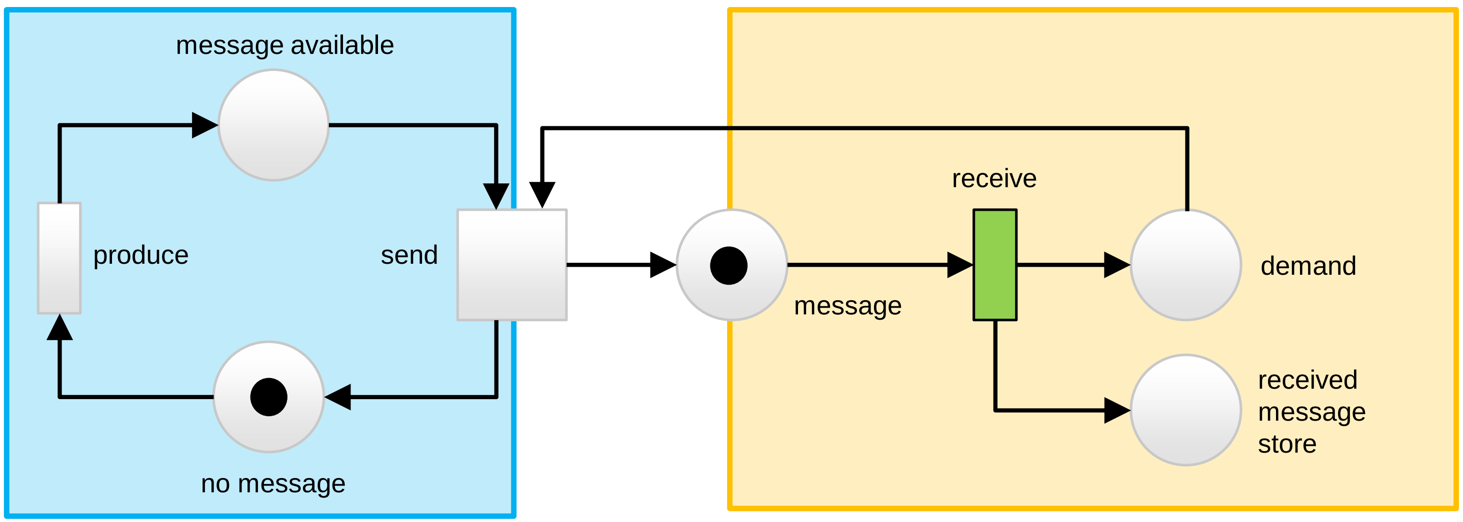
Patterns for Communication Direct Producer-Consumer



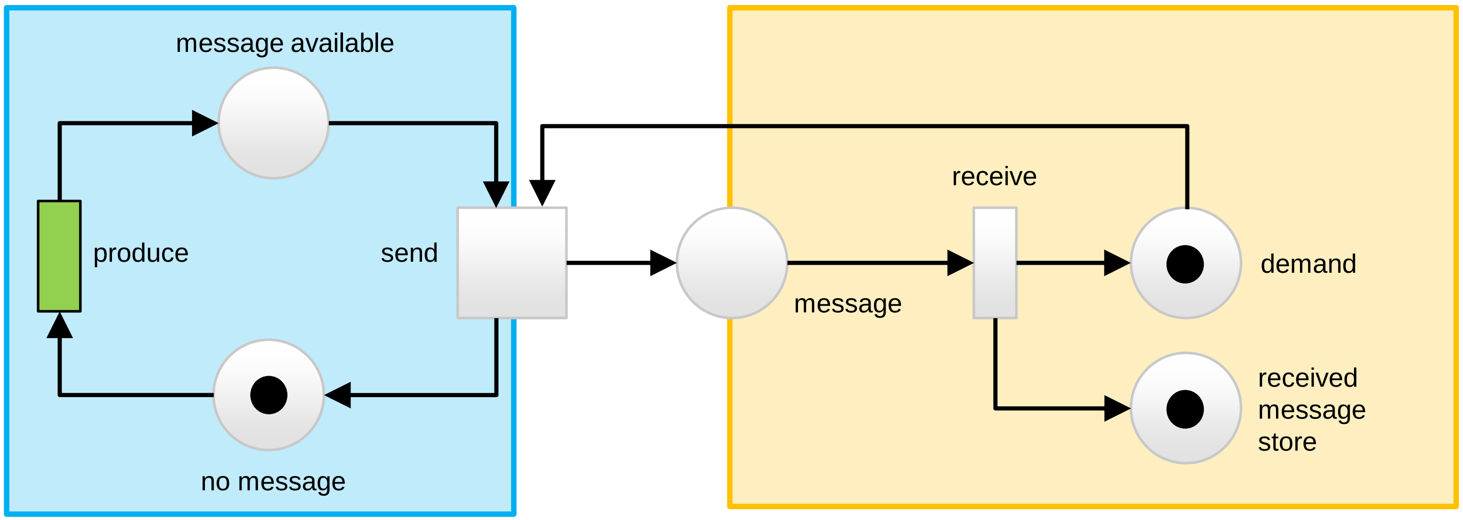
Patterns for Communication
Direct Producer-Consumer



Patterns for Communication
Direct Producer-Consumer

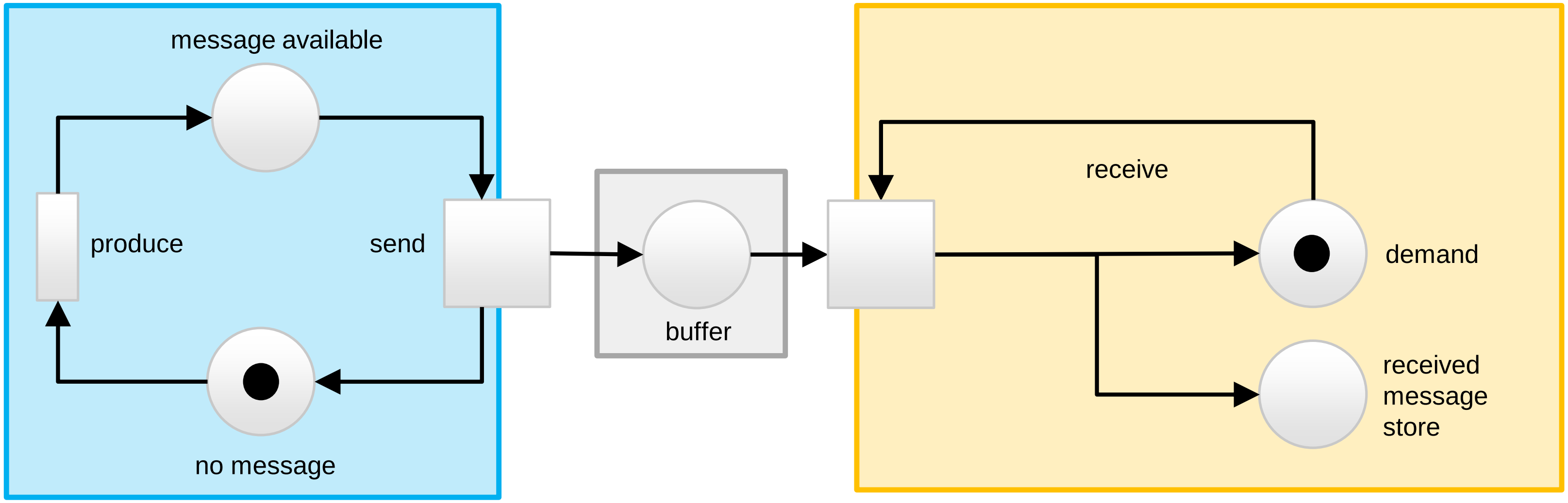


Patterns for Communication Direct Producer-Consumer



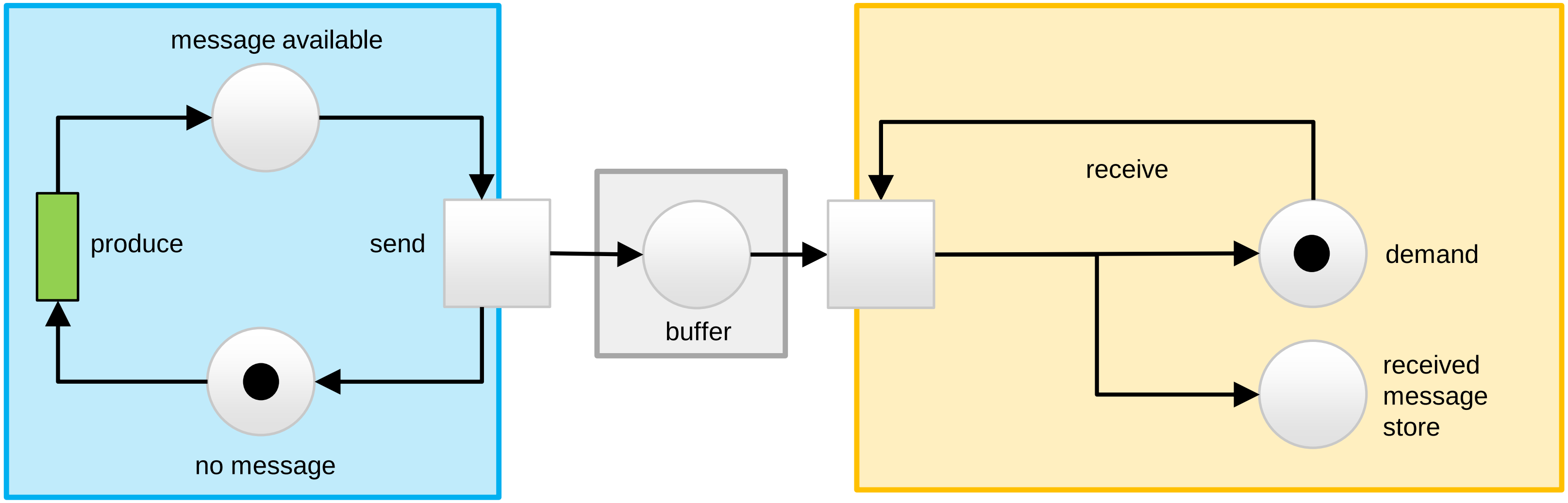
Patterns for Communication

- Producer Consumer with Buffer



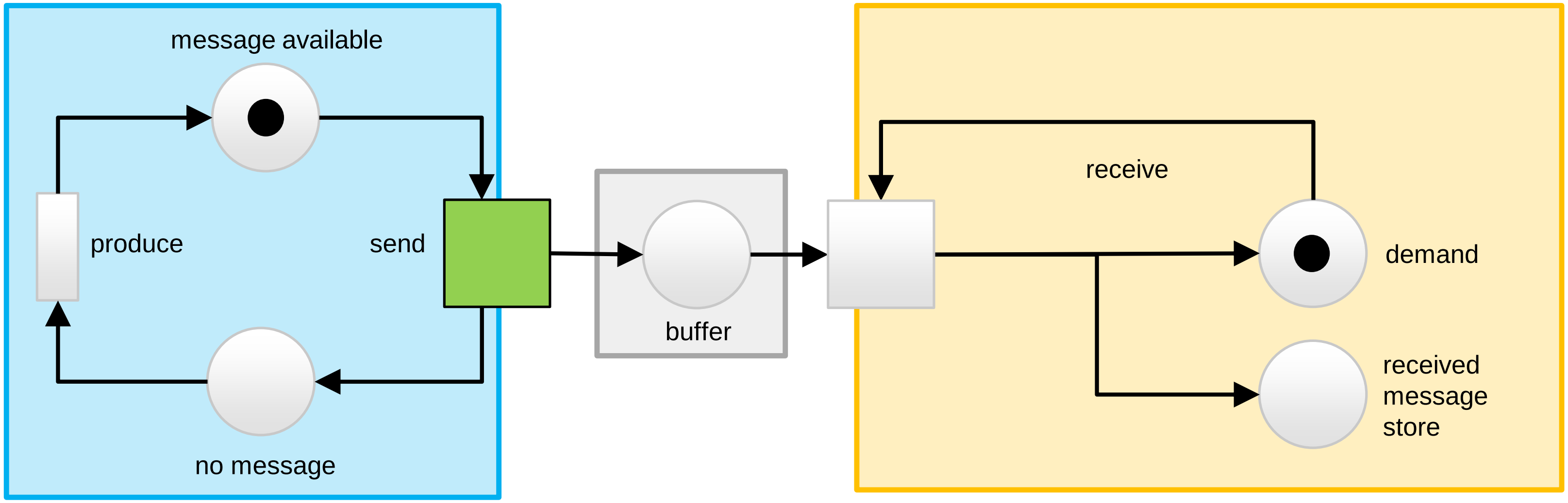
Patterns for Communication

- Producer Consumer with Buffer



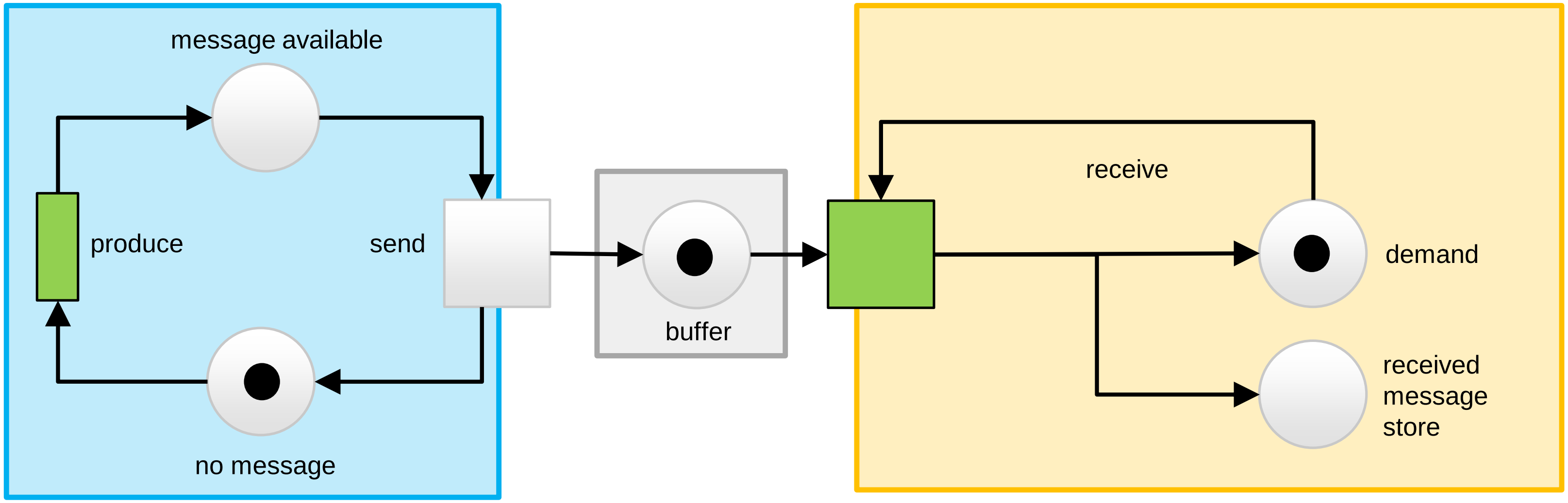
Patterns for Communication

- Producer Consumer with Buffer



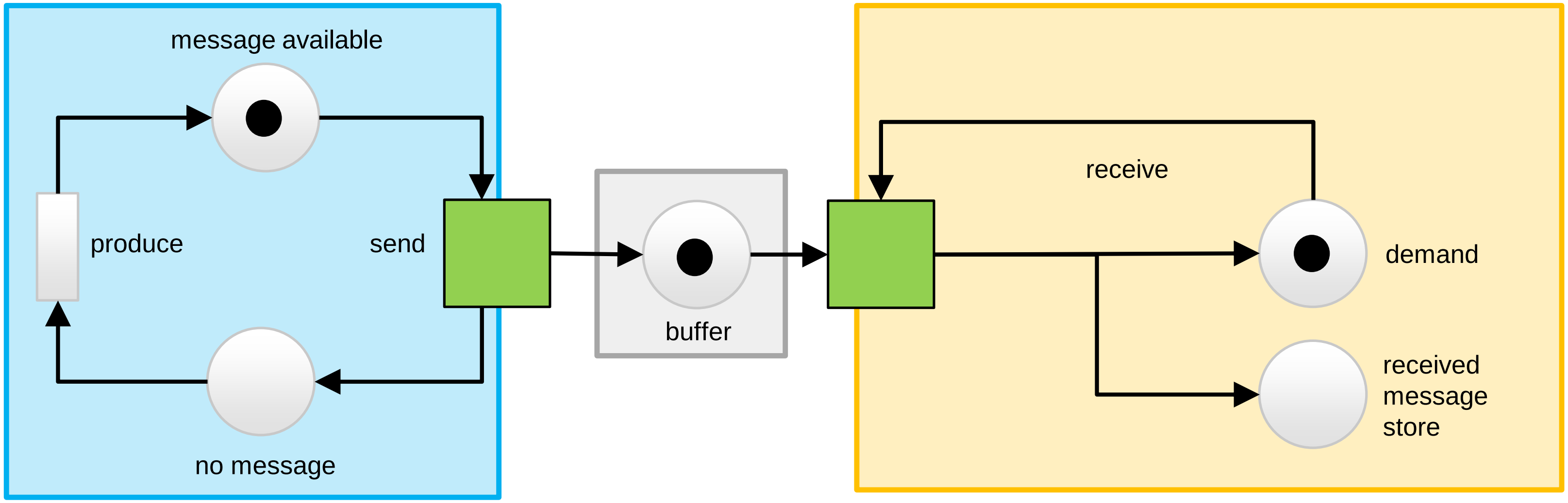
Patterns for Communication

➢ Producer Consumer with Buffer



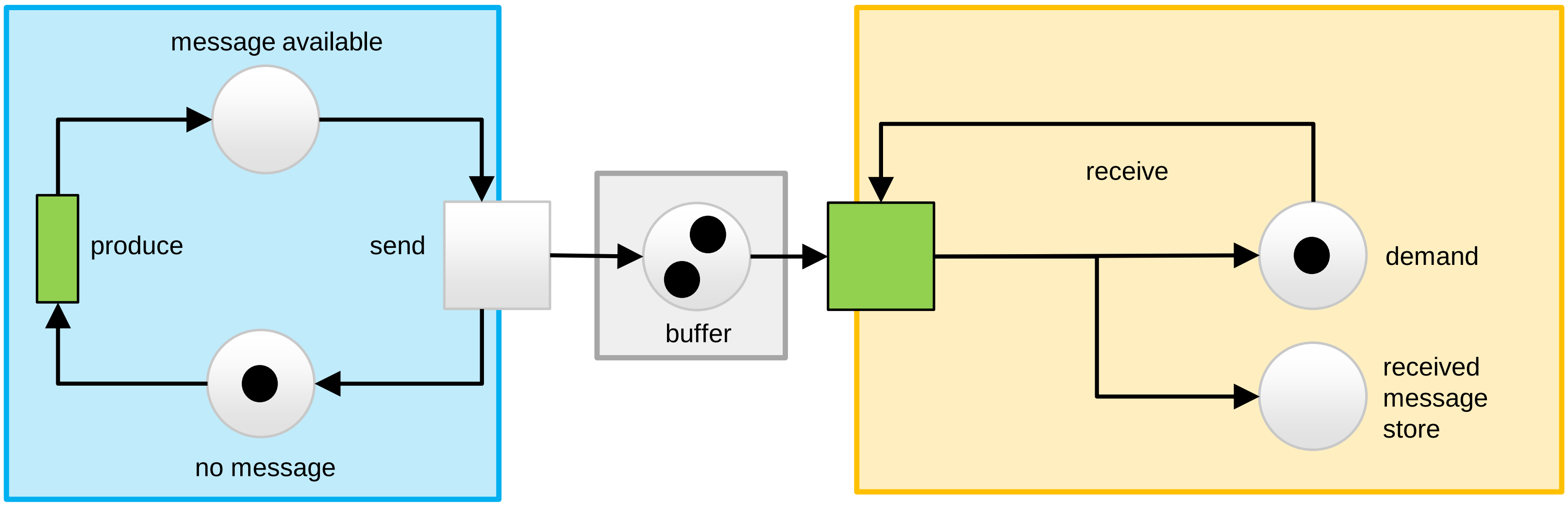
Patterns for Communication

- Producer Consumer with Buffer



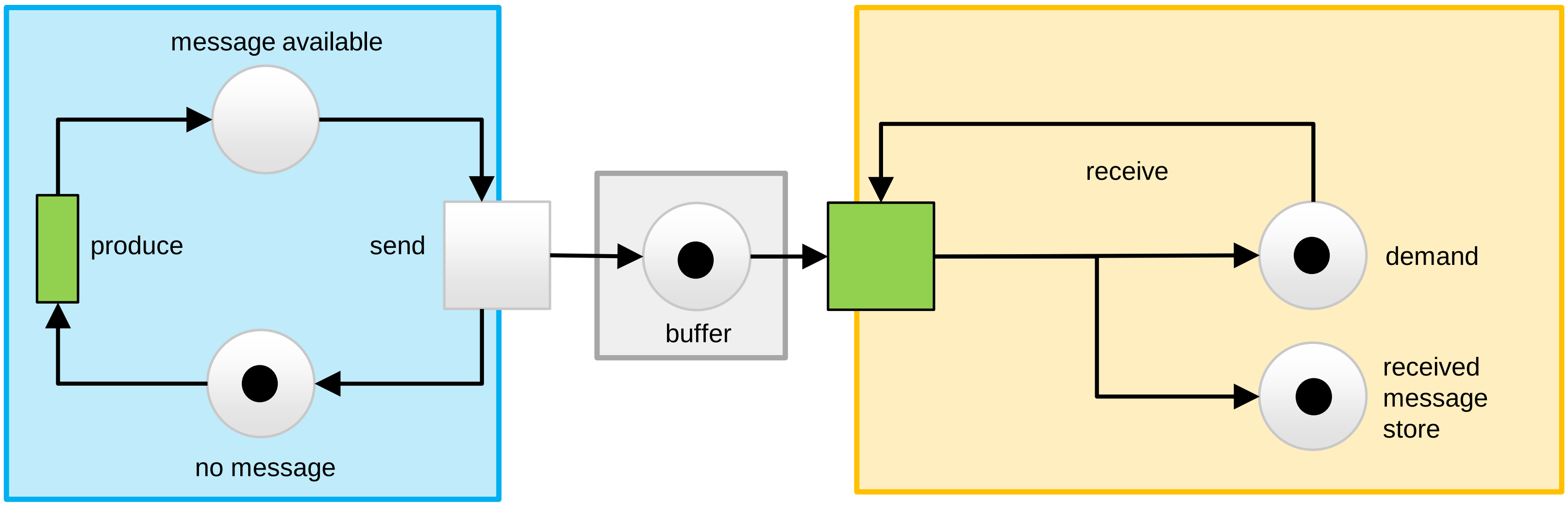
Patterns for Communication

- Producer Consumer with Buffer



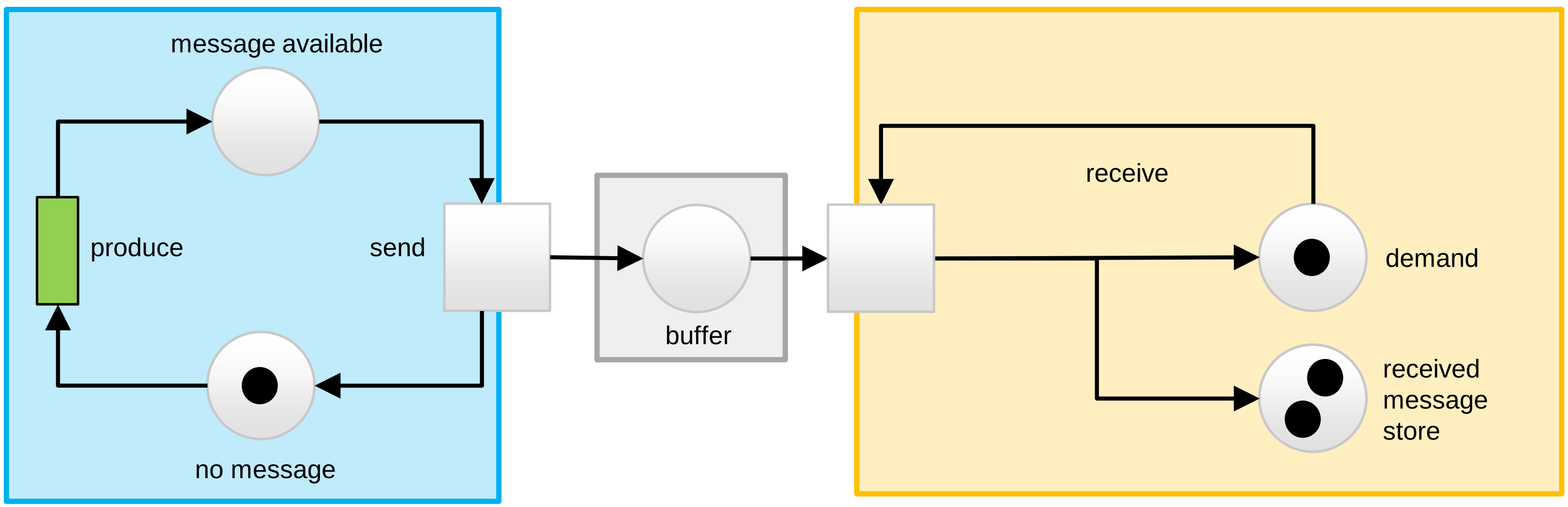
Patterns for Communication

- Producer Consumer with Buffer



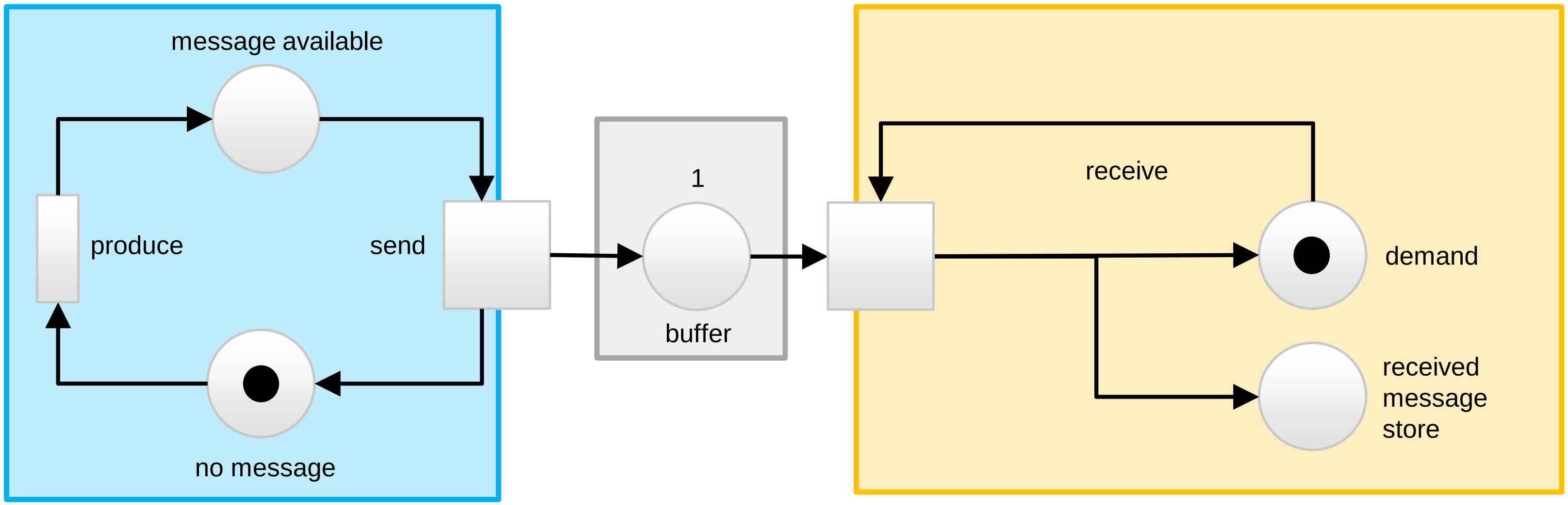
Patterns for Communication

- Producer Consumer with Buffer



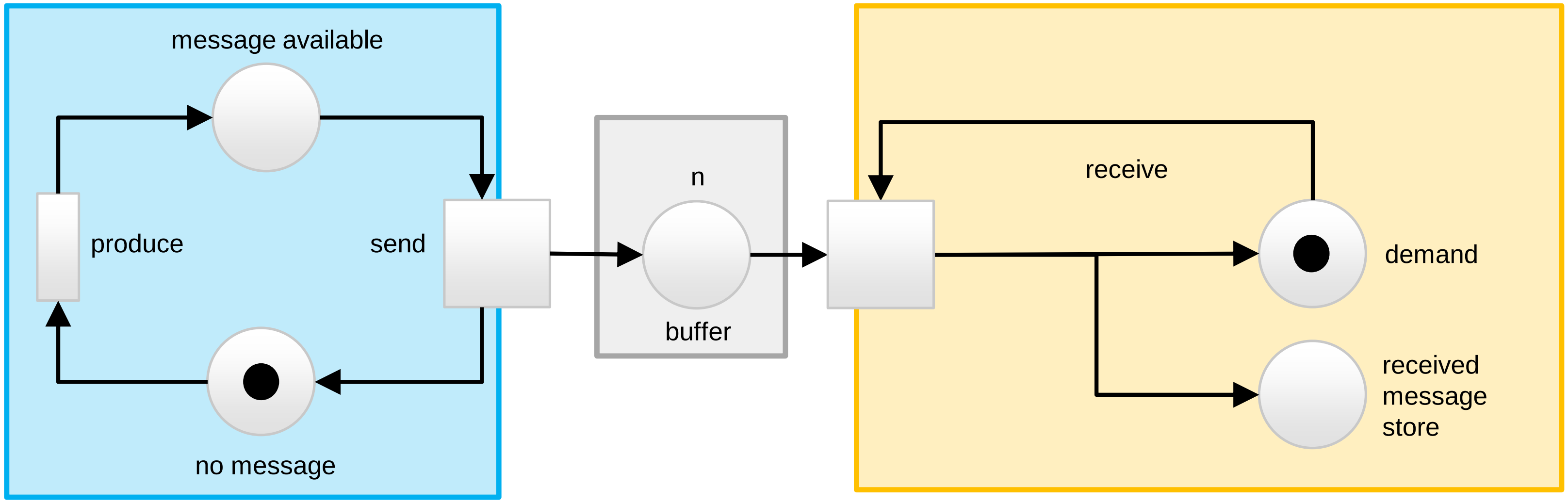
Patterns for Communication

- Producer Consumer with Buffer (size 1 message)



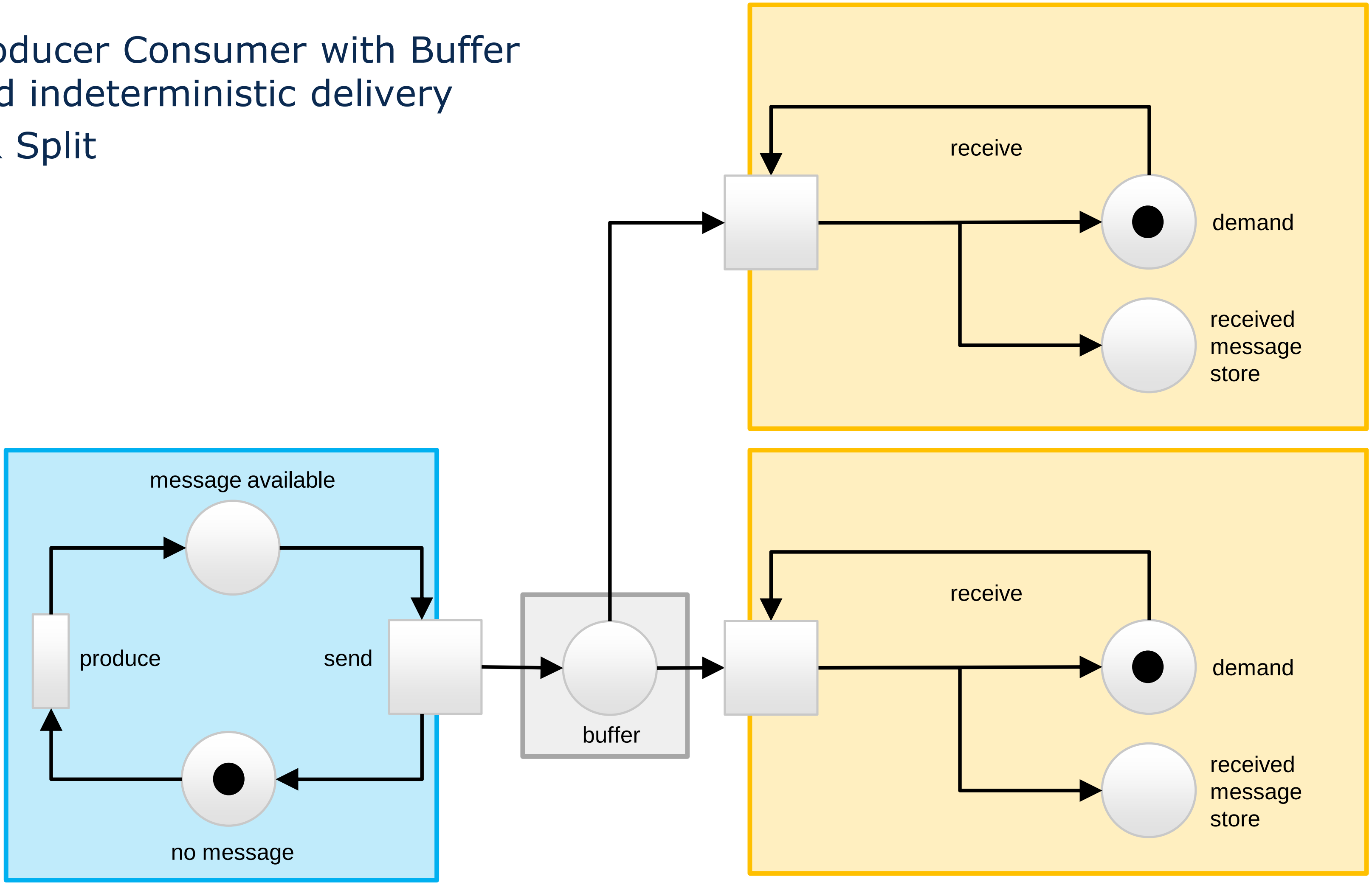
Patterns for Communication

- Producer Consumer with Buffer (size n message)



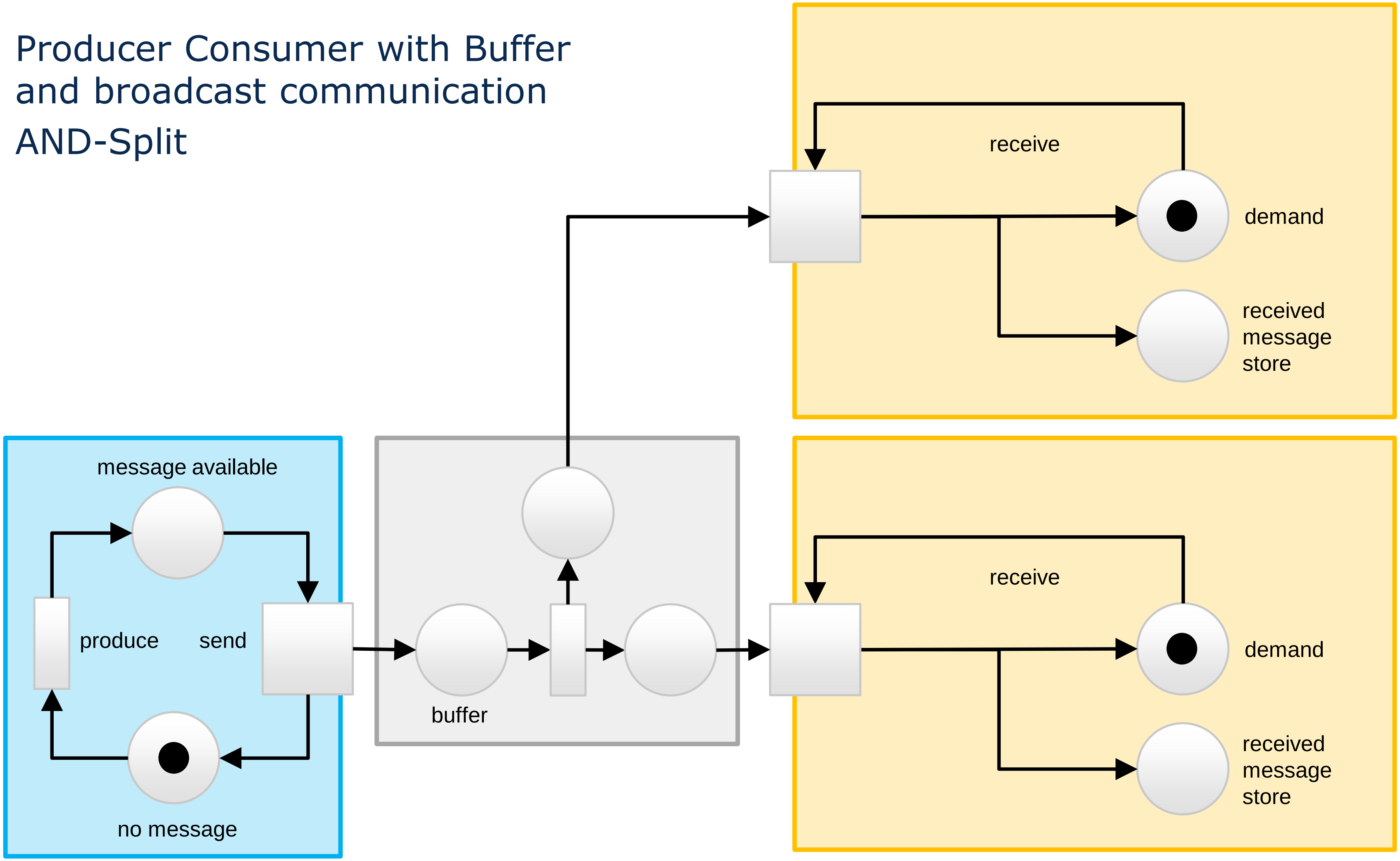
Patterns for Communication

- Producer Consumer with Buffer and indeterministic delivery
- OR Split



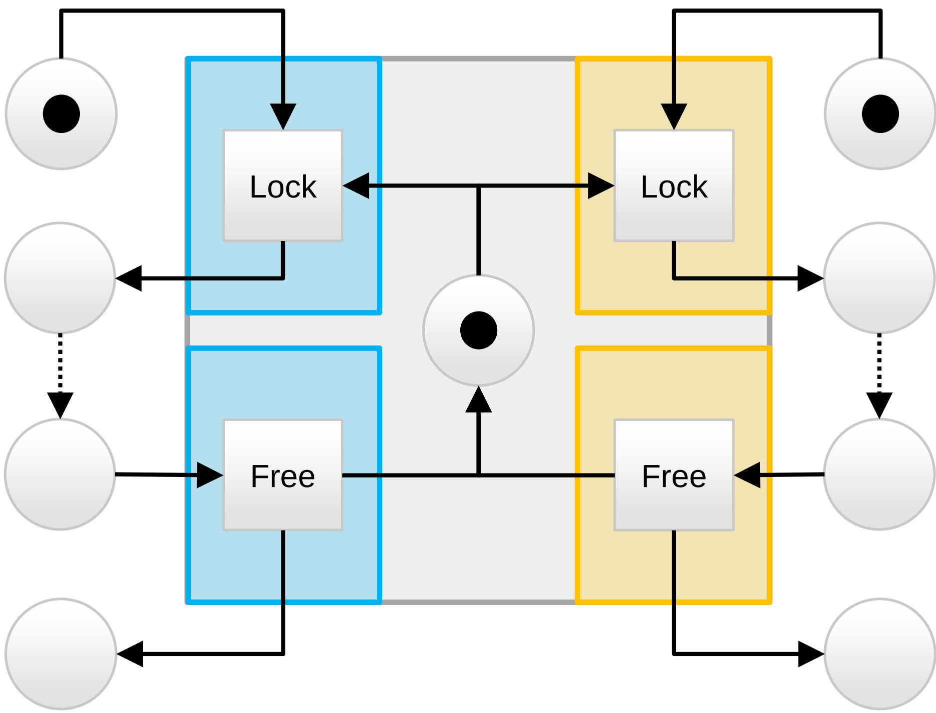
Patterns for Communication

- Producer Consumer with Buffer and broadcast communication
- AND-Split



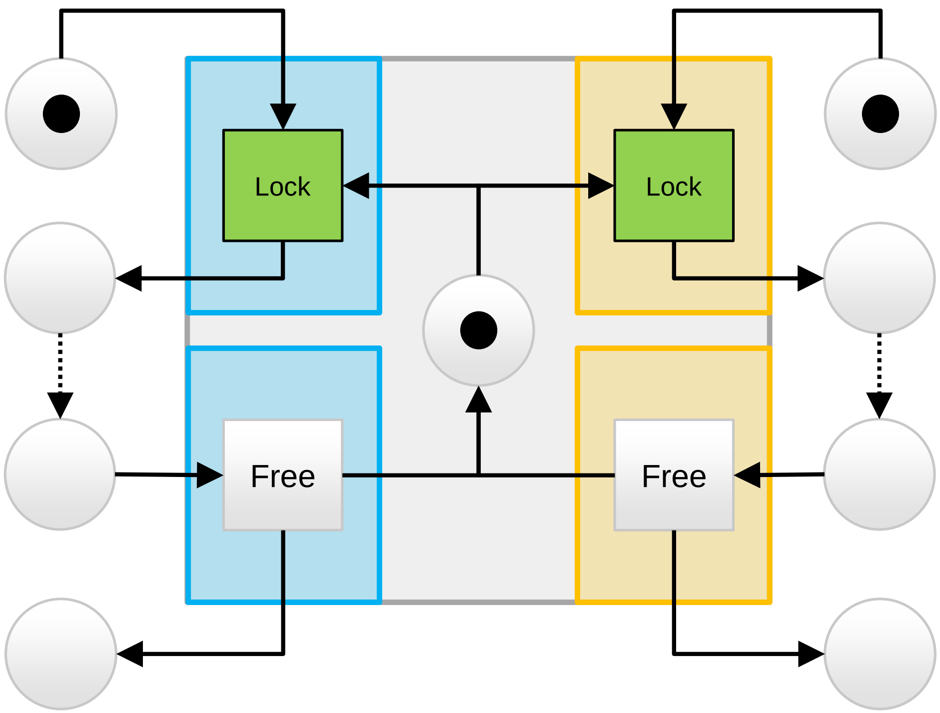
Semaphores For Mutual Exclusion

- Binary or counting semaphores offer their lock and free operations as transitions
- Distinguished by the capacity of the semaphore place

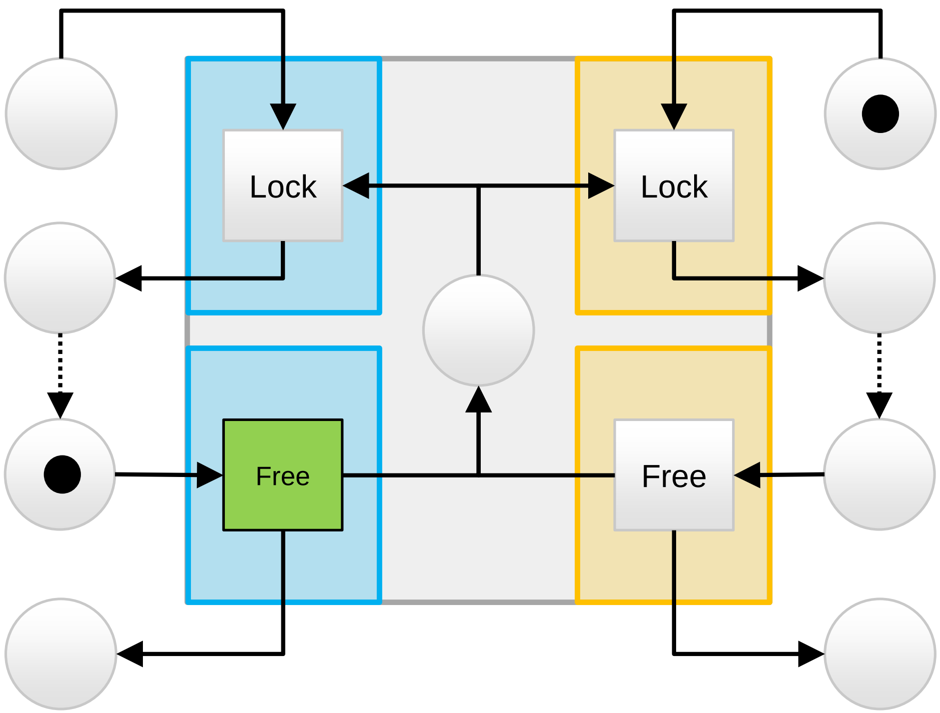


Semaphores For Mutual Exclusion

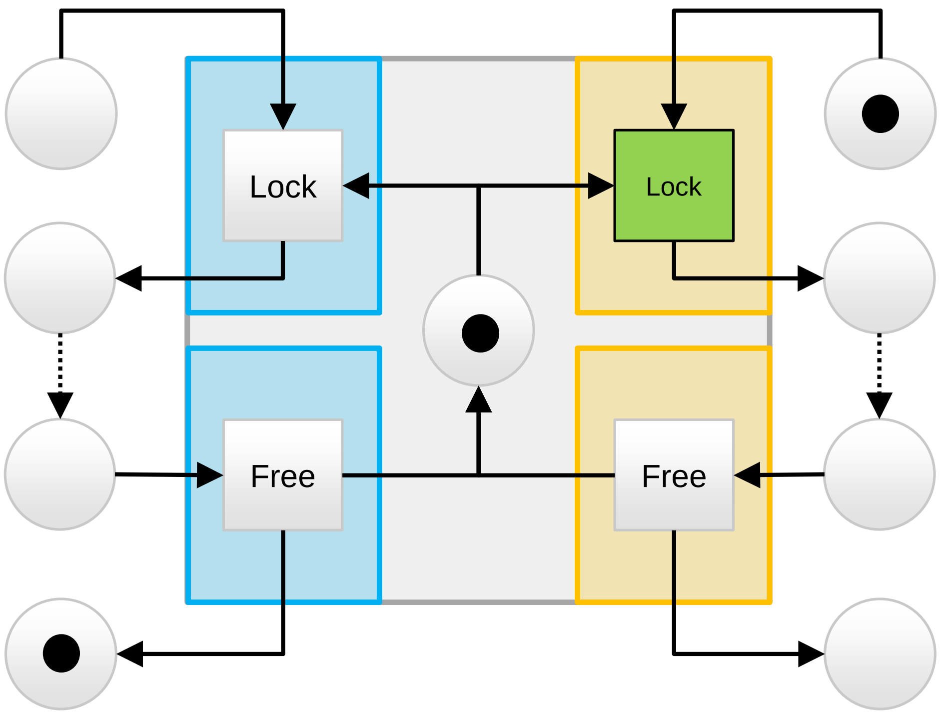
- Binary or counting semaphores offer their lock and free operations as transitions
- Distinguished by the capacity of the semaphore place



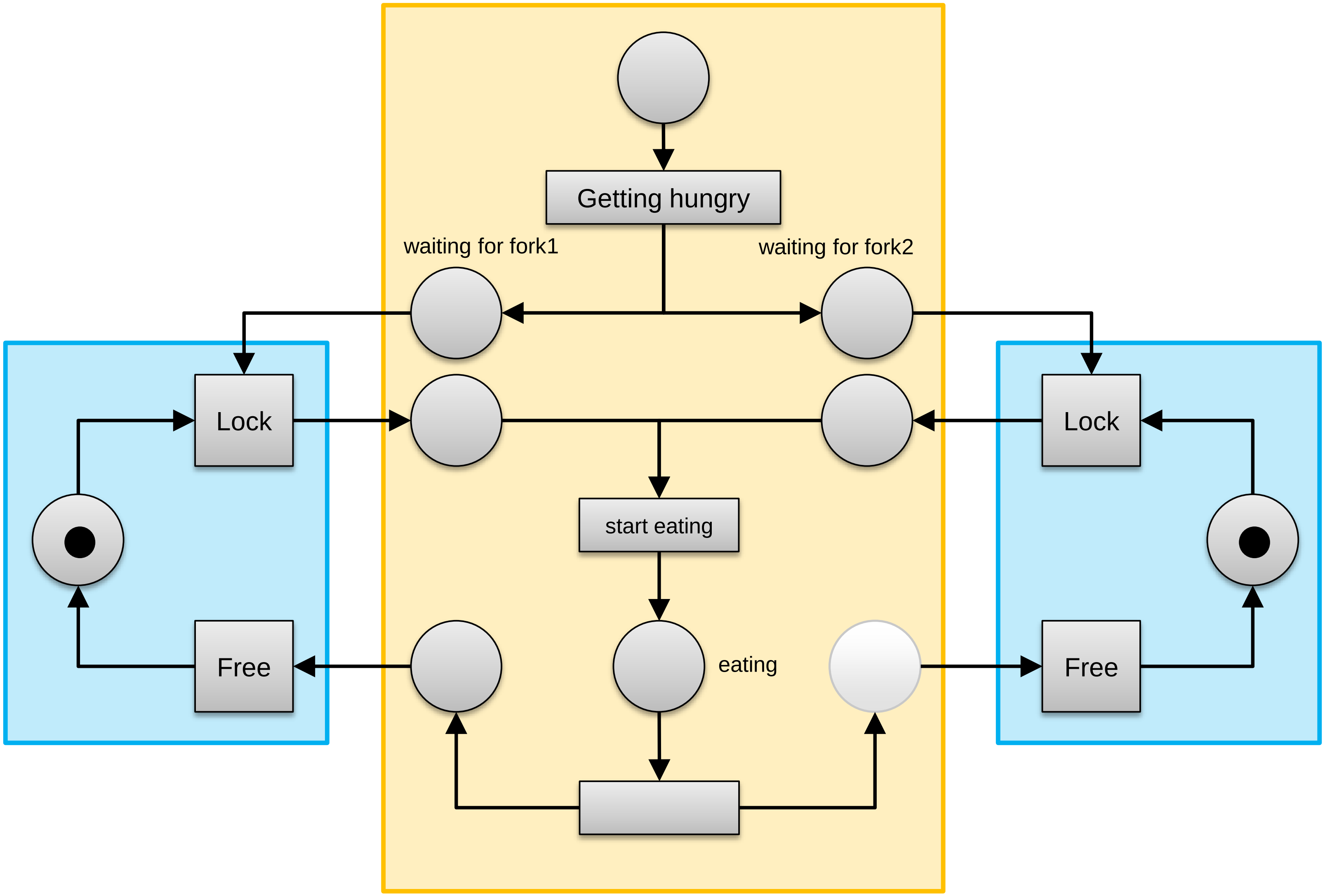
Semaphores For Mutual Exclusion



Semaphores For Mutual Exclusion



Dining Philosophers (Shared Resources)



Advantage

- Patterns can be used to model specific requirements

- PN can be checked for patterns by Pattern Matching (context-free Graph Rewriting)
  - Patterns can be restructured (refactorings)
  - Patterns can be composed (composition)
- PN can be simplified by graph transformation rules

- Further semantic analysis of PN: Parallel, indeterministic systems can be checked for
  - **Absence of deadlocks**: will the parallel system run without getting stuck?
  - **Liveness**: will all parts of the system work forever?
  - **Fairness**: will all parts of the system be loaded equally?
  - **Bounded resources**: will the system use limited memory, and how much? (important for embedded systems)
  - Whether **predicates hold** in certain states (model checking)

3.3 The Application to Modelling

Petri Nets Generalize UML Behavioral Diagrams

Activity Diagrams

- Activity Diagrams are similar to Marked Graphs, but not formally grounded
  - Without markings
  - No liveness analysis
  - No resource consumption analysis with boundness
  - No correspondence to UML-Statechart
- Difficult to prove something about activity diagrams and difficult to generate parallel code

Data-flow diagrams

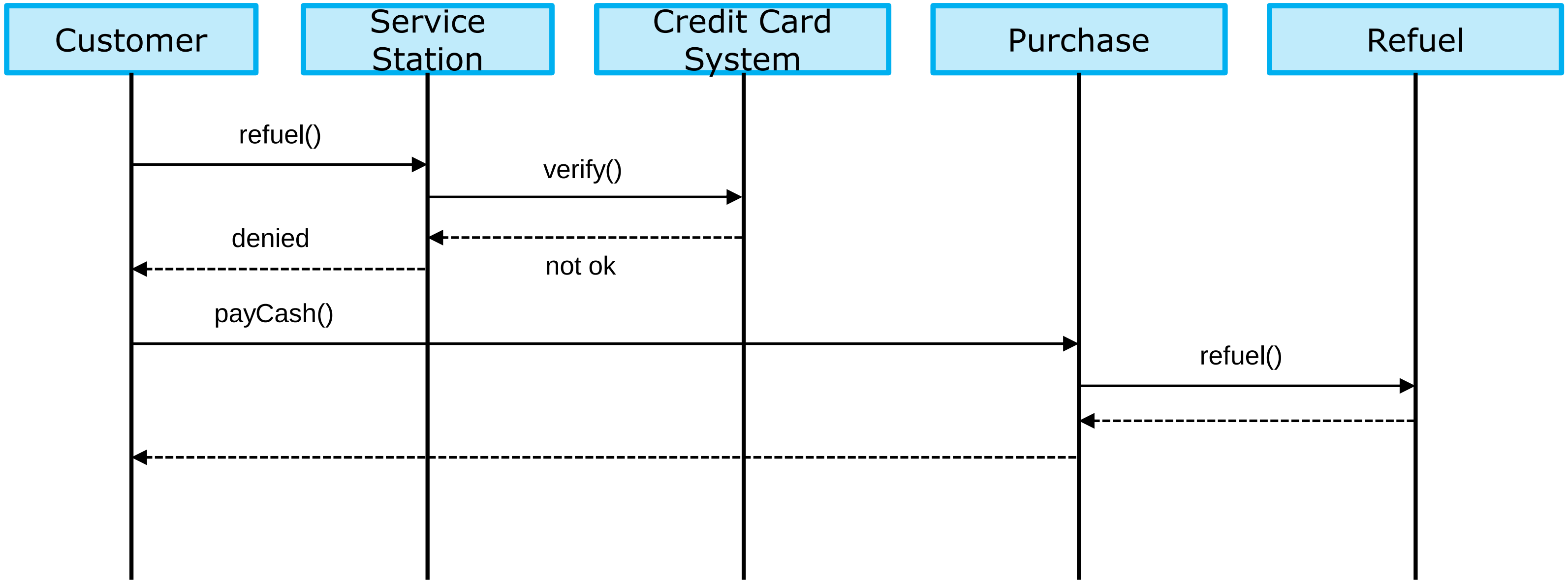
- DFD are special form of activity diagrams
  - Non-shared-memory DFD correspond to Marked Graphs

Statecharts

- Finite automata are restricted form of Petri nets
- Hierarchical structuring in Statecharts is available in High-Level Petri Nets (e.g., CPN)

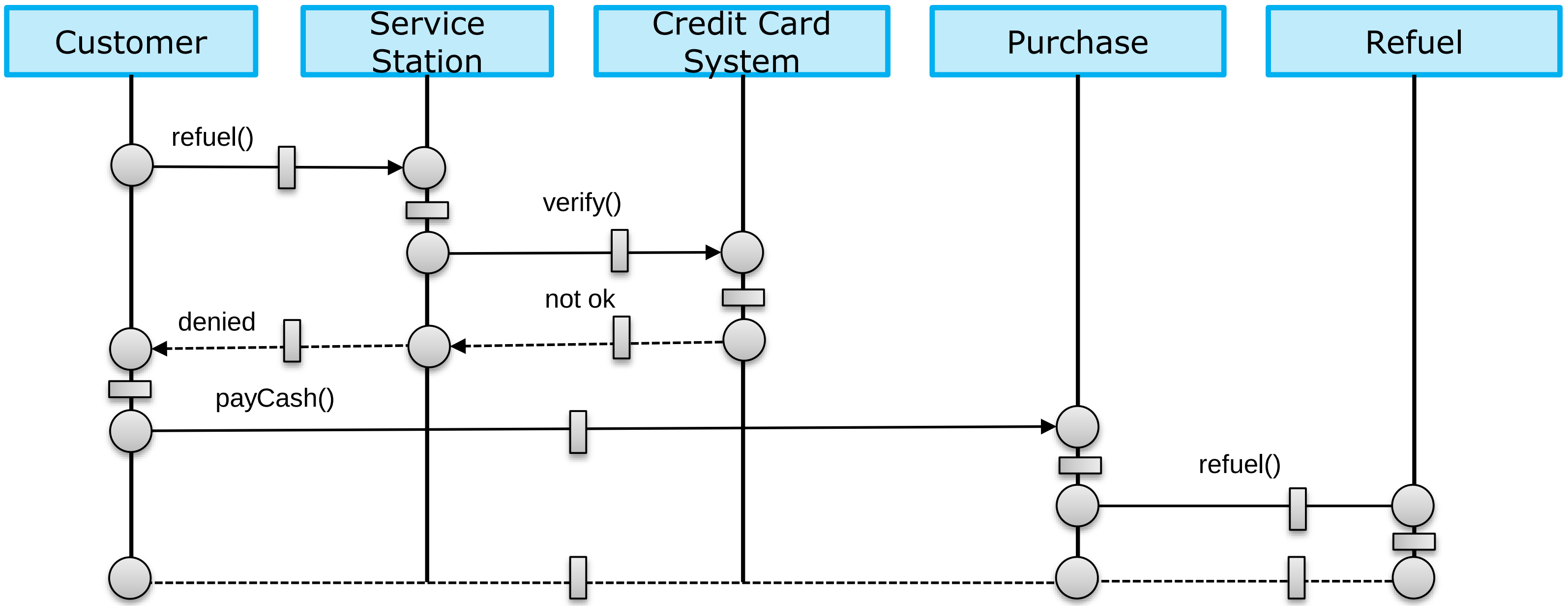
Petri Nets Generalize UML Sequence Diagrams

- The life lines of a sequence diagram can be grouped into state such that a PN results
- All of a sudden, liveness conditions can be studied
  - Is there a deadlock in the sequence diagram?
  - Are objects treated fair?



Petri Nets Generalize UML Sequence Diagrams

- The life lines of a sequence diagram can be grouped into state such that a PN results
- All of a sudden, liveness conditions can be studied
  - Is there a deadlock in the sequence diagram?
  - Are objects treated fair?



A Simple Modelling Process for Safety-Critical Software with CPN

- **Elaboration**:
  1. Identify active and passive parts of the system
     - Active become transitions, passive to places
  2. Find the relations between places and transitions
  3. How should the tokens look like: boolean? Integers? Structured data?
     - Active become transitions, passive to places
- **Restructure**: Group out subnets to separate "pages"
- **Refactor**: Simplify by reduction rules
- **Verify**: Analyse the specification on liveness, boundedness, reachability graphs, fairness. Use a model checker to verify the CPN
- **Transform Representation**: Produce views as statecharts, sequence, collaboration, and activity diagrams.

How to Solve the Reactor Software Problem?

- Specify the reactor core with UML and CPN
  - Map the static parts to the net
  - Map the flow of things to tokens
  - Map the state chances to token flow
  - Think about synchronizations
  - Specify in PN views
- Verify it with a model checker
  - Let a prototype be generated
  - Test it
  - Freeze the assembler
- Verify the assembler, because you should not trust the CPN tool nor the compiler
  - Any certification agency in the world will require a proof of the assembler!
- However, this is much simpler than programming reactors by hand...

The Gloomy Future of PN

- PN will become the major tool in a future CASE tool or IDEs
  - Different views on the PN: state chart view, sequence view, activity view, collaboration view!
- Many isolated tools for PN exist, and the world waits for a full integration into UML
- CPN will be applied in scenarios where parallelism is required
  - Architectural languages
  - Web service langauges (BPEL, BPMN, ...)
  - Workflow languages
  - Coordination languages

The End

- Thanks to Björn Svensson for help to summarize [Murata] in slides