Formal modelling with Petri nets algebras

Franck Pommereau

IBISC, University of Évry / Paris-Saclay

http://www.ibisc.univ-evry.fr/~fpommereau

Master CNS/SA

WARNING

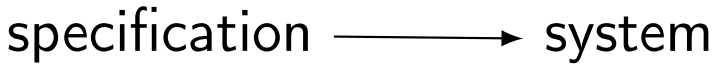
You wan get these slides from my homepage

BUT

you should also get and read my habilitation thesis

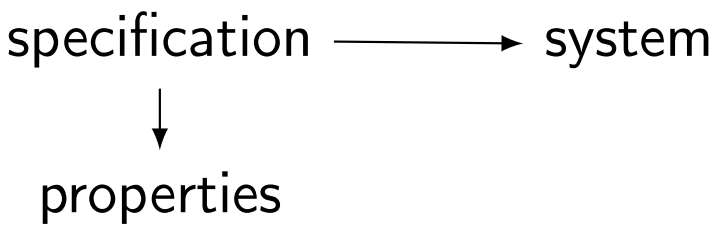
(because the slides lack all the oral explanations while the thesis is detailed)

Formal models for verification



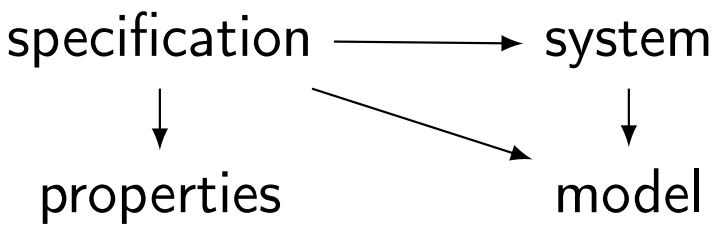
- specification languages
  - informal: UML, EXPRESS, BPMN, flowcharts, SDL, ...
  - formal: Promela, Z and B notations, CO-OPN, Petri script, ...

Formal models for verification



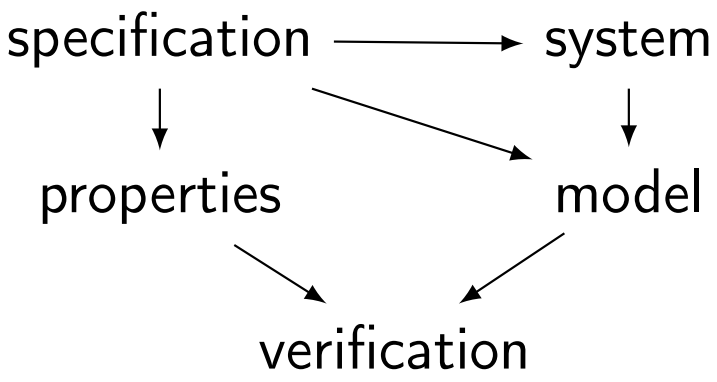
- specification languages
  - informal: UML, EXPRESS, BPMN, flowcharts, SDL, ...
  - formal: Promela, Z and B notations, CO-OPN, Petri script, ...
- languages for properties
  - classical/temporal/timed/modal/$\cdots$ logics

Formal models for verification



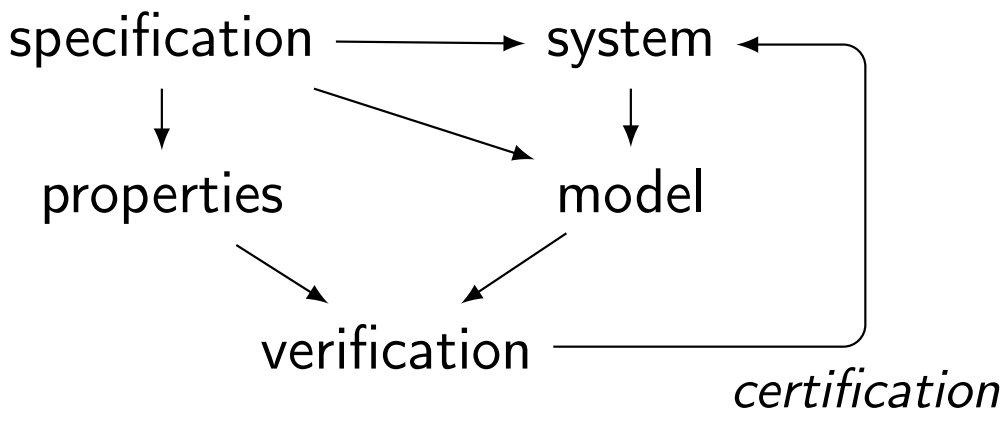
- specification languages
  - informal: UML, EXPRESS, BPMN, flowcharts, SDL, ...
  - formal: Promela, Z and B notations, CO-OPN, Petri script, ...
- languages for properties
  - classical/temporal/timed/modal/$\cdots$ logics
- modelling formalisms
  - timed automata, process algebras, Petri nets, logics, ...

Formal models for verification



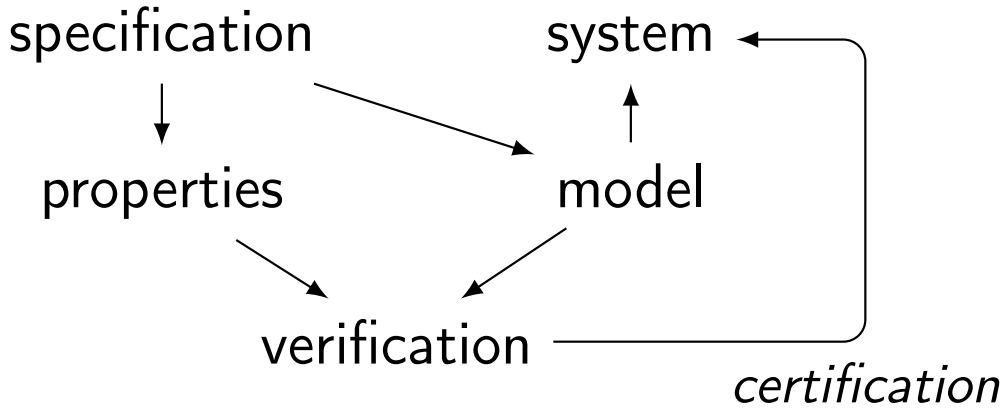
- specification languages
  - informal: UML, EXPRESS, BPMN, flowcharts, SDL, ...
  - formal: Promela, Z and B notations, CO-OPN, Petri script, ...
- languages for properties
  - classical/temporal/timed/modal/$\cdots$ logics
- modelling formalisms
  - timed automata, process algebras, Petri nets, logics, ...

Formal models for verification



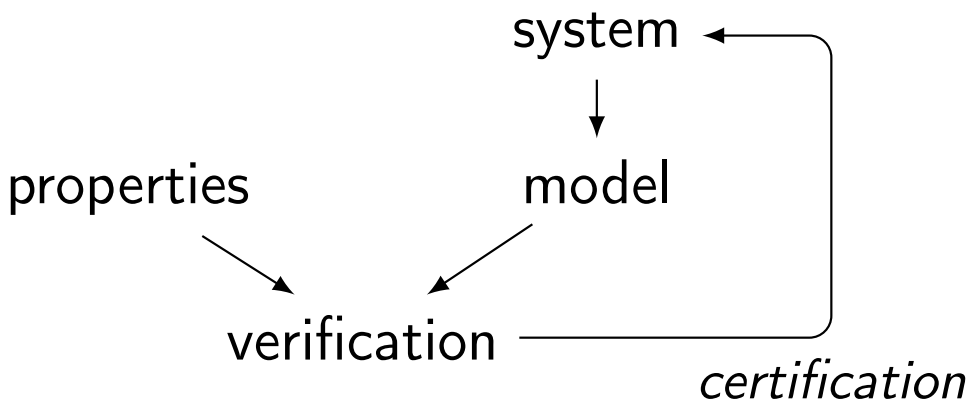
- specification languages
  - informal: UML, EXPRESS, BPMN, flowcharts, SDL, ...
  - formal: Promela, Z and B notations, CO-OPN, Petri script, ...
- languages for properties
  - classical/temporal/timed/modal/$\cdots$ logics
- modelling formalisms
  - timed automata, process algebras, Petri nets, logics, ...

Formal models for verification



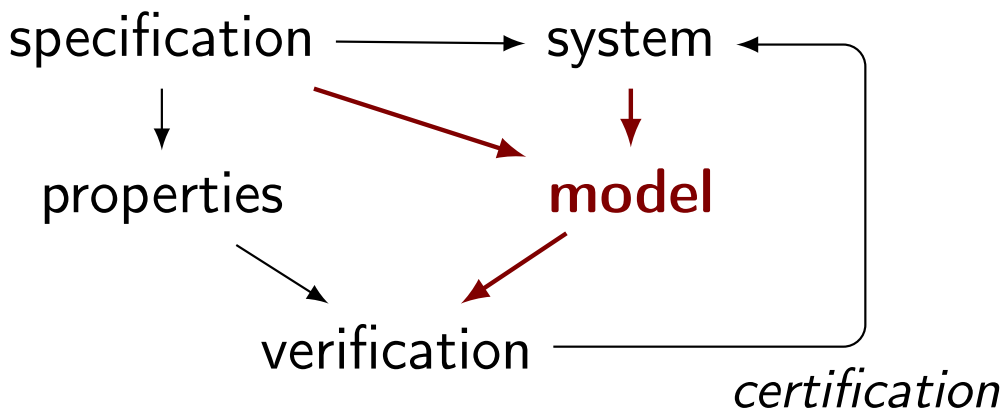
- specification languages
  - informal: UML, EXPRESS, BPMN, flowcharts, SDL, ...
  - formal: Promela, Z and B notations, CO-OPN, Petri script, ...
- languages for properties
  - classical/temporal/timed/modal/$\cdots$ logics
- modelling formalisms
  - timed automata, process algebras, Petri nets, logics, ...

Formal models for verification



- specification languages
  - informal: UML, EXPRESS, BPMN, flowcharts, SDL, ...
  - formal: Promela, Z and B notations, CO-OPN, Petri script, ...
- languages for properties
  - classical/temporal/timed/modal/$\cdots$ logics
- modelling formalisms
  - timed automata, process algebras, Petri nets, logics, ...

Formal models for verification

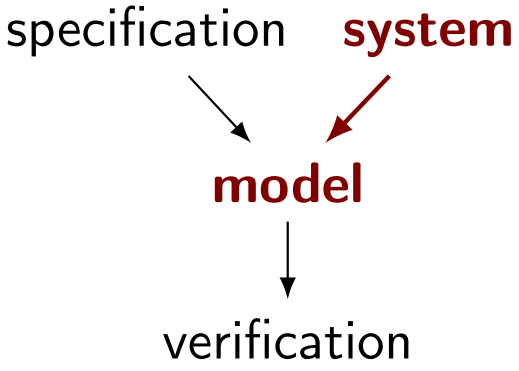


- specification languages
  - informal: UML, EXPRESS, BPMN, flowcharts, SDL, ...
  - formal: Promela, Z and B notations, CO-OPN, Petri script, ...
- languages for properties
  - classical/temporal/timed/modal/$\cdots$ logics
- **modelling formalisms**
  - timed automata, process algebras, Petri nets, logics, ...

Formalisms for practical modelling

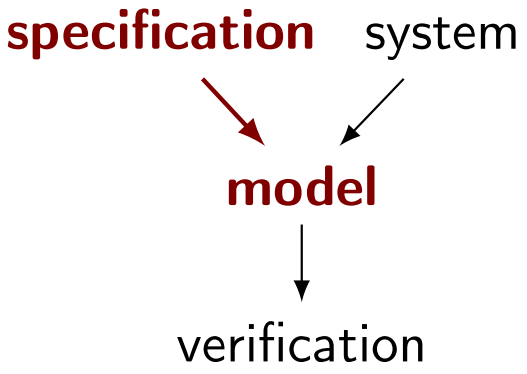
. . . with efficient verification

- suitability to application domains
  - many features needed, great expressivity
  - not always compatible with verification
  - target an application domain
  - modular framework (vs *ad-hoc* formalisms)



Formalisms for practical modelling

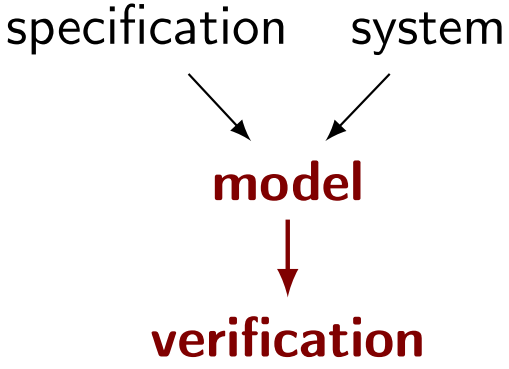
. . . with efficient verification



- suitability to application domains
  - many features needed, great expressivity
  - not always compatible with verification
  - target an application domain
  - modular framework (vs *ad-hoc* formalisms)
- user-friendly formalisms
  - provide user-oriented syntaxes
  - formal semantics under the hood

Formalisms for practical modelling

. . . with efficient verification



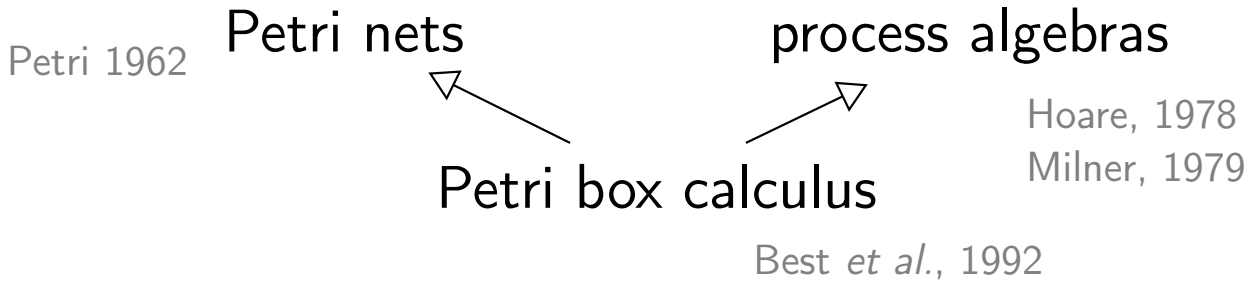
- suitability to application domains
  - many features needed, great expressivity
  - not always compatible with verification
  - target an application domain
  - modular framework (vs *ad-hoc* formalisms)
- user-friendly formalisms
  - provide user-oriented syntaxes
  - formal semantics under the hood
- allow for "fast enough" verification
  - address the issues related to each specific feature
  - general issues are addressed by model-checking people
  - time spent in modelling matters

Algebras of Petri nets

Petri 1962 Petri nets process algebras Hoare, 1978 Milner, 1979

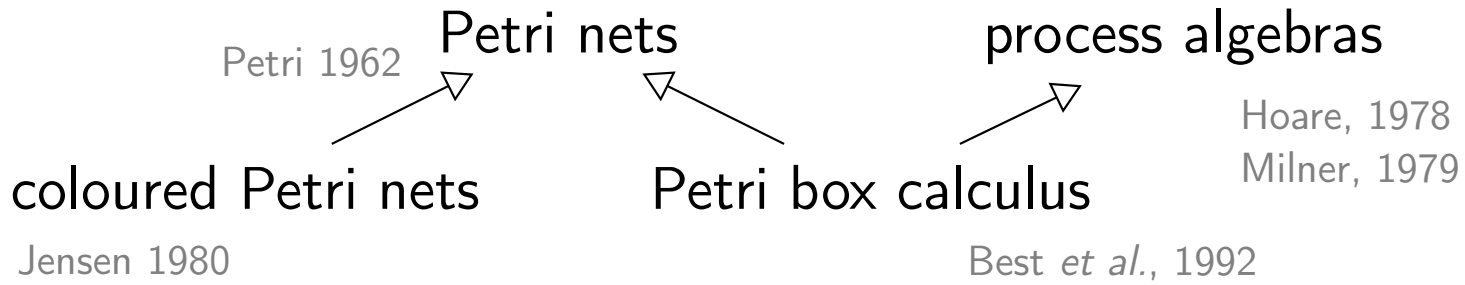
- Petri nets
  - graphical notation, resource production/consumption/sharing, conflicts, concurrency, causality, locality, ...
  - numerous verification techniques (structural and behavioural)
- process algebras (CSP, CCS, π-calculus, ...)
  - textual notation
  - compositionality

Algebras of Petri nets



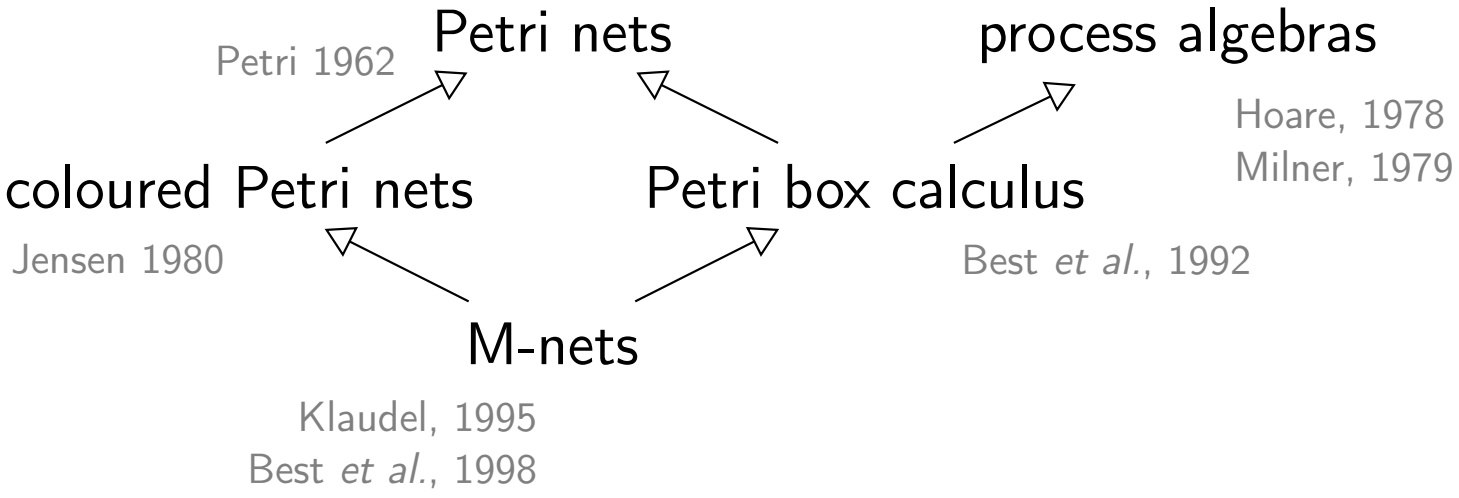
- Petri nets
  - graphical notation, resource production/consumption/sharing, conflicts, concurrency, causality, locality, ...
  - numerous verification techniques (structural and behavioural)
- process algebras (CSP, CCS, π-calculus, ...)
  - textual notation
  - compositionality

Algebras of Petri nets



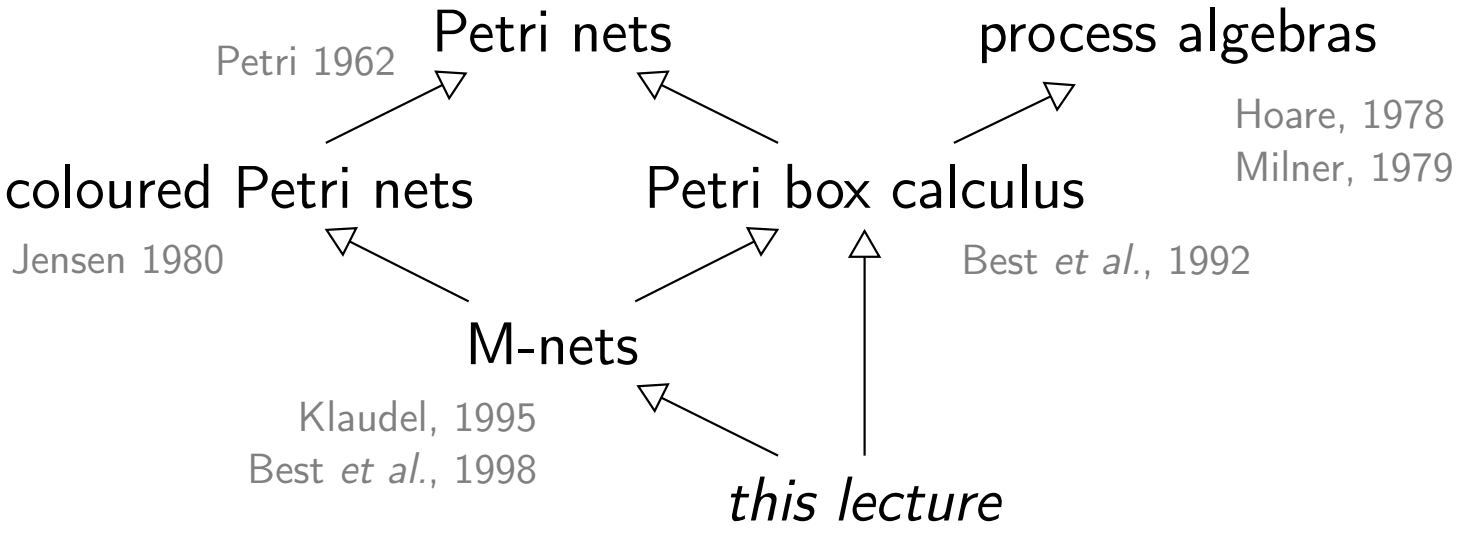
- Petri nets
  - graphical notation, resource production/consumption/sharing, conflicts, concurrency, causality, locality, ...
  - numerous verification techniques (structural and behavioural)
  - compact representation of complex data
- process algebras (CSP, CCS, π-calculus, ...)
  - textual notation
  - compositionality

Algebras of Petri nets



- Petri nets
  - graphical notation, resource production/consumption/sharing, conflicts, concurrency, causality, locality, ...
  - numerous verification techniques (structural and behavioural)
  - compact representation of complex data
- process algebras (CSP, CCS, π-calculus, ...)
  - textual notation
  - compositionality

Algebras of Petri nets



- Petri nets
  - graphical notation, resource production/consumption/sharing, conflicts, concurrency, causality, locality, ...
  - numerous verification techniques (structural and behavioural)
  - compact representation of complex data
- process algebras (CSP, CCS, π-calculus, ...)
  - textual notation
  - compositionality

Outline

A modular framework

ABCD of modelling

More than simulation

Three application domains

Verification issues

Conclusion

Outline

A modular framework
- Coloured Petri nets
- Synchronous communication
- Control flow
- Exceptions
- Threads

ABCD of modelling

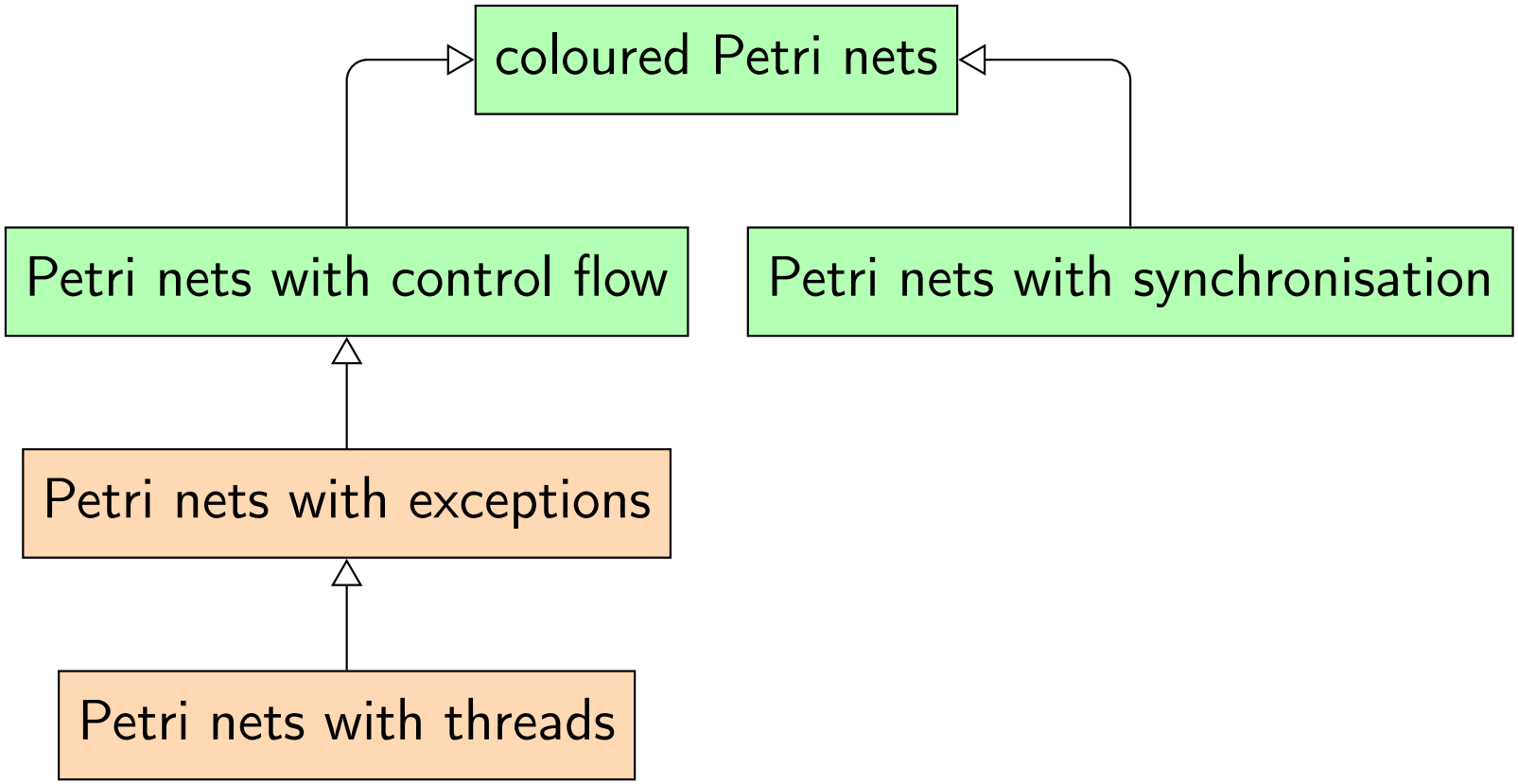
More than simulation

Three application domains

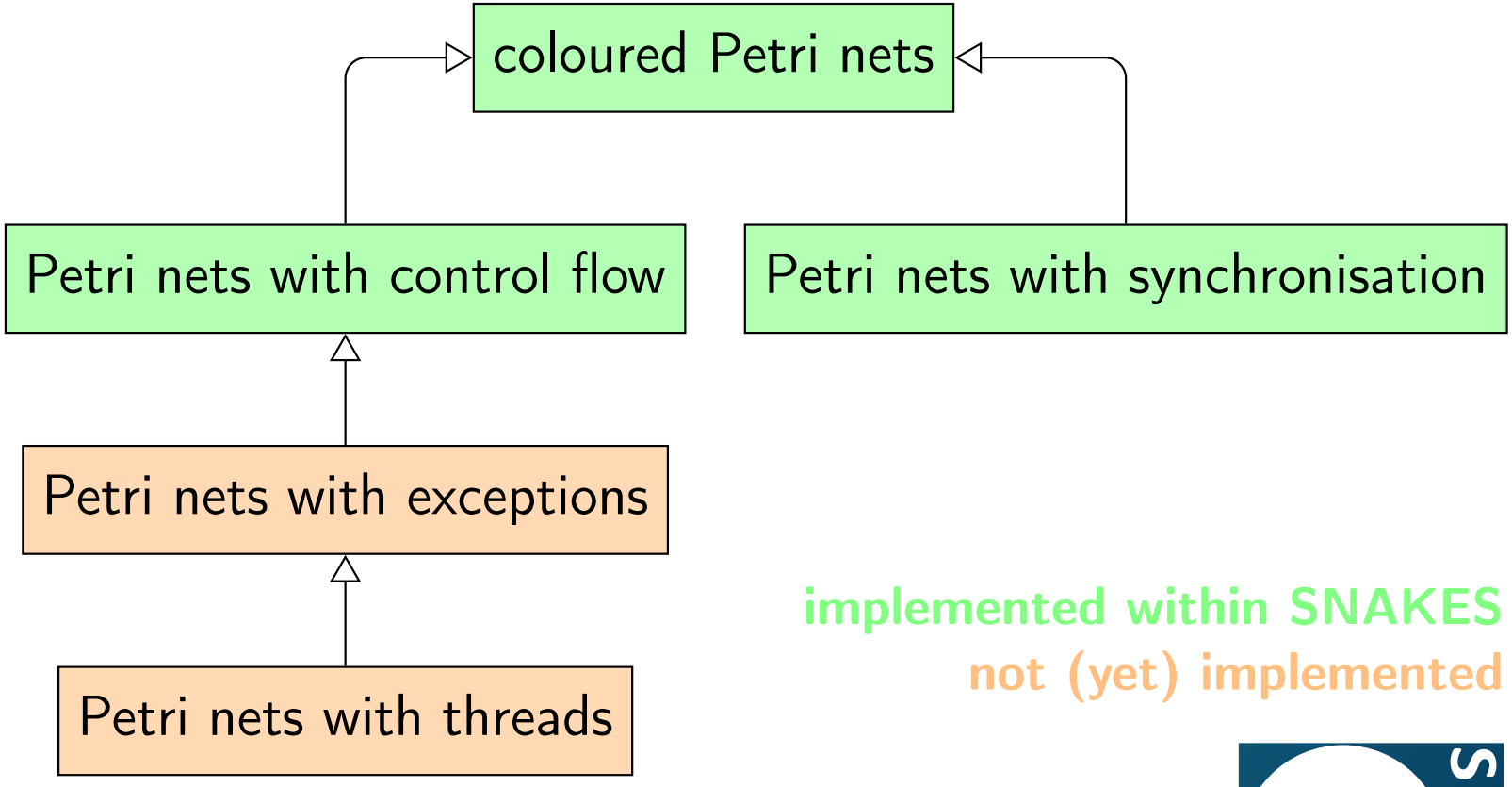
Verification issues

Conclusion

A family of Petri net formalisms

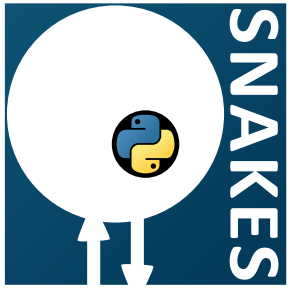


A family of Petri net formalisms



implemented within SNAKES

not (yet) implemented



Outline

A modular framework
- Coloured Petri nets
  - Synchronous communication
  - Control flow
  - Exceptions
  - Threads

ABCD of modelling

More than simulation

Three application domains

Verification issues

Conclusion

Petri nets with abstract colour domain

Colour domain

- data values $\mathbb{D}$ and "undefined" value $\perp$
- variables $\mathbb{V}$
- expressions $\mathbb{E}$ (with $\mathbb{D} \cup \mathbb{V} \subset \mathbb{E}$)
- bindings $\beta : \mathbb{V} \to \mathbb{D}$
- evaluation of $e \in \mathbb{E}$: $\beta(e) \in \mathbb{D} \uplus \{\perp\}$

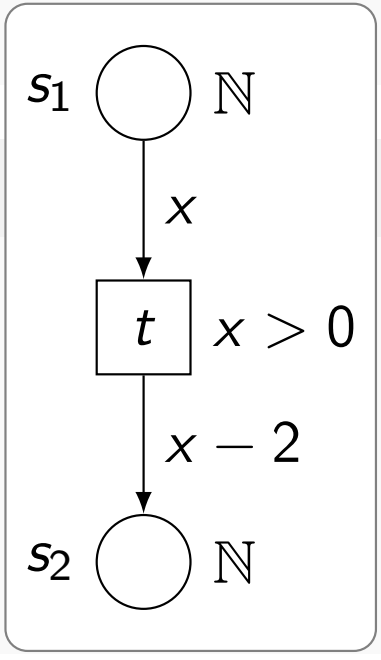
Petri nets with abstract colour domain

Colour domain

- data values $\mathbb{D}$ and "undefined" value $\bot$
- variables $\mathbb{V}$
- expressions $\mathbb{E}$ (with $\mathbb{D} \cup \mathbb{V} \subset \mathbb{E}$)
- bindings $\beta : \mathbb{V} \to \mathbb{D}$
- evaluation of $e \in \mathbb{E}$: $\beta(e) \in \mathbb{D} \uplus \{\bot\}$

Coloured Petri net $N \stackrel{\text{df}}{=} (S, T, \ell)$

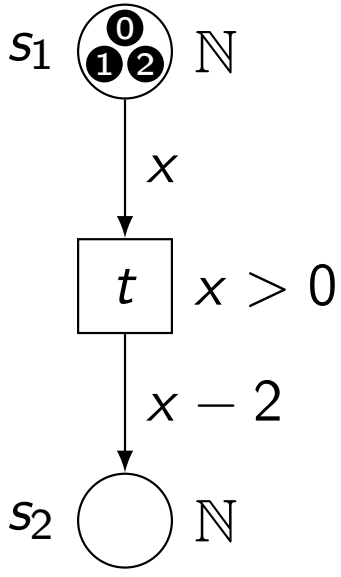
- places S, transitions T, annotations ℓ
- type of place $s \in S$: $\ell(s) \subseteq \mathbb{D}$
- guard of transition $t \in T$: $\ell(t) \in \mathbb{E}$
- arcs $(x, y) \in (S \times T) \cup (T \times S)$: $\ell(x, y) \in \mathbb{E}^\star$



Marking places and firing transitions

- marking of $s \in S$: $M(s) \in \ell(s)^\star$
- transition $s \in T$ enabled by β iff
  - enough tokens $M(s) \geq \beta(\ell(s,t))$
  - guard satisfied $\beta(\ell(t)) = \mathit{true}$
  - place types respected $\beta(\ell(t,s)) \in \ell(s)^\star$
- firing $M\,[t,\beta\rangle\,M'$

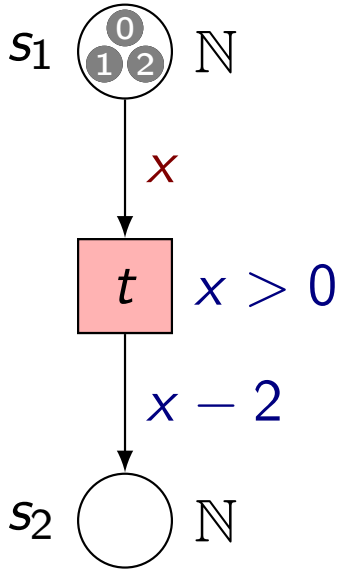
  $\forall s \in S\colon\ M'(s) \stackrel{\text{df}}{=} M(s) - \beta(\ell(s,t)) + \beta(\ell(t,s))$



Marking places and firing transitions

- marking of $s \in S$: $M(s) \in \ell(s)^\star$
- transition $s \in T$ enabled by β iff
  - enough tokens $M(s) \geq \beta(\ell(s,t))$
  - guard satisfied $\beta(\ell(t)) = \mathit{true}$
  - place types respected $\beta(\ell(t,s)) \in \ell(s)^\star$
- firing $M\,[t,\beta\rangle\,M'$

  $\forall s \in S\colon\ M'(s) \stackrel{\text{df}}{=} M(s) - \beta(\ell(s,t)) + \beta(\ell(t,s))$



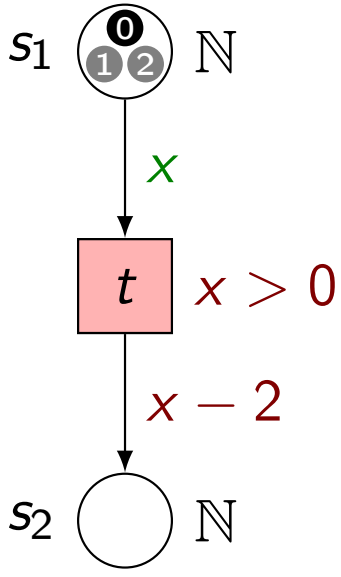
Example

- $\beta_n : x \mapsto n$ $\quad (n \notin \{0,1,2\})$

Marking places and firing transitions

- marking of $s \in S$: $M(s) \in \ell(s)^\star$
- transition $s \in T$ enabled by β iff
  - enough tokens $M(s) \geq \beta(\ell(s,t))$
  - guard satisfied $\beta(\ell(t)) = \mathit{true}$
  - place types respected $\beta(\ell(t,s)) \in \ell(s)^\star$
- firing $M\,[t,\beta\rangle\,M'$

  $\forall s \in S\colon\ M'(s) \stackrel{\text{df}}{=} M(s) - \beta(\ell(s,t)) + \beta(\ell(t,s))$



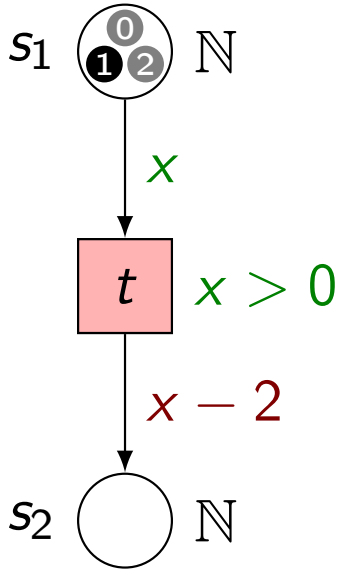
Example

- $\beta_n : x \mapsto n \qquad (n \notin \{0,1,2\})$
- $\beta_0 : x \mapsto 0$

Marking places and firing transitions

- marking of $s \in S$: $M(s) \in \ell(s)^\star$
- transition $s \in T$ enabled by β iff
  - enough tokens $M(s) \geq \beta(\ell(s,t))$
  - guard satisfied $\beta(\ell(t)) = \mathit{true}$
  - place types respected $\beta(\ell(t,s)) \in \ell(s)^\star$
- firing $M\,[t,\beta\rangle\,M'$

  $\forall s \in S\colon\ M'(s) \stackrel{\text{df}}{=} M(s) - \beta(\ell(s,t)) + \beta(\ell(t,s))$



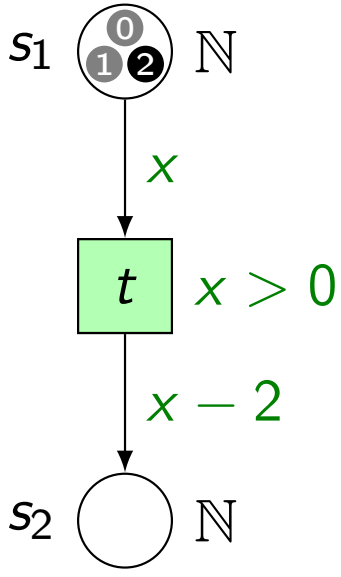
Example

- $\beta_n : x \mapsto n \qquad (n \notin \{0,1,2\})$
- $\beta_0 : x \mapsto 0$
- $\beta_1 : x \mapsto 1$

Marking places and firing transitions

- marking of $s \in S$: $M(s) \in \ell(s)^\star$
- transition $s \in T$ enabled by β iff
  - enough tokens $M(s) \geq \beta(\ell(s,t))$
  - guard satisfied $\beta(\ell(t)) = \mathit{true}$
  - place types respected $\beta(\ell(t,s)) \in \ell(s)^\star$
- firing $M\,[t,\beta\rangle\,M'$

  $\forall s \in S\colon\ M'(s) \stackrel{\text{df}}{=} M(s) - \beta(\ell(s,t)) + \beta(\ell(t,s))$



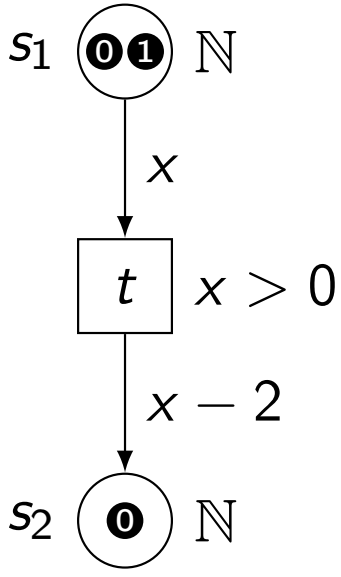
Example

- $\beta_n : x \mapsto n \qquad (n \notin \{0,1,2\})$
- $\beta_0 : x \mapsto 0$
- $\beta_1 : x \mapsto 1$
- $\beta_2 : x \mapsto 2$

Marking places and firing transitions

- marking of $s \in S$: $M(s) \in \ell(s)^\star$
- transition $s \in T$ enabled by β iff
  - enough tokens $M(s) \geq \beta(\ell(s,t))$
  - guard satisfied $\beta(\ell(t)) = \mathit{true}$
  - place types respected $\beta(\ell(t,s)) \in \ell(s)^\star$
- firing $M\,[t,\beta\rangle\,M'$

  $\forall s \in S\colon\ M'(s) \stackrel{\text{df}}{=} M(s) - \beta(\ell(s,t)) + \beta(\ell(t,s))$



Example

- $\beta_n : x \mapsto n \qquad (n \notin \{0,1,2\})$
- $\beta_0 : x \mapsto 0$
- $\beta_1 : x \mapsto 1$
- $\beta_2 : x \mapsto 2$

Semantics of Petri nets

- sequential semantics: marking graph
  - smallest graph (V, E) such that
    - the initial marking is a node $M_0 \in V$
    - each firing creates an edge $M\,[t, \beta\rangle\, M' \Rightarrow M' \in V, (t, \beta) \in E$
  - contains all the traces (sequential runs)
  - may be infinite (infinite markings, infinite data domain, ...)
  - can be seen as a LTS (labelled transition system)
  - or as a Kripke structure
- various concurrent semantics
  - steps: fire more than one transition simultaneously
  - branching process: partial order representation of one concurrent run
  - unfolding: compact representation of all the branching processes

Outline

A modular framework
- Coloured Petri nets
- Synchronous communication
- Control flow
- Exceptions
- Threads

ABCD of modelling

More than simulation

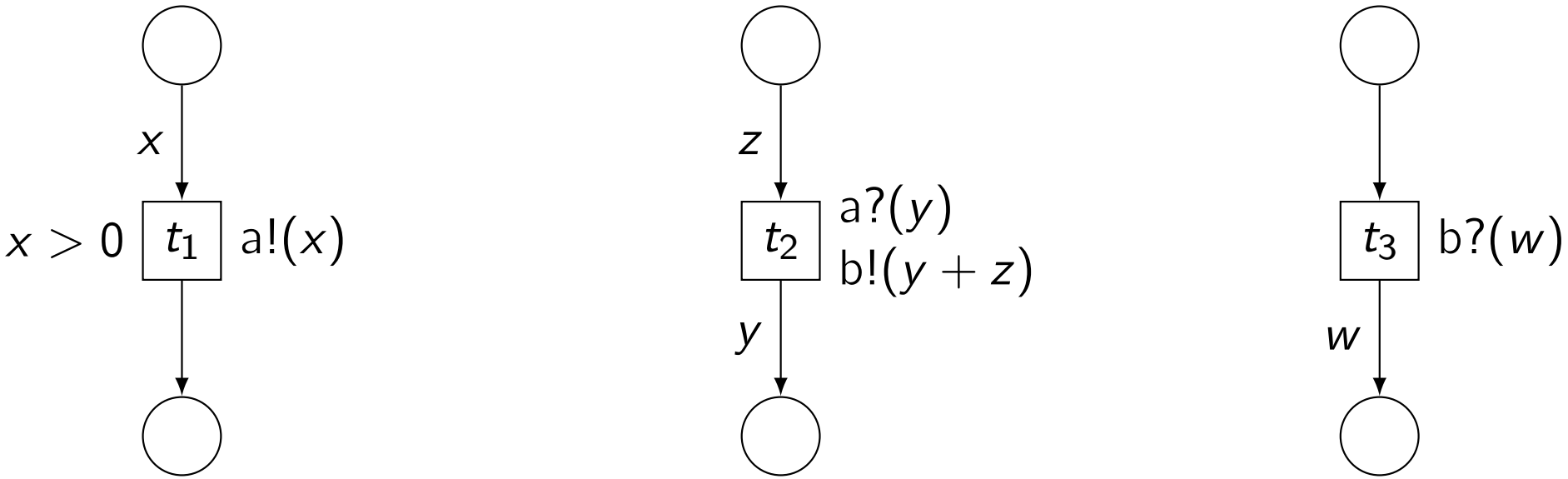
Three application domains

Verification issues

Conclusion

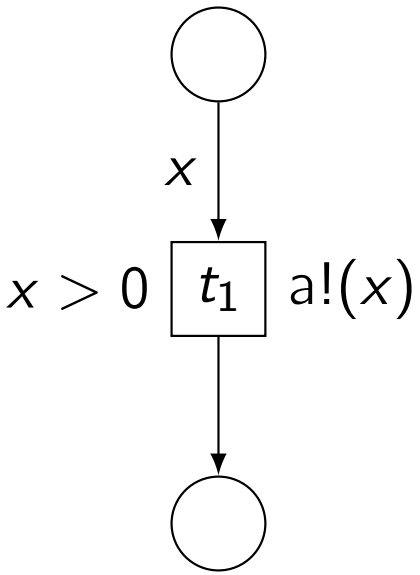
Multi-way synchronisation with data passing

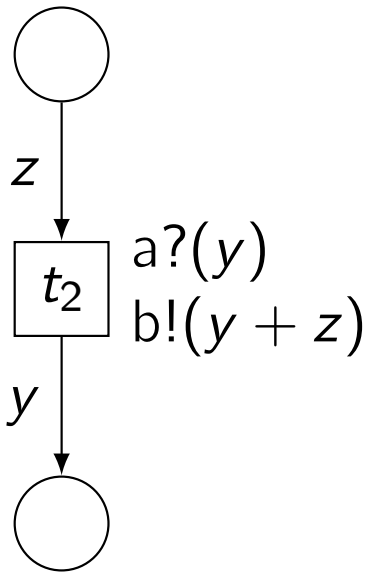
- multi-actions N_0

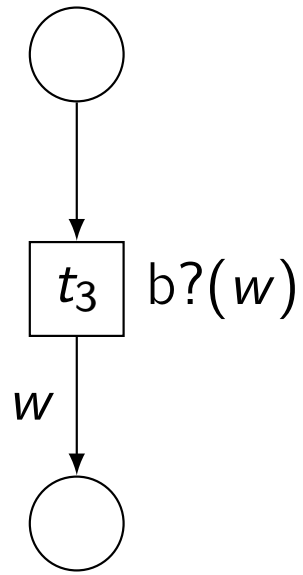


Multi-way synchronisation with data passing

- multi-actions N_0
- synchronisation $N_1 \stackrel{\text{df}}{=} N_0 \; sy \; \mathsf{a}$

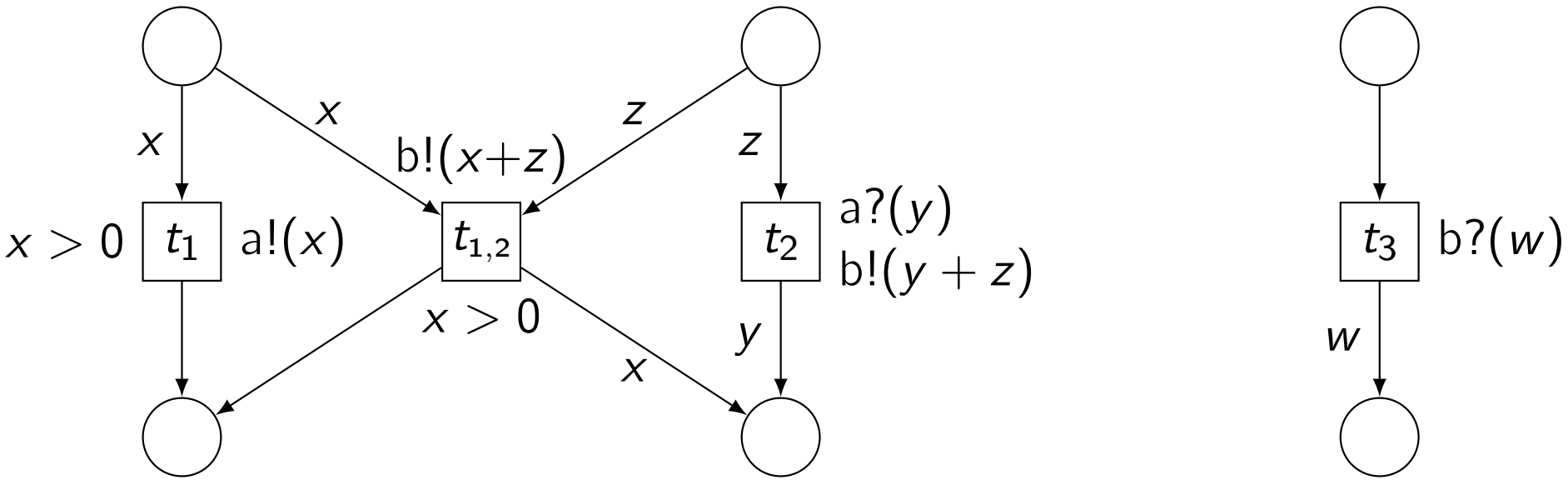






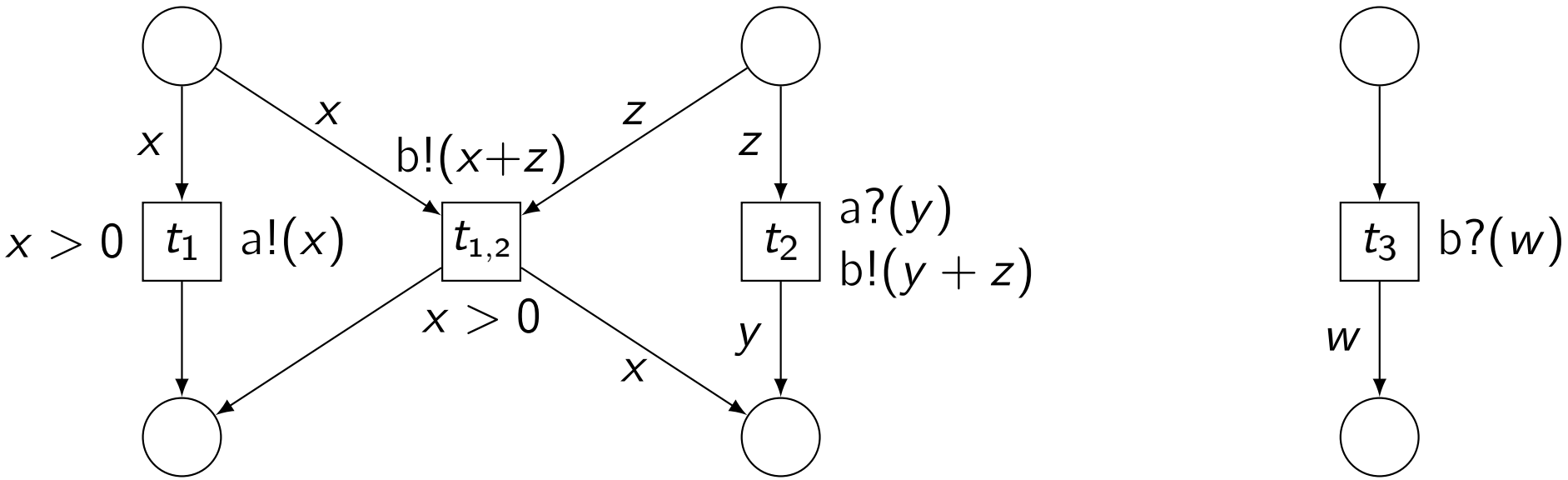
Multi-way synchronisation with data passing

- multi-actions N_0
- synchronisation $N_1 \stackrel{\text{df}}{=} N_0 \; sy \; \mathsf{a} \qquad (y \mapsto x)$



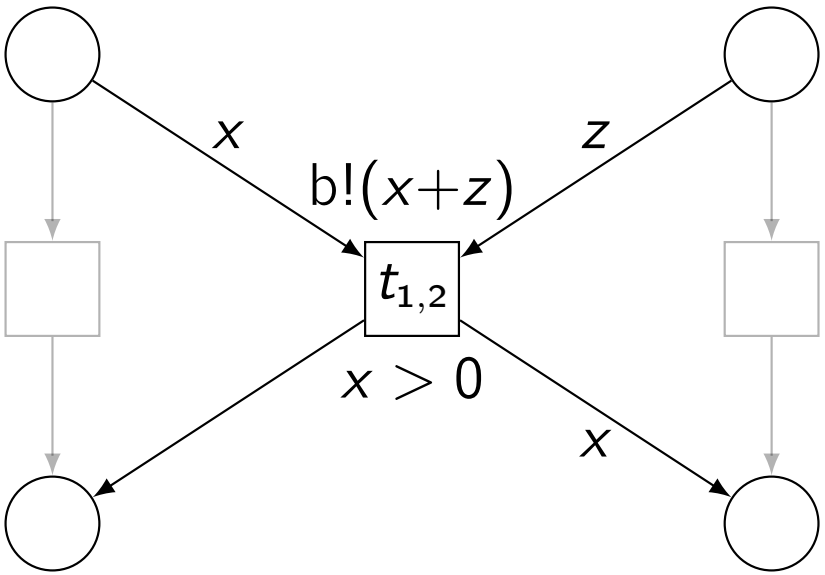
Multi-way synchronisation with data passing

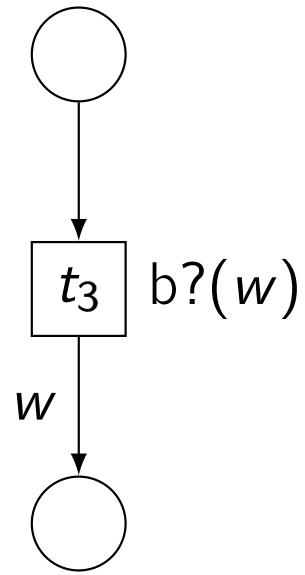
- multi-actions N_0
- synchronisation $N_1 \stackrel{\text{df}}{=} N_0\ sy\ \mathrm{a} \qquad (y \mapsto x)$
- restriction $N_2 \stackrel{\text{df}}{=} N_1\ rs\ \mathrm{a}$



Multi-way synchronisation with data passing

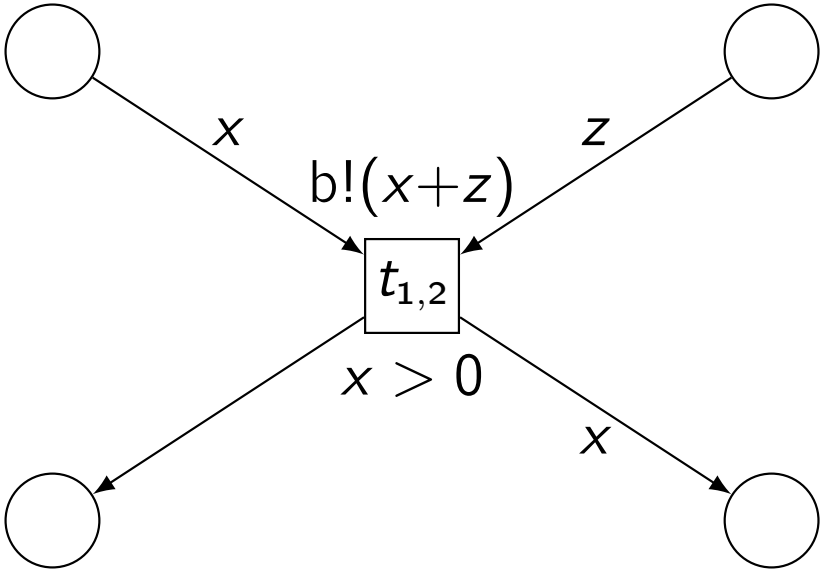
- multi-actions N_0
- synchronisation $N_1 \stackrel{\text{df}}{=} N_0\ sy\ \mathrm{a}$ $(y \mapsto x)$
- restriction $N_2 \stackrel{\text{df}}{=} N_1\ rs\ \mathrm{a}$

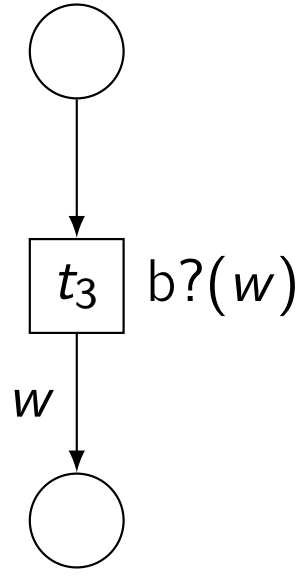




Multi-way synchronisation with data passing

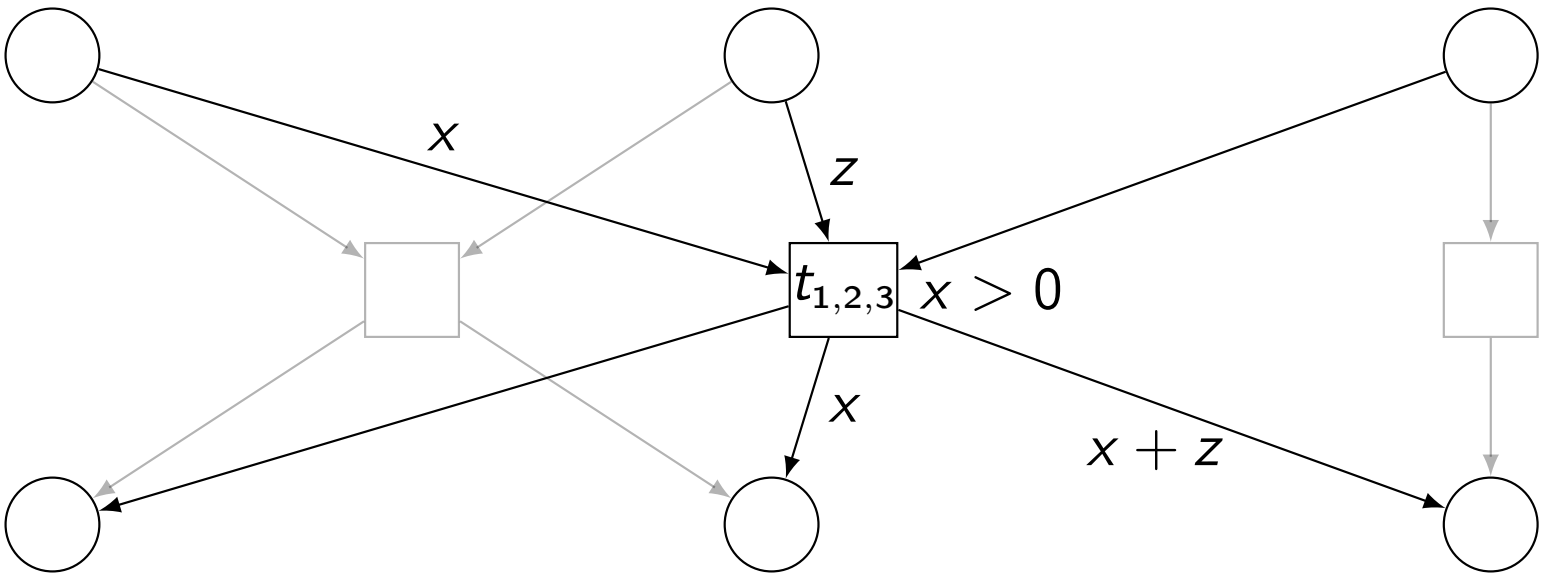
- multi-actions N_0
- synchronisation $N_1 \stackrel{\text{df}}{=} N_0\ sy\ \mathrm{a} \qquad (y \mapsto x)$
- restriction $N_2 \stackrel{\text{df}}{=} N_1\ rs\ \mathrm{a}$
- scoping $N_3 \stackrel{\text{df}}{=} N_2\ sc\ \mathrm{b} \stackrel{\text{df}}{=} (N_2\ sy\ \mathrm{b})\ rs\ \mathrm{b}$





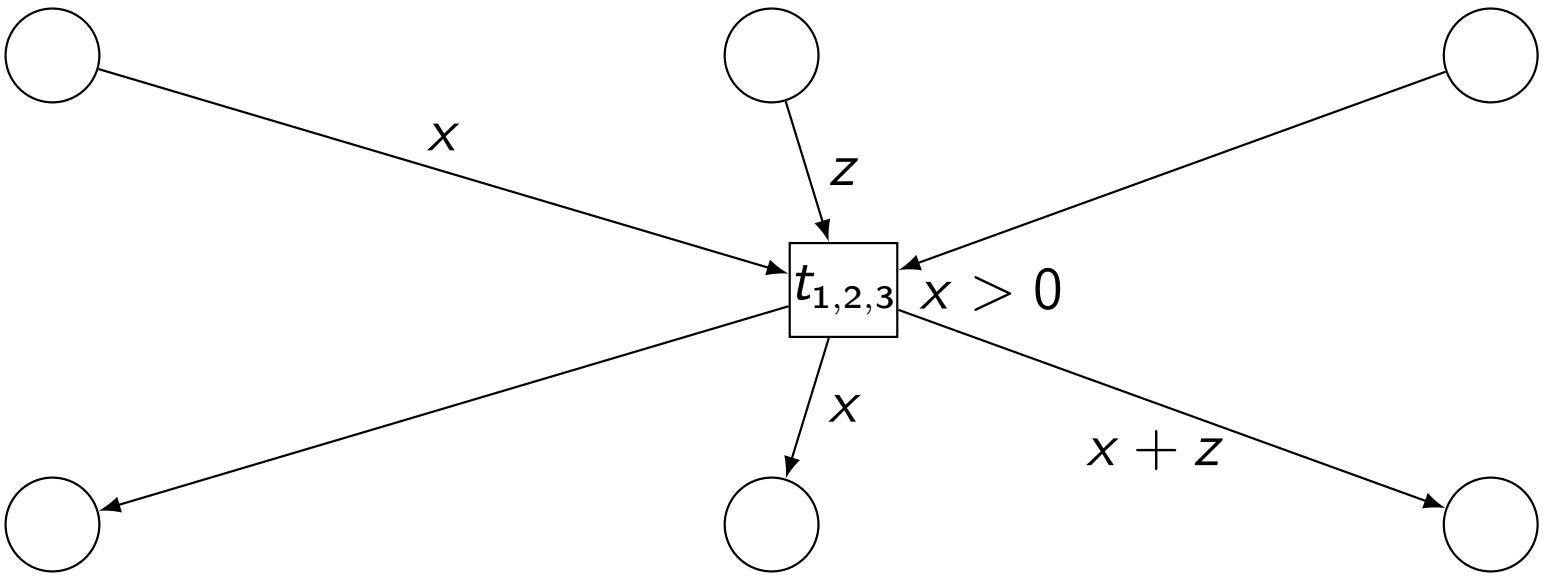
Multi-way synchronisation with data passing

- multi-actions N_0
- synchronisation $N_1 \stackrel{\text{df}}{=} N_0 \; sy \; \mathsf{a}$ $\quad (y \mapsto x)$
- restriction $N_2 \stackrel{\text{df}}{=} N_1 \; rs \; \mathsf{a}$
- scoping $N_3 \stackrel{\text{df}}{=} N_2 \; sc \; \mathsf{b} \stackrel{\text{df}}{=} (N_2 \; sy \; \mathsf{b}) \; rs \; \mathsf{b}$ $\quad (w \mapsto x + z)$



Multi-way synchronisation with data passing

- multi-actions N_0
- synchronisation $N_1 \stackrel{\text{df}}{=} N_0 \; sy \; \mathsf{a}$ $\quad (y \mapsto x)$
- restriction $N_2 \stackrel{\text{df}}{=} N_1 \; rs \; \mathsf{a}$
- scoping $N_3 \stackrel{\text{df}}{=} N_2 \; sc \; \mathsf{b} \stackrel{\text{df}}{=} (N_2 \; sy \; \mathsf{b}) \; rs \; \mathsf{b}$ $\quad (w \mapsto x + z)$



Outline

A modular framework
- Coloured Petri nets
- Synchronous communication
- Control flow
- Exceptions
- Threads

ABCD of modelling

More than simulation

Three application domains

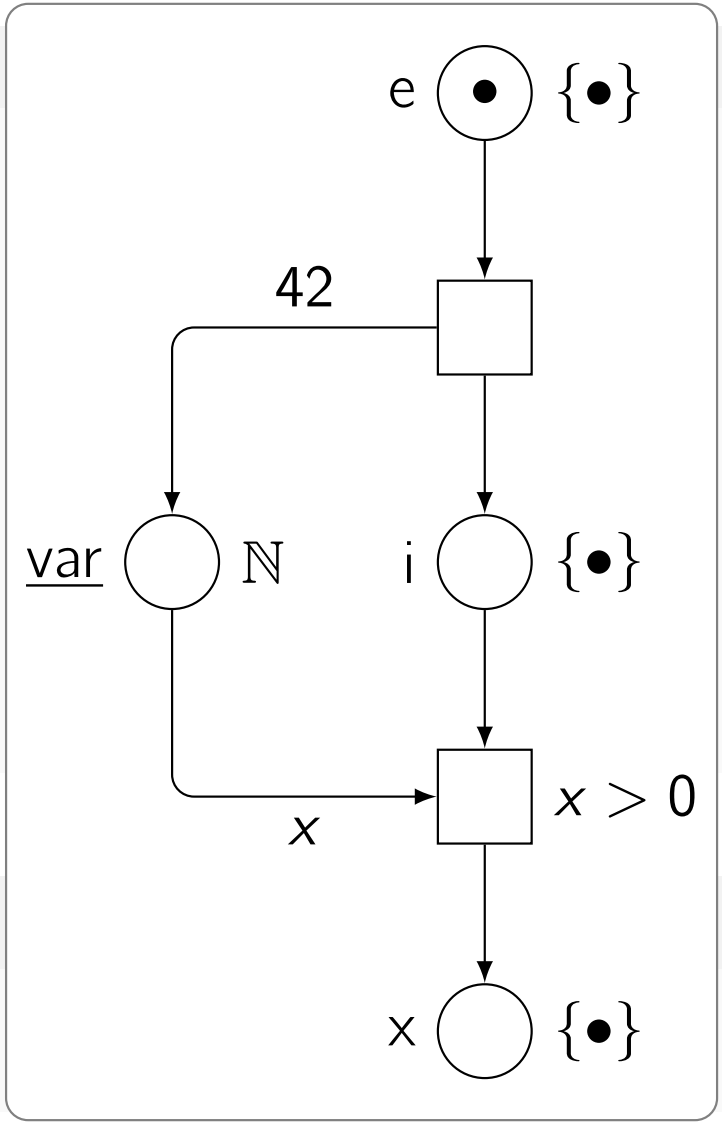
Verification issues

Conclusion

Separating data places from control-flow places

Place statuses $\mathbb{S}$

- additional labelling $\sigma : S \to \mathbb{S}$
- control-flow places
  - entry e
  - internal i
  - exit x
- data places
  - anonymous ε
  - named buffer, count, ...



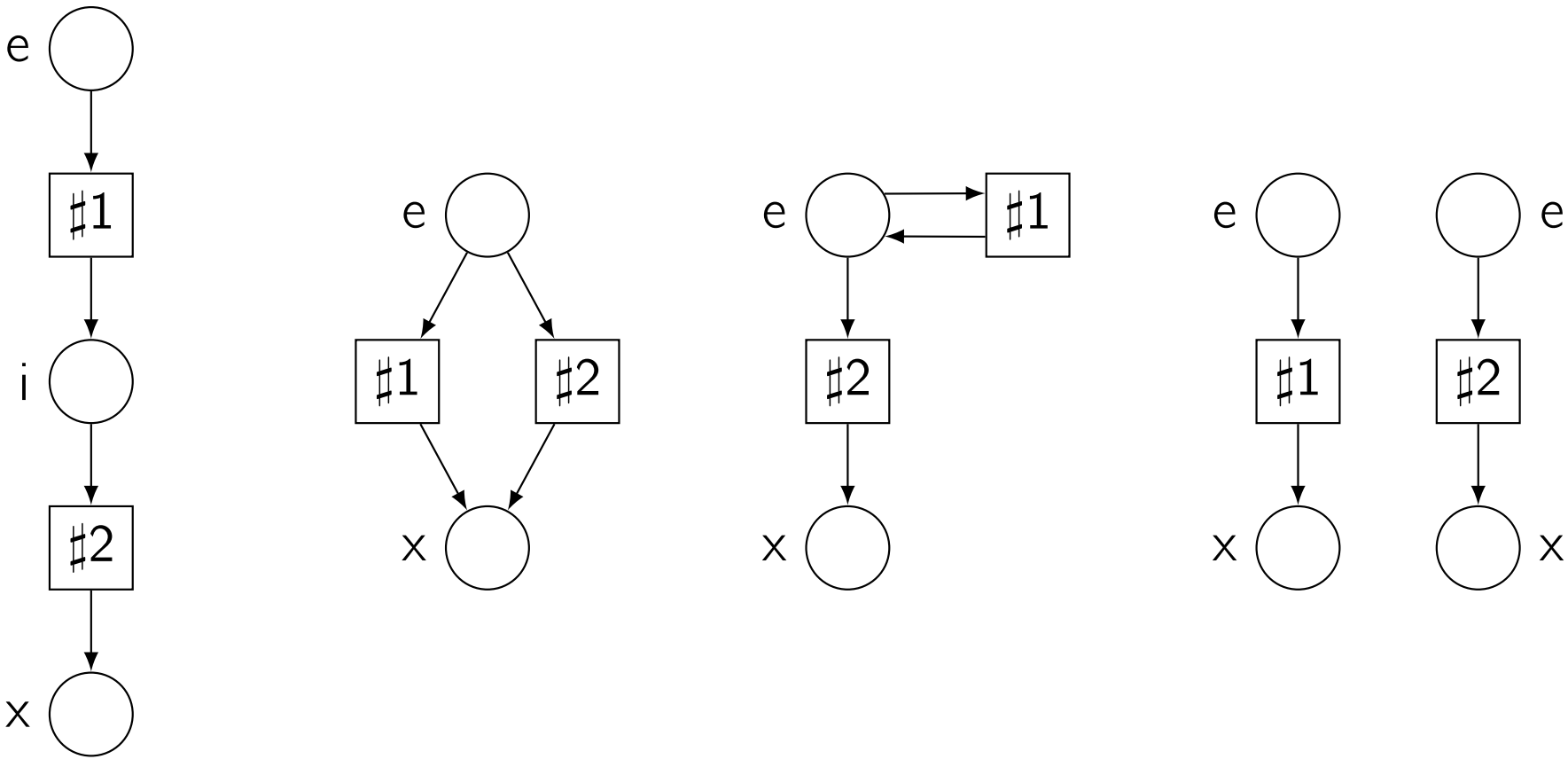
Syntactical restriction

- if $\sigma(s) \in \{\mathsf{e}, \mathsf{i}, \mathsf{x}\}$ then $\ell(s) = \{\bullet\}$

Specifying control-flow operators

with operator nets

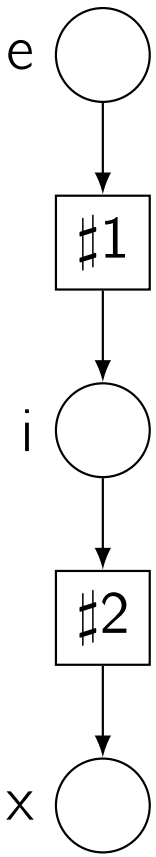
sequence $\fatsemi$ choice $\Box$ iteration $\circledast$ parallel $\|$



Specifying control-flow operators

with operator nets

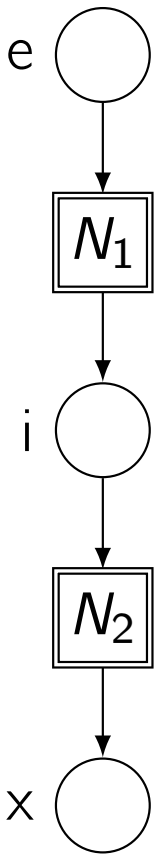
sequence $\fatsemi$



Specifying control-flow operators

with operator nets

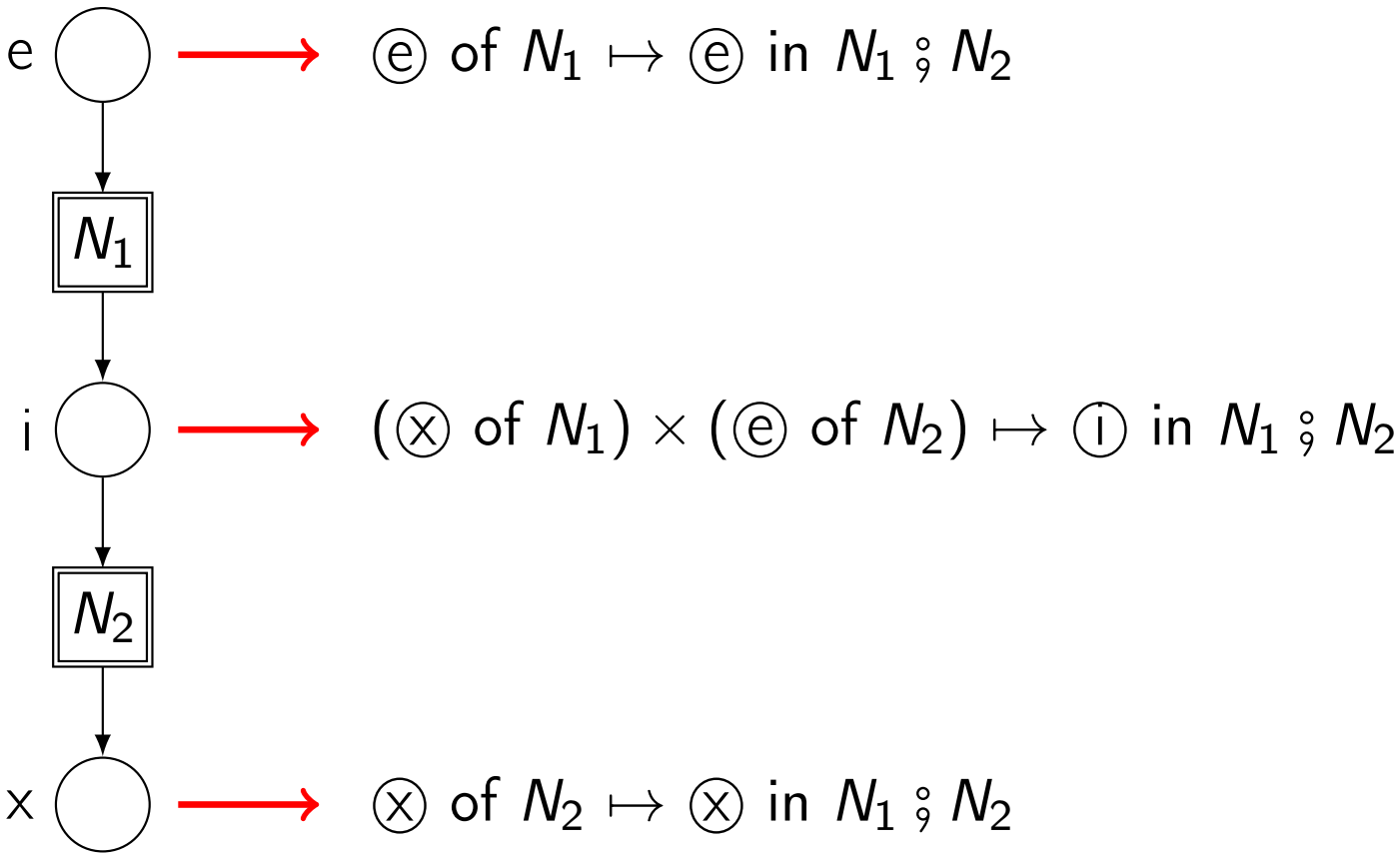
sequence $\fatsemi$



Specifying control-flow operators

with operator nets

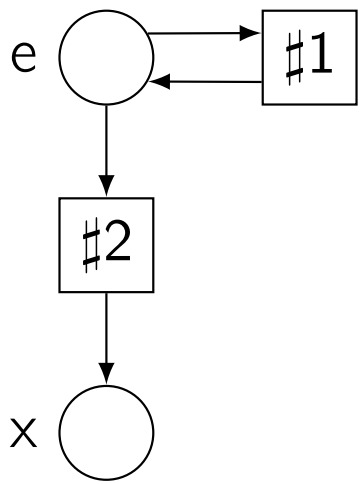
sequence $\fatsemi$



Specifying control-flow operators

with operator nets

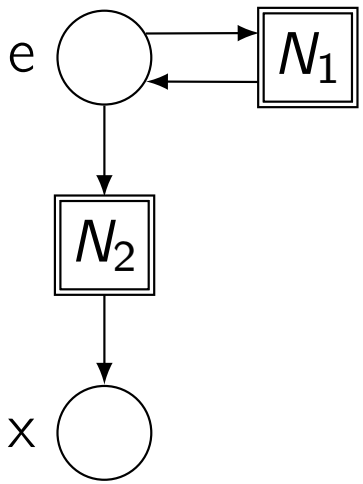
iteration $\circledast$



Specifying control-flow operators

with operator nets

iteration $\circledast$

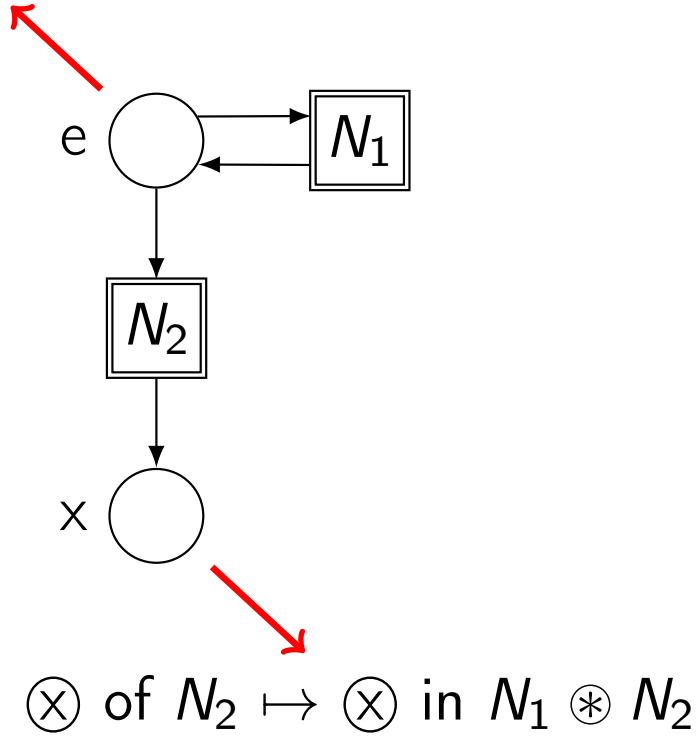


Specifying control-flow operators

with operator nets

iteration $\circledast$

$(\textcircled{e} \text{ of } N_1) \times (\textcircled{e} \text{ of } N_2) \times (\textcircled{x} \text{ of } N_1) \mapsto \textcircled{e} \text{ in } N_1 \circledast N_2$

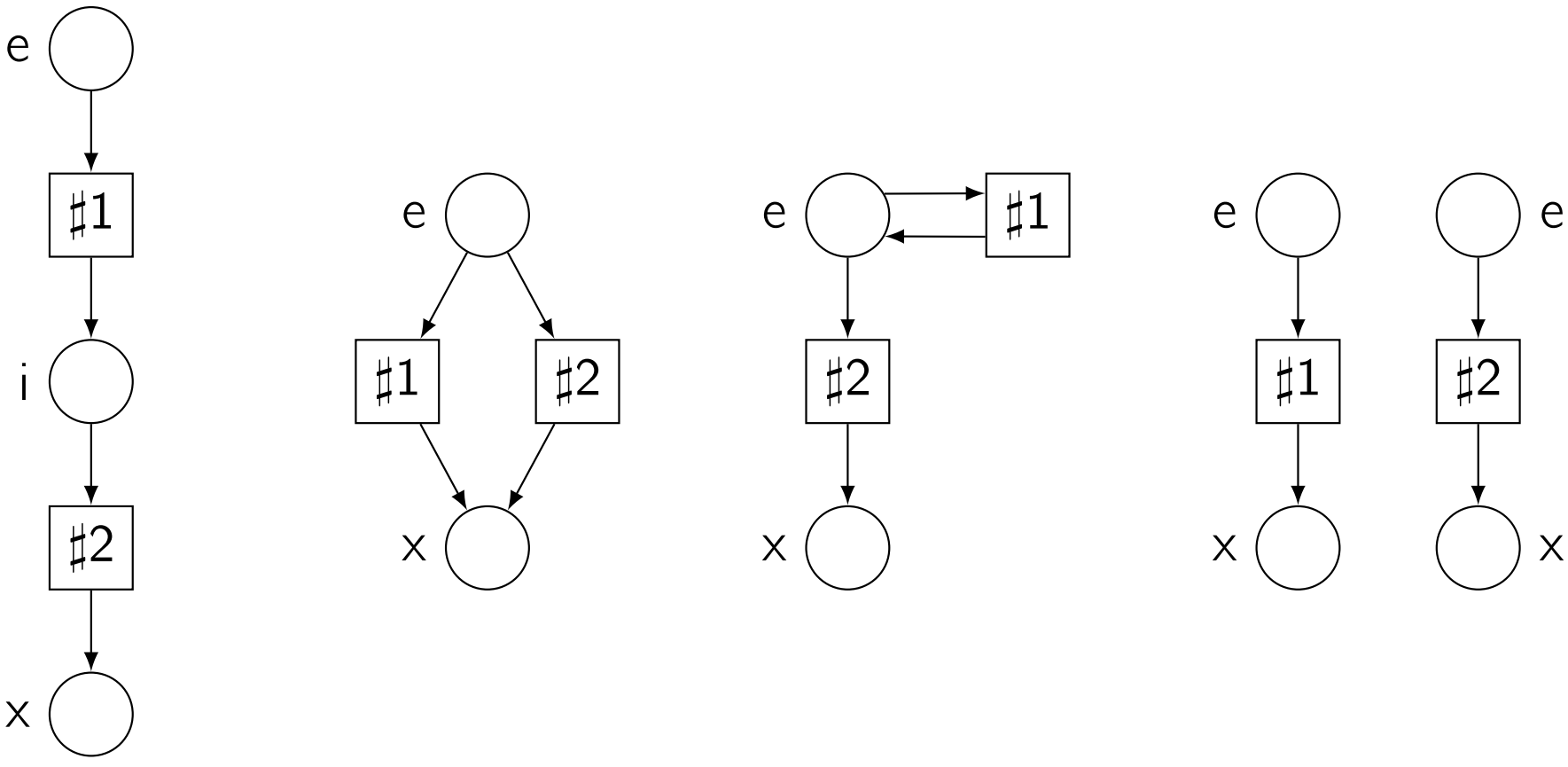


$\textcircled{x} \text{ of } N_2 \mapsto \textcircled{x} \text{ in } N_1 \circledast N_2$

Specifying control-flow operators

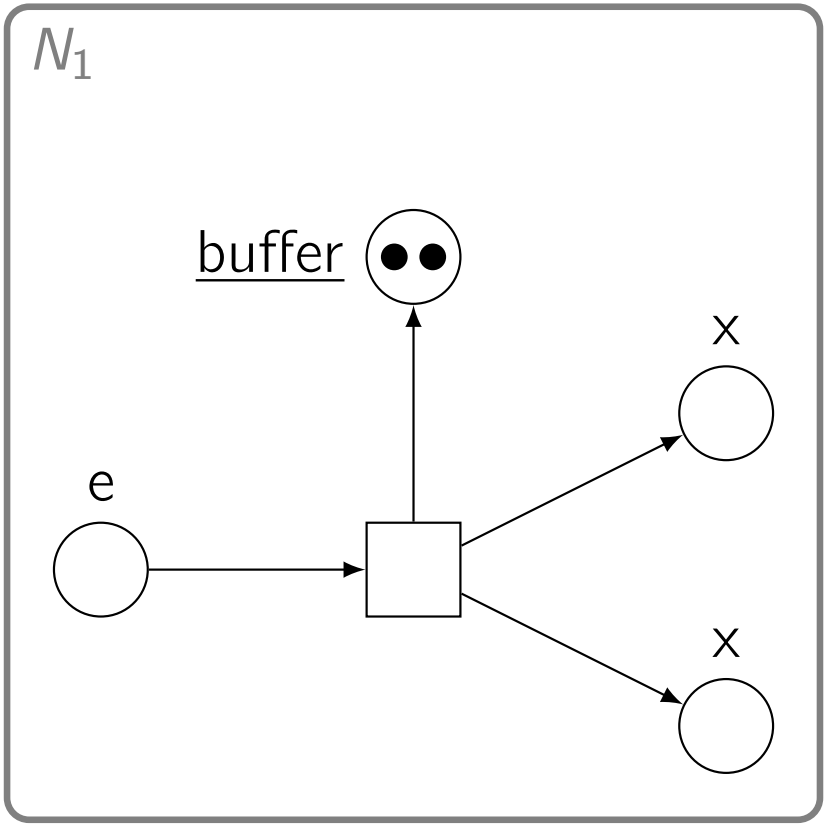
with operator nets

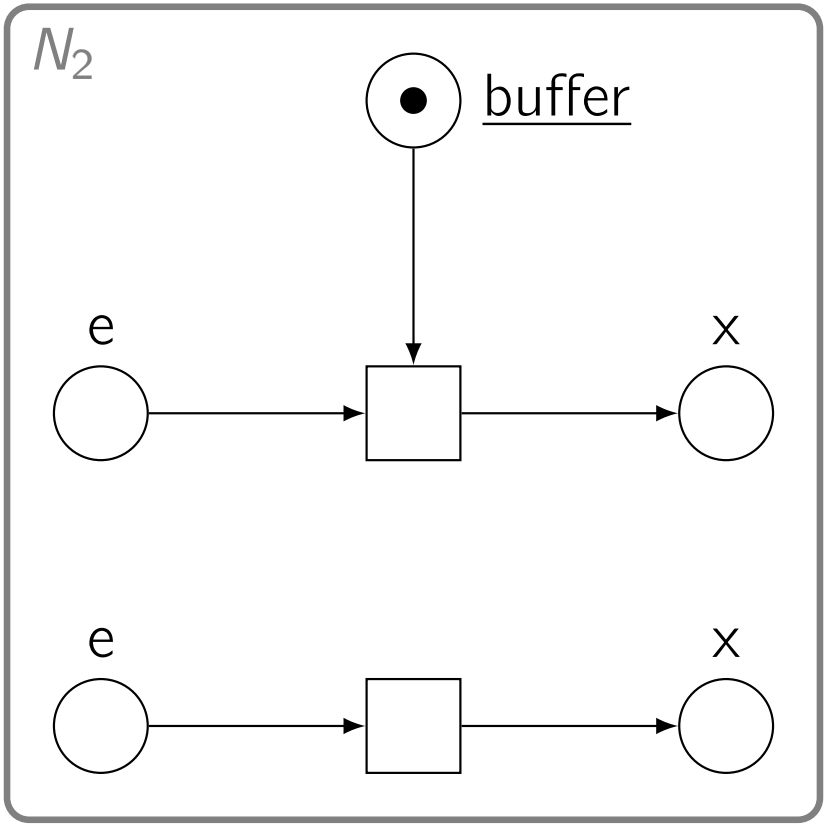
sequence $\fatsemi$ choice $\Box$ iteration $\circledast$ parallel $\|$



Composing nets

Example: sequential composition

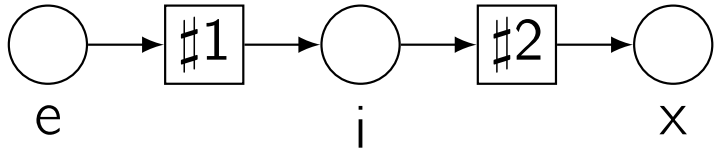


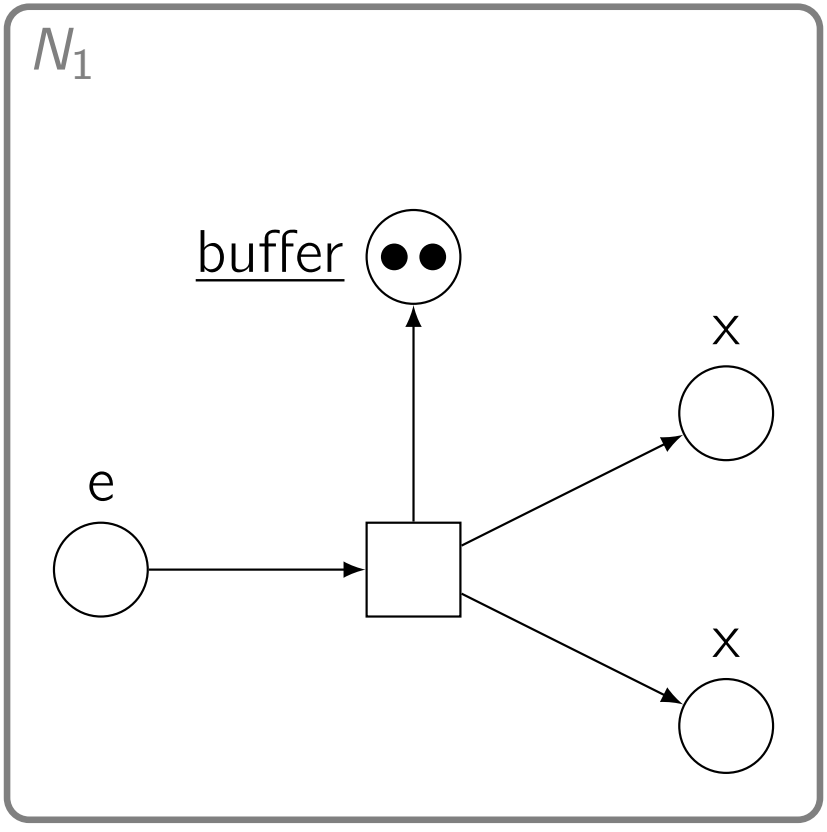


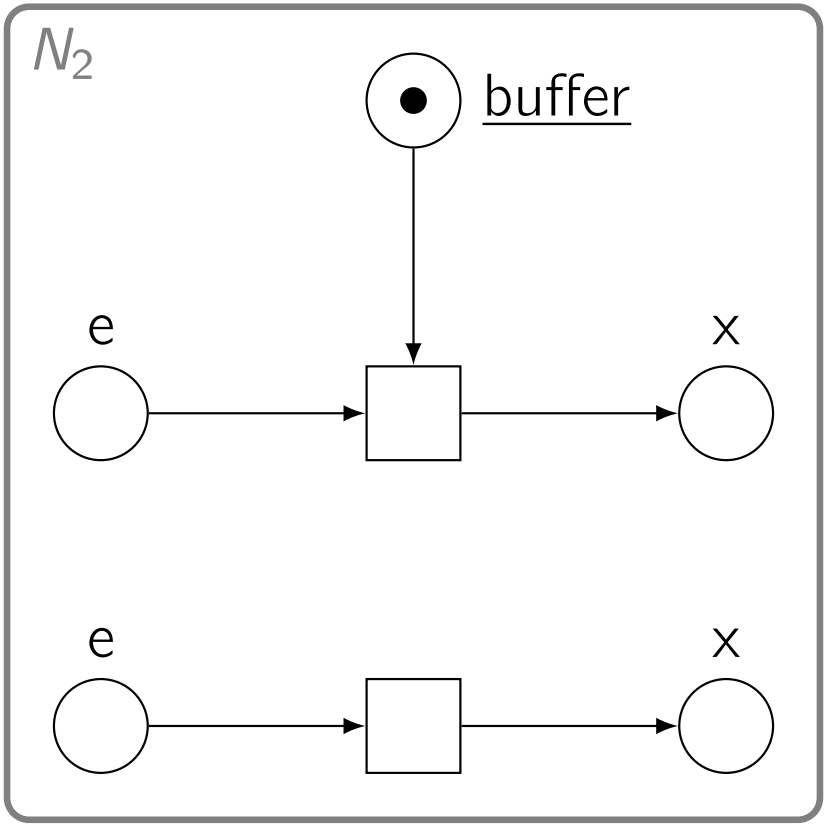
Composing nets

Example: sequential composition

- gluing phase



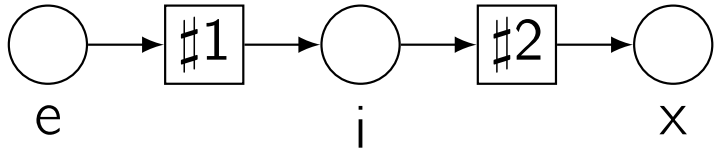


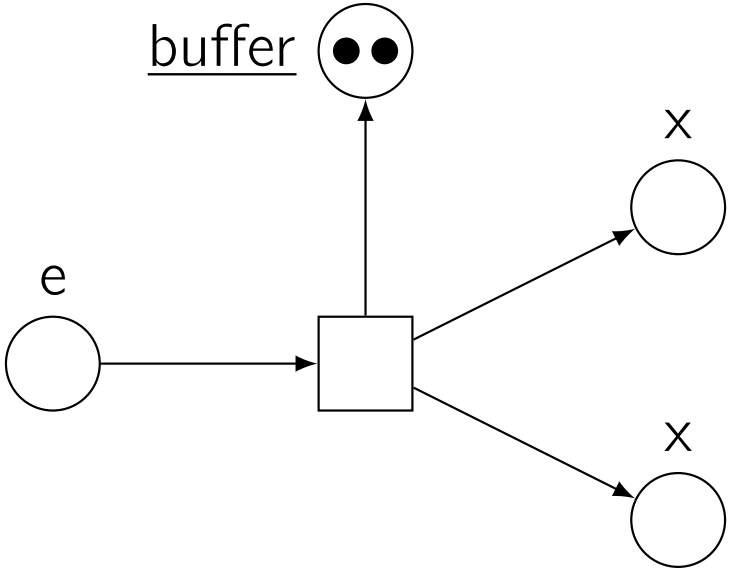


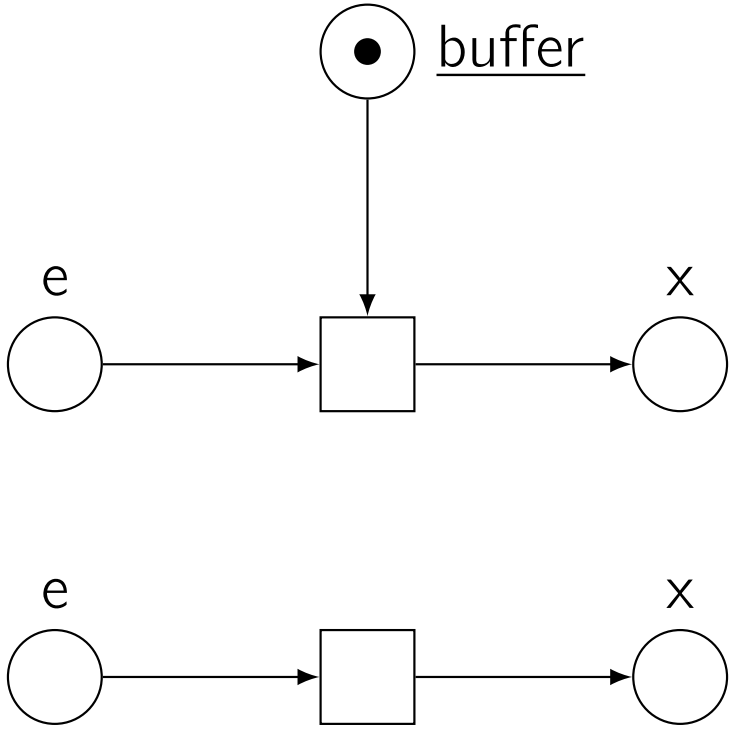
Composing nets

Example: sequential composition

- gluing phase



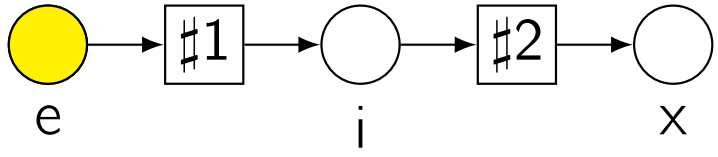


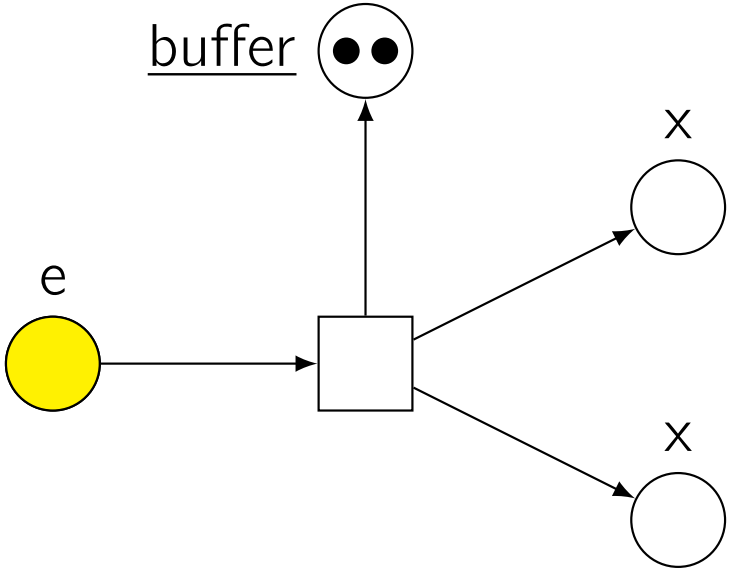


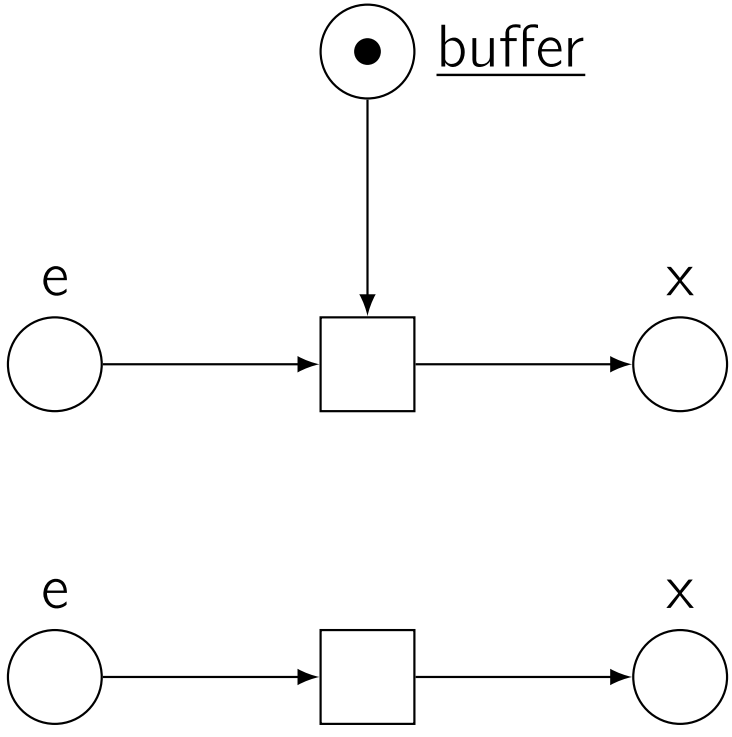
Composing nets

Example: sequential composition

- gluing phase



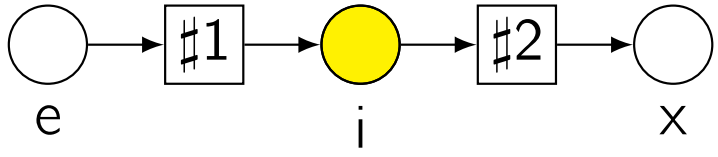


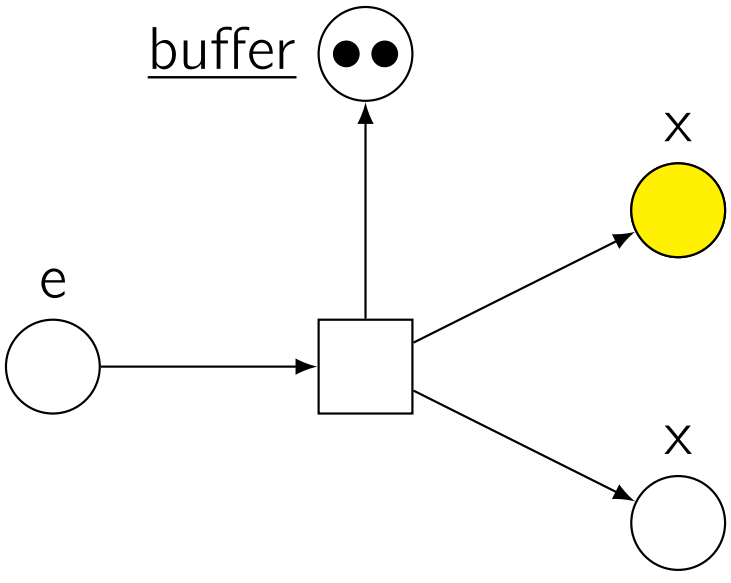


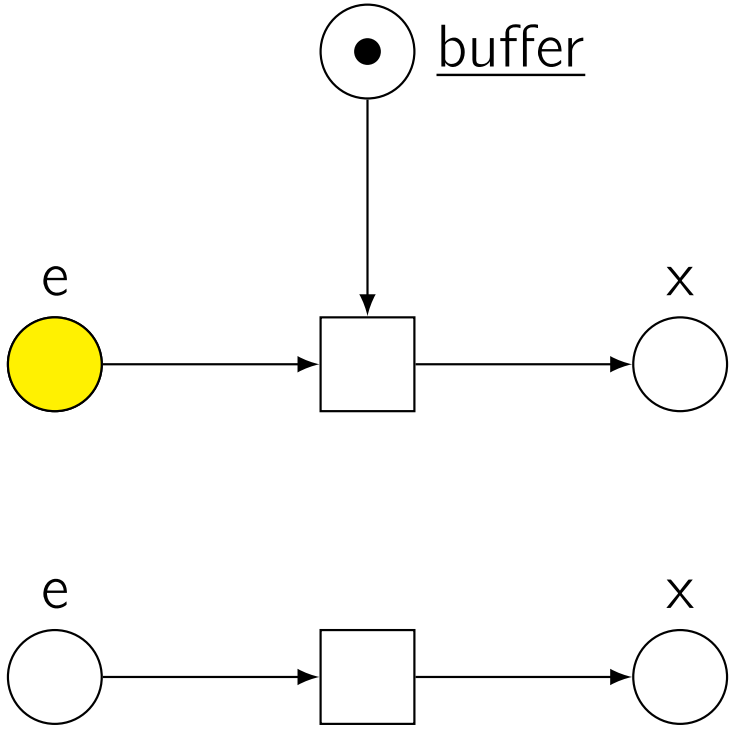
Composing nets

Example: sequential composition

- gluing phase



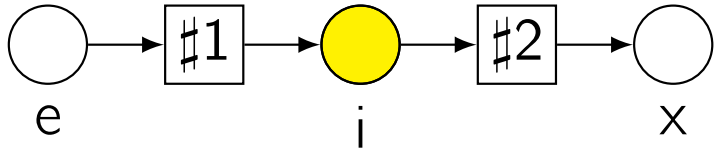


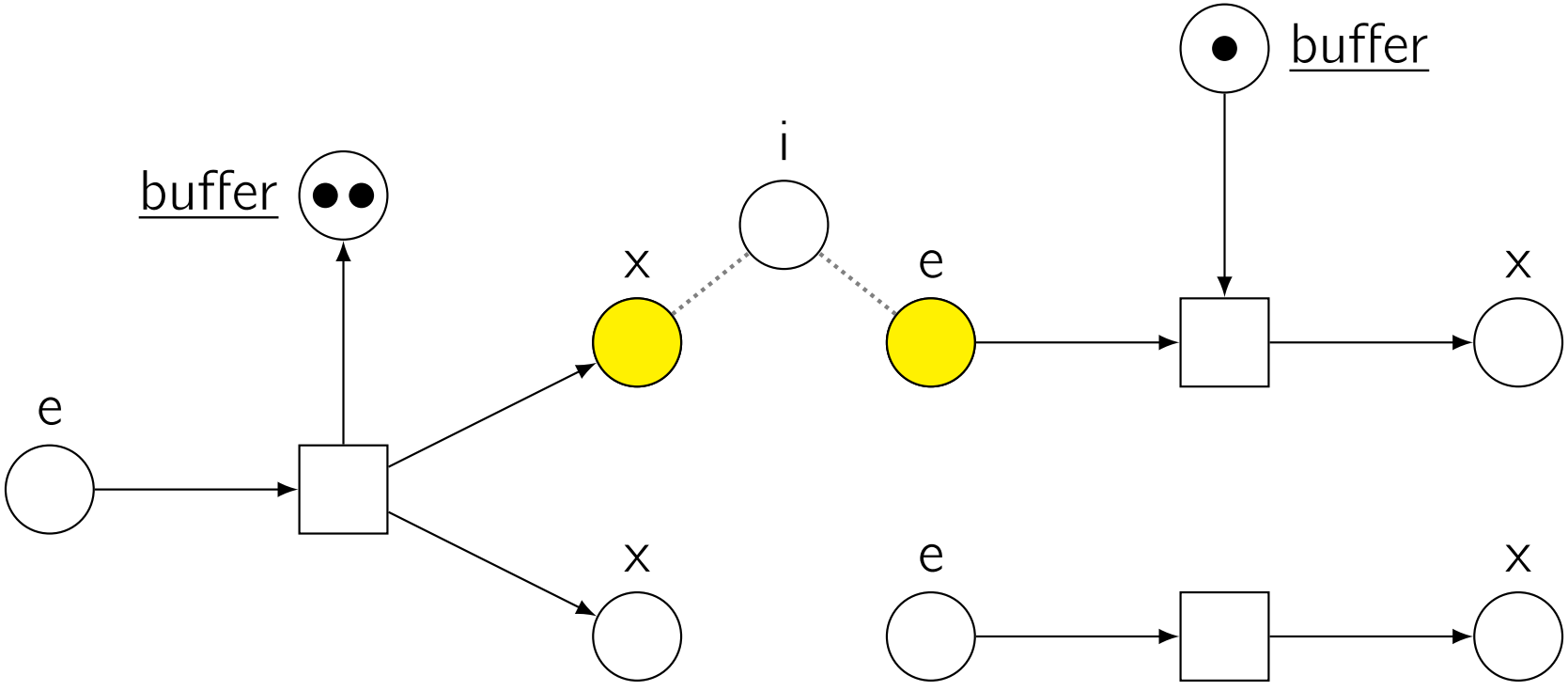


Composing nets

Example: sequential composition

- gluing phase

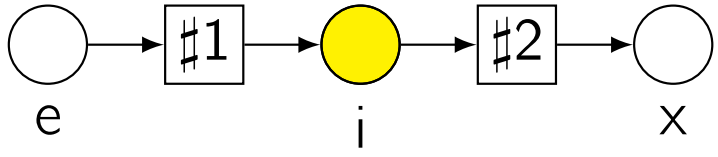


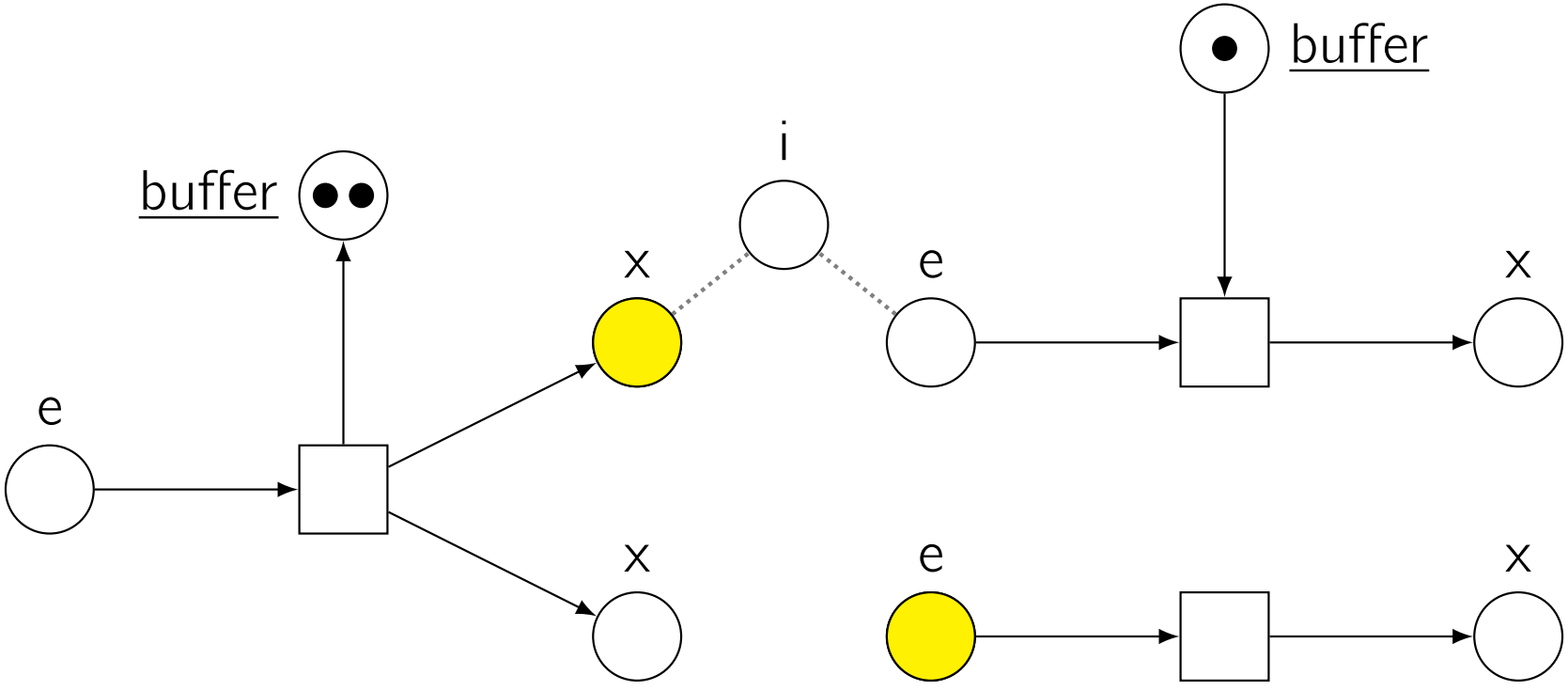


Composing nets

Example: sequential composition

- gluing phase

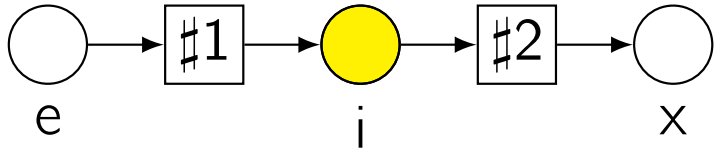


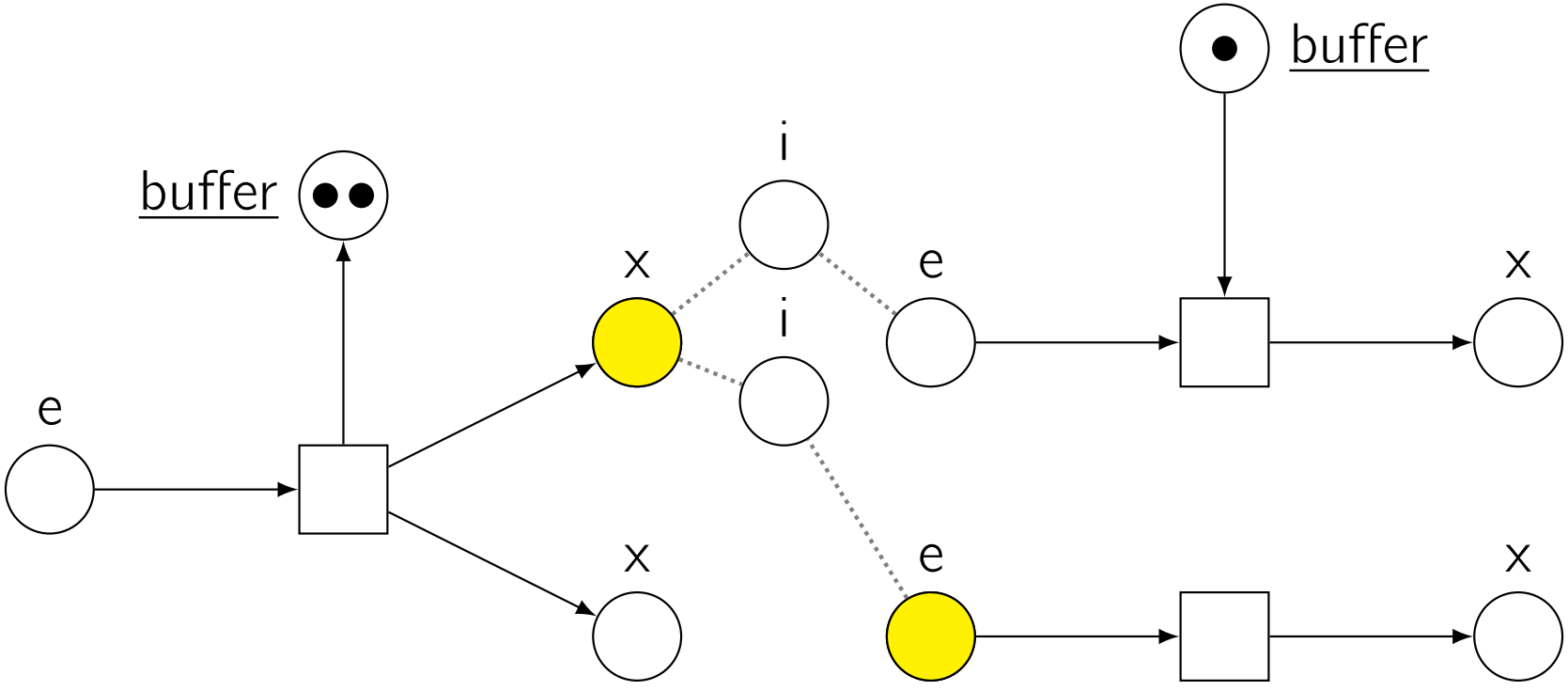


Composing nets

Example: sequential composition

- gluing phase

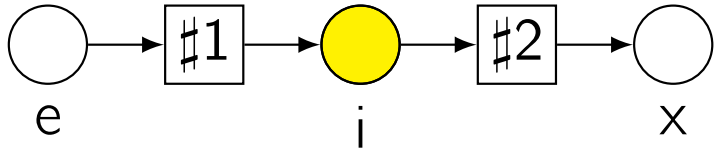


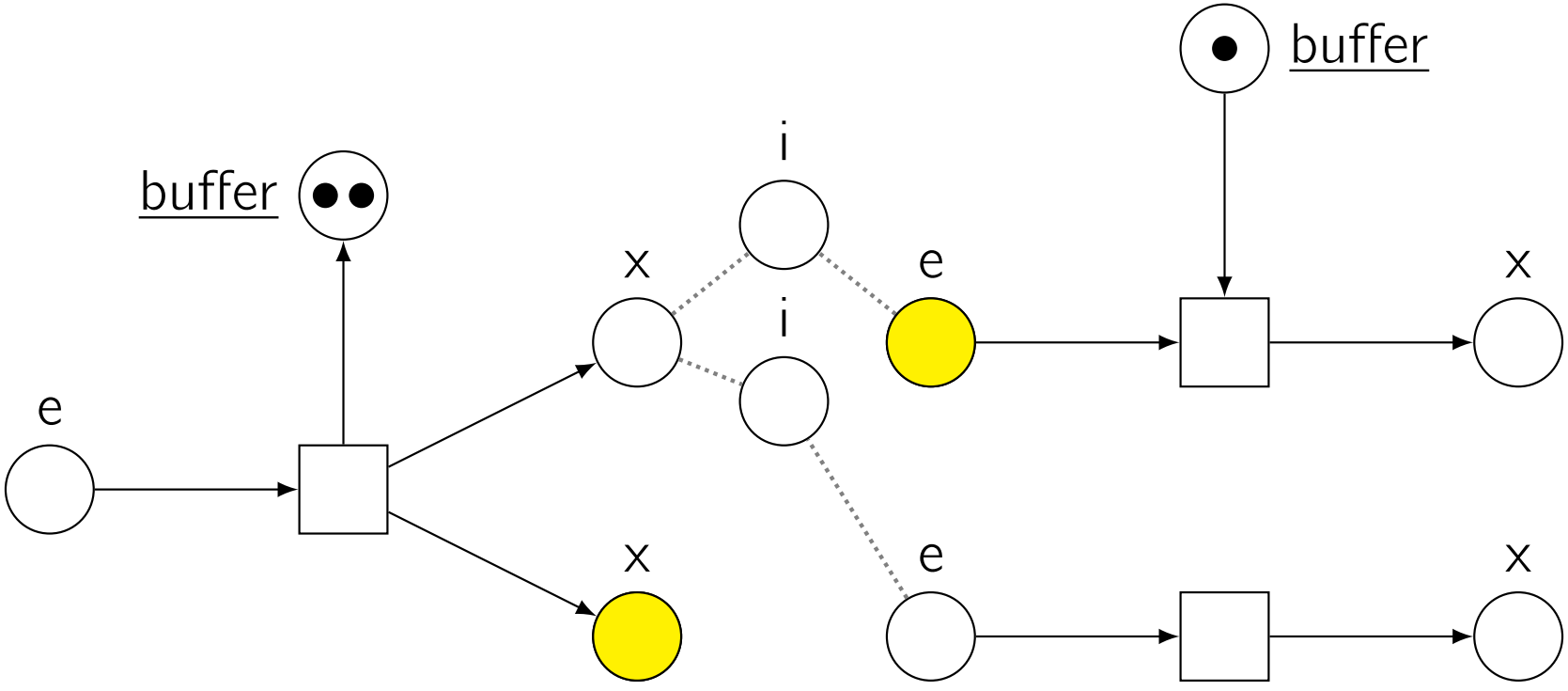


Composing nets

Example: sequential composition

- gluing phase

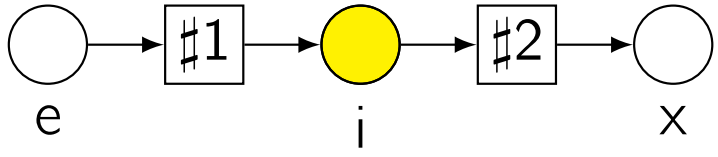


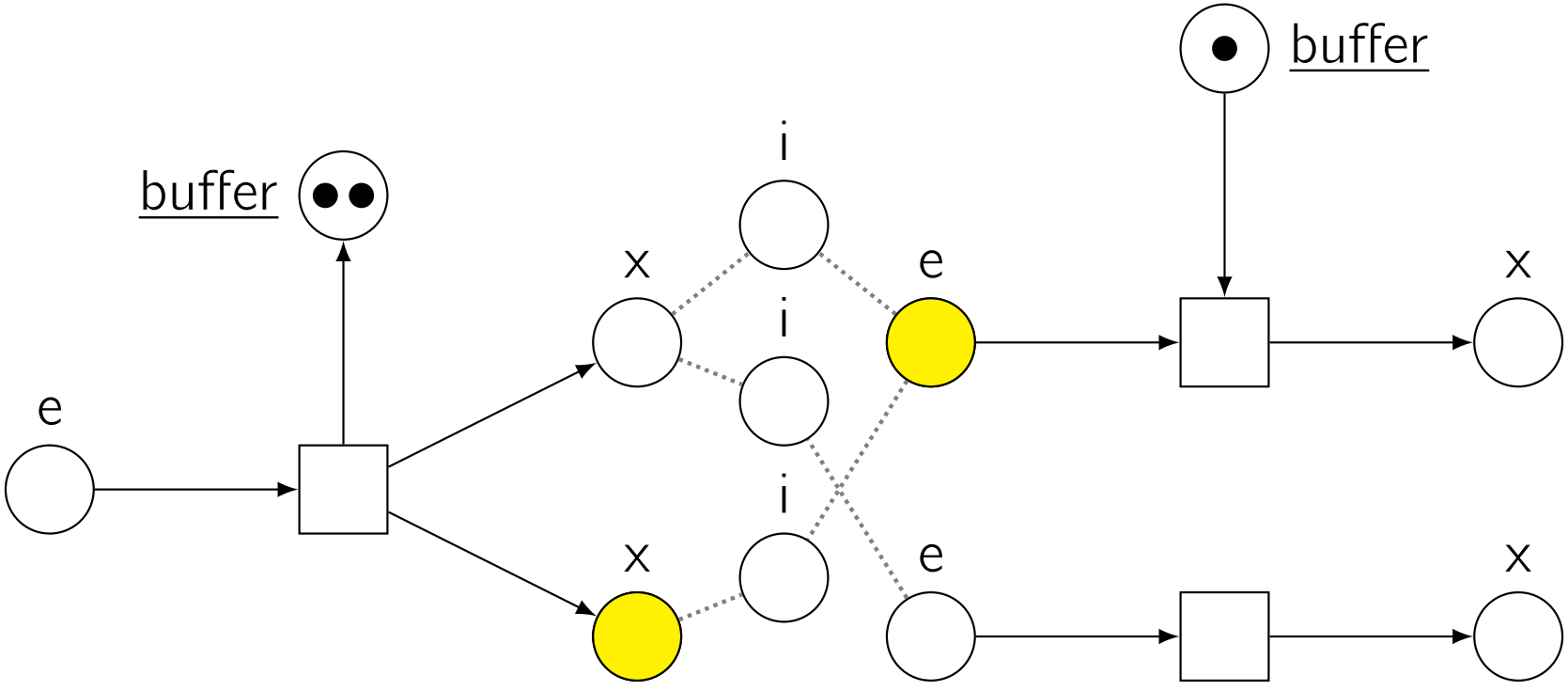


Composing nets

Example: sequential composition

- gluing phase

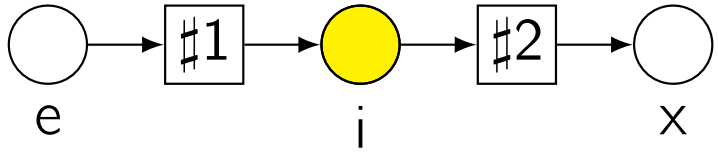


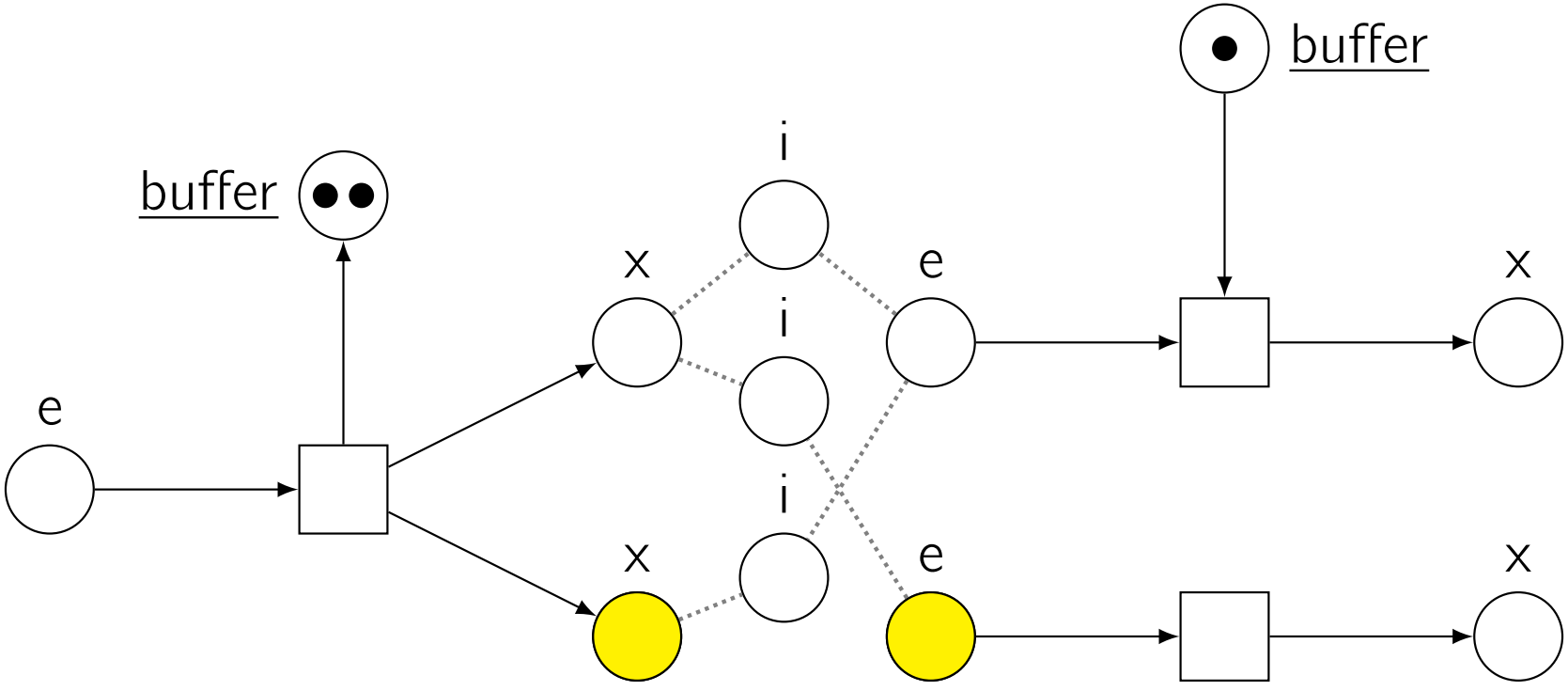


Composing nets

Example: sequential composition

- gluing phase

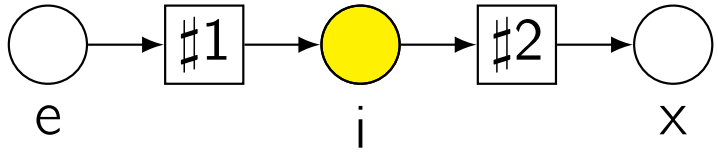


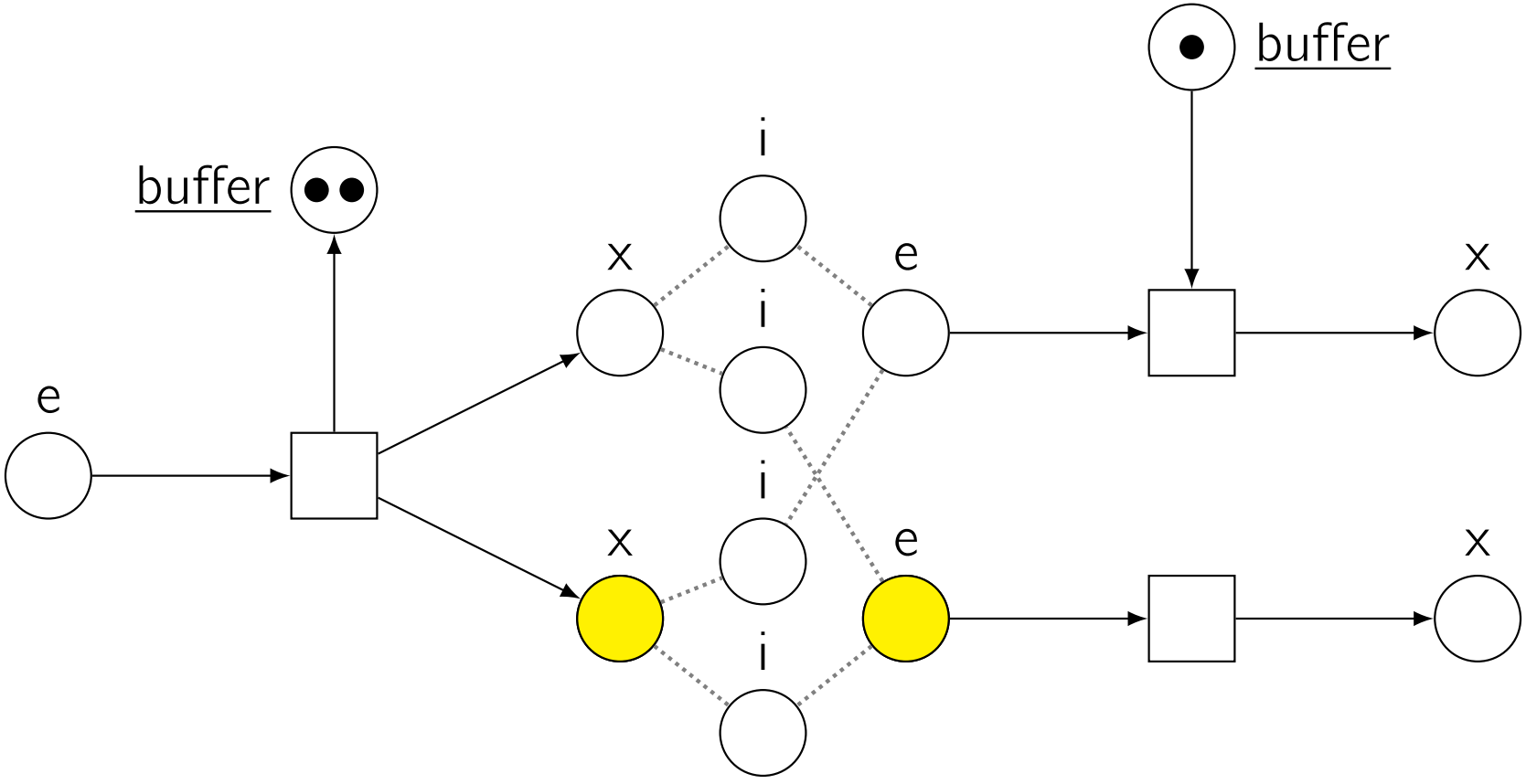


Composing nets

Example: sequential composition

- gluing phase

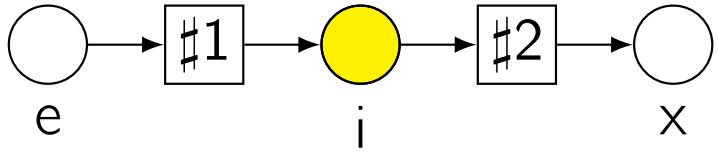


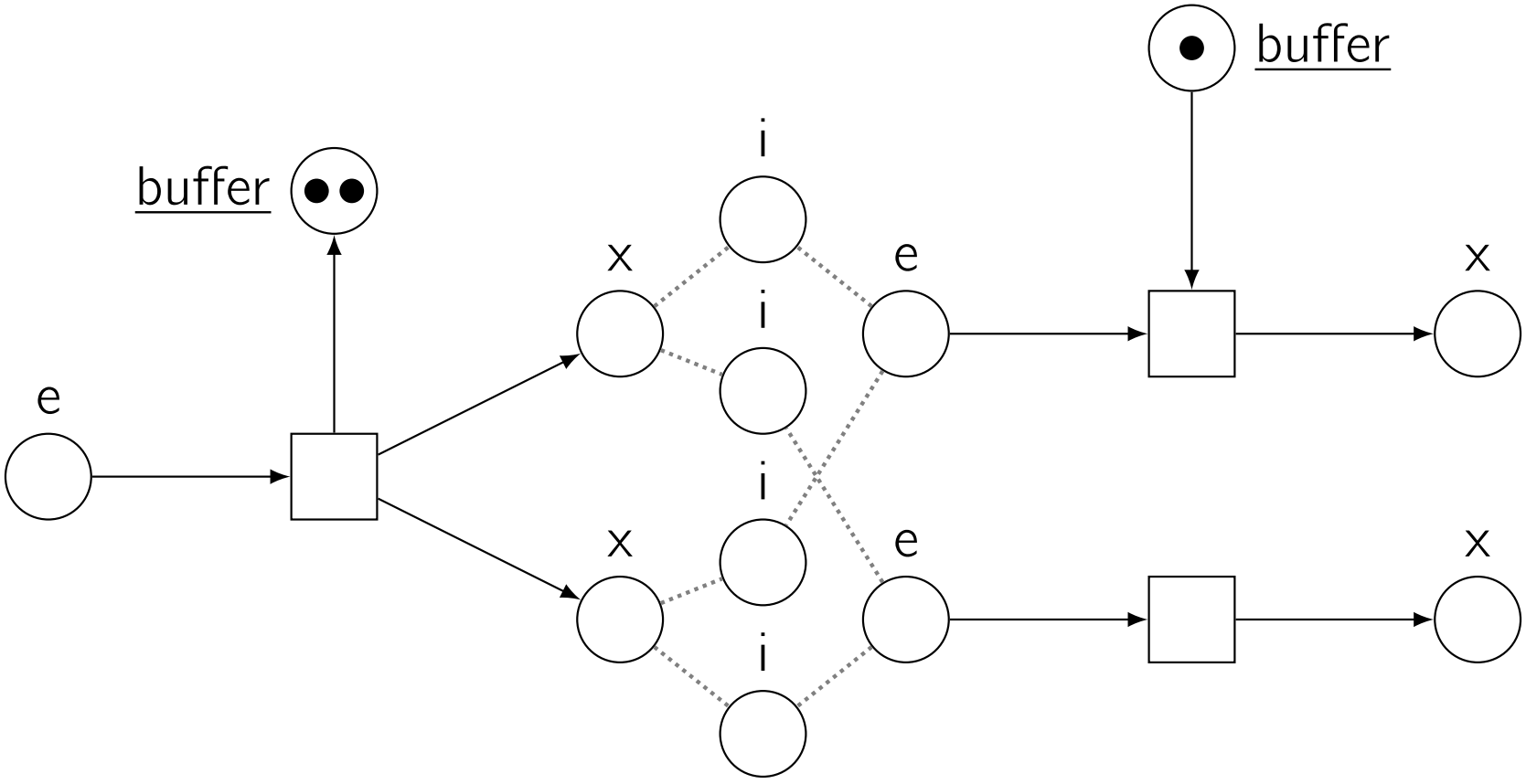


Composing nets

Example: sequential composition

- gluing phase

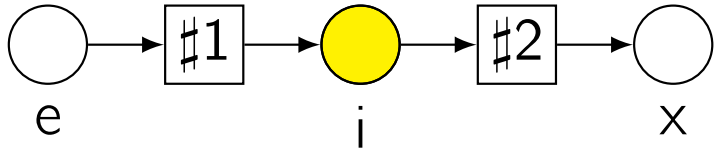


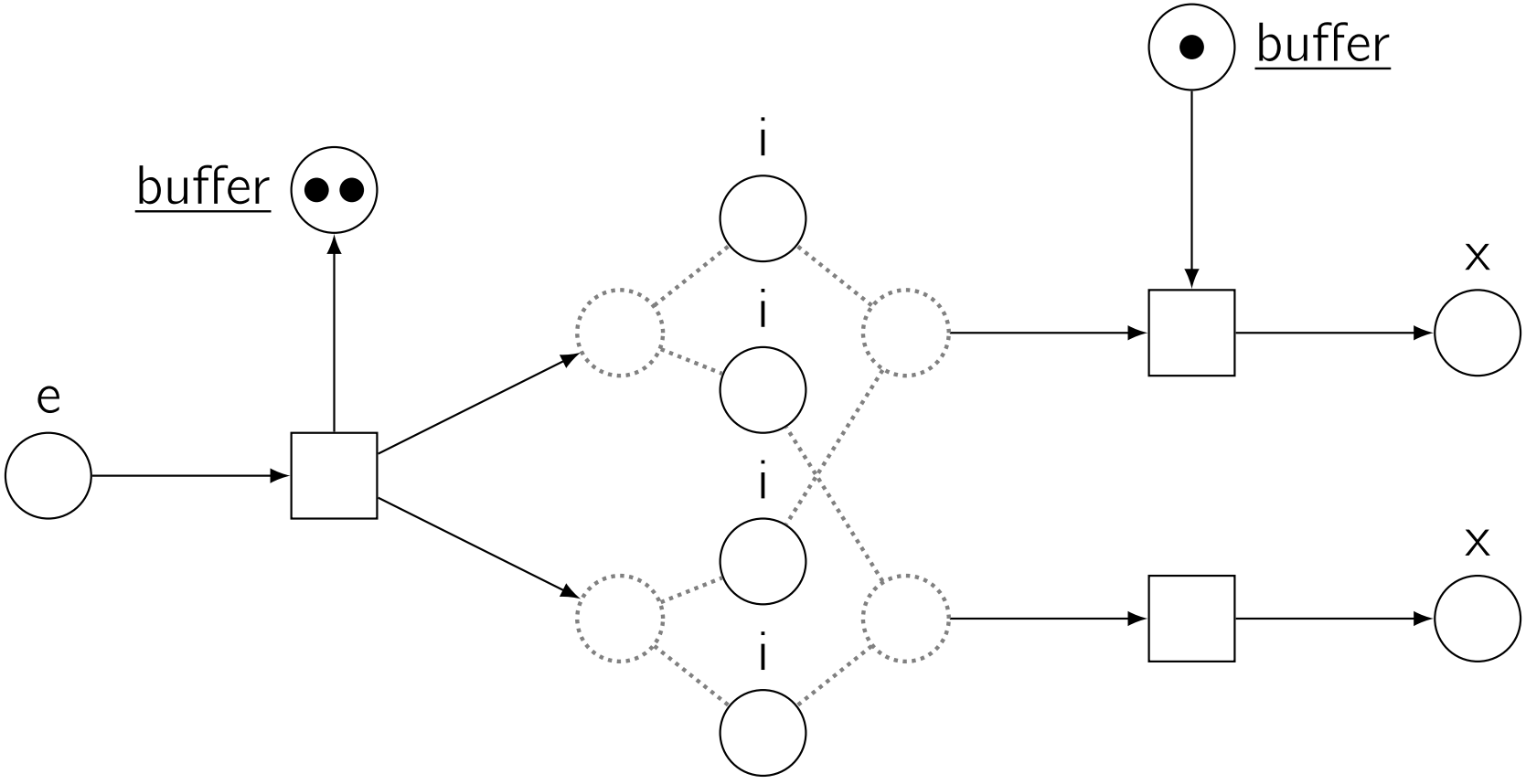


Composing nets

Example: sequential composition

- gluing phase

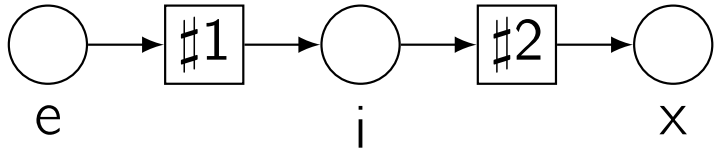


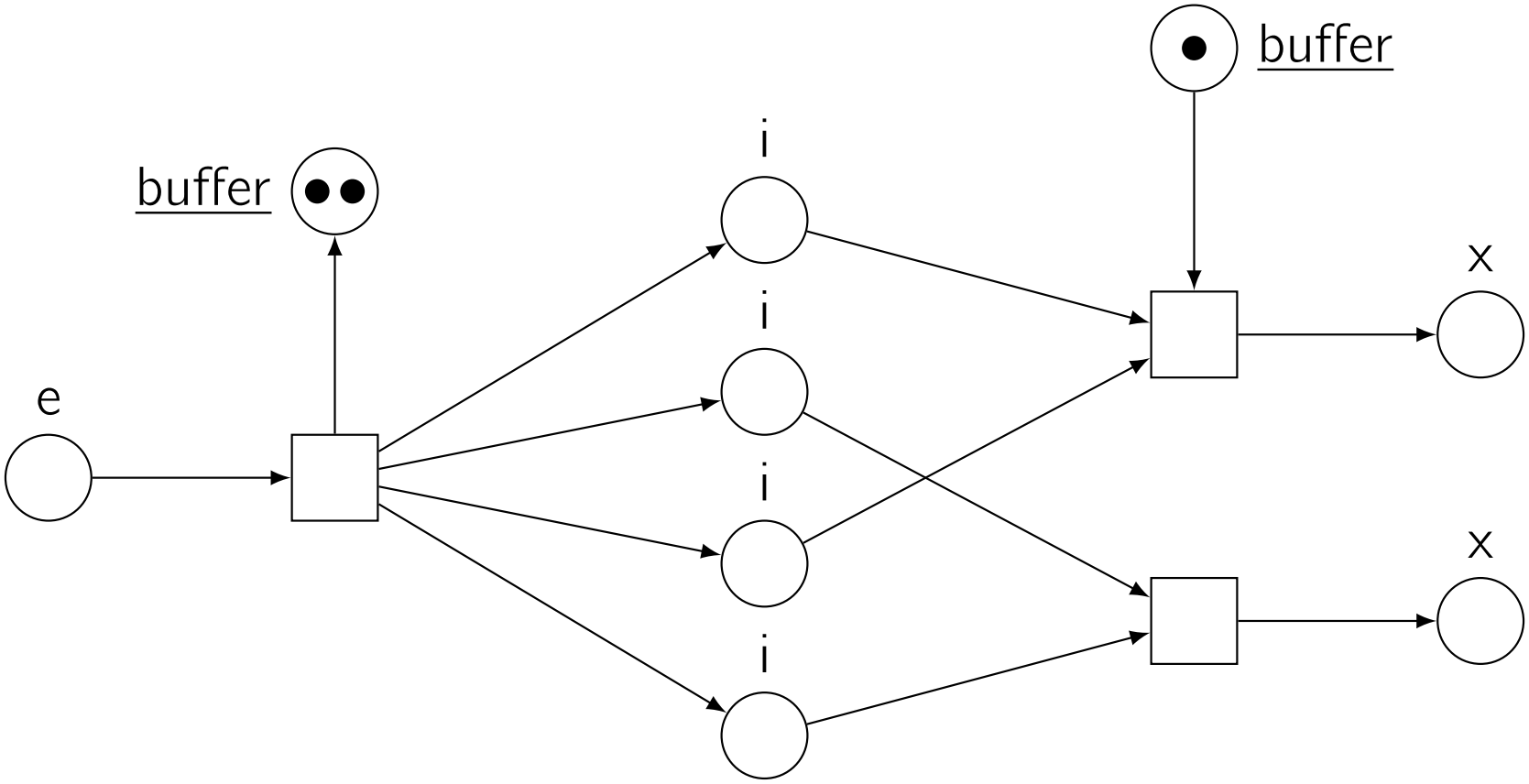


Composing nets

Example: sequential composition

- gluing phase

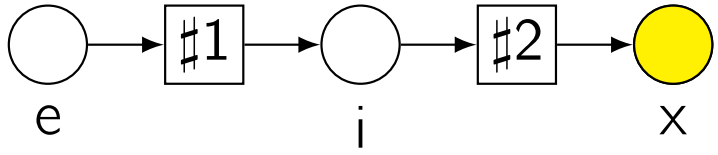


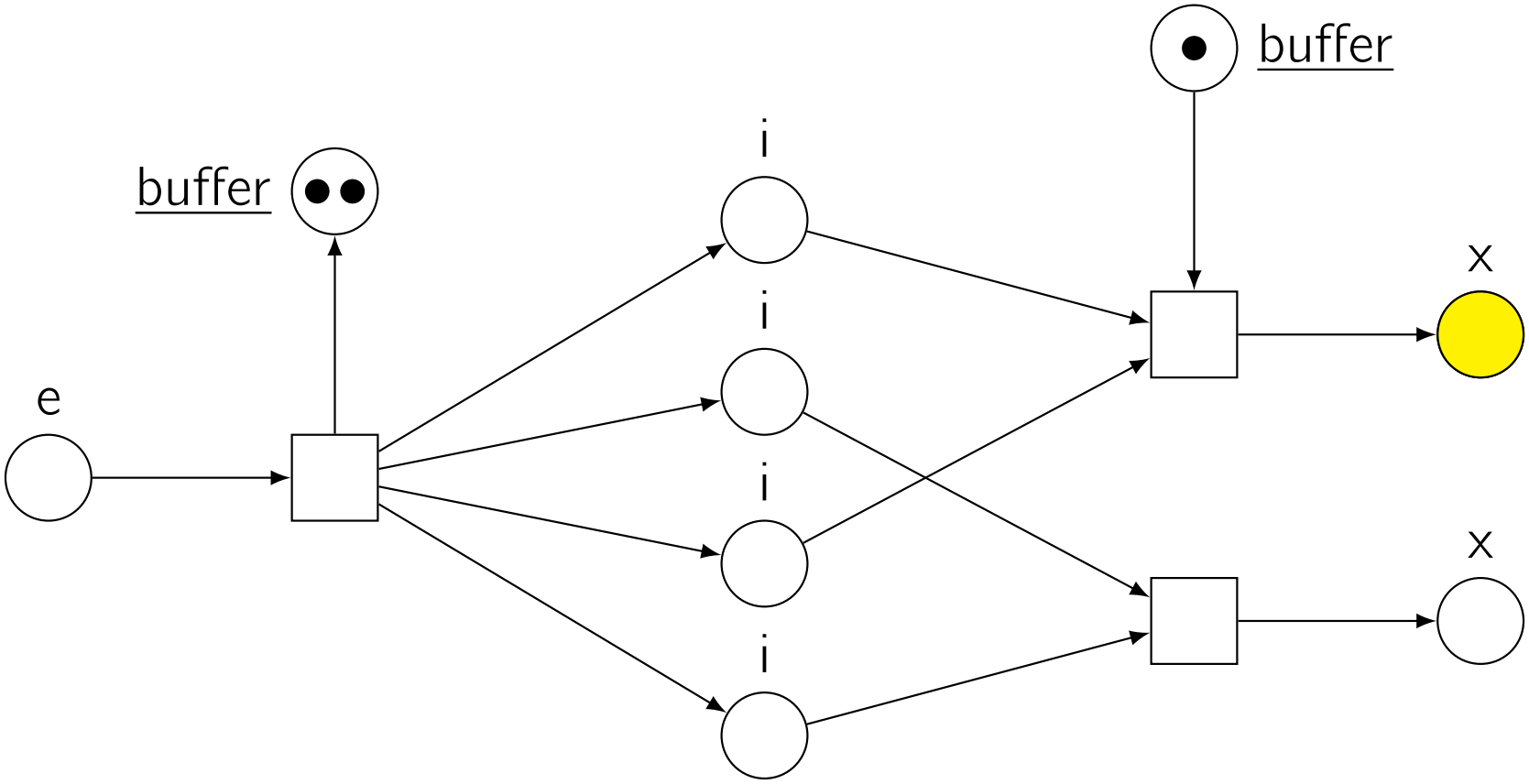


Composing nets

Example: sequential composition

- gluing phase

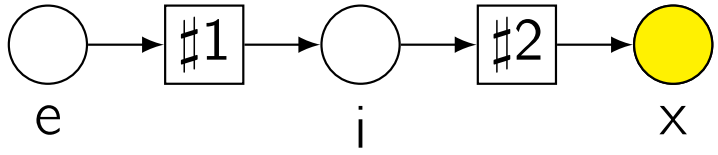


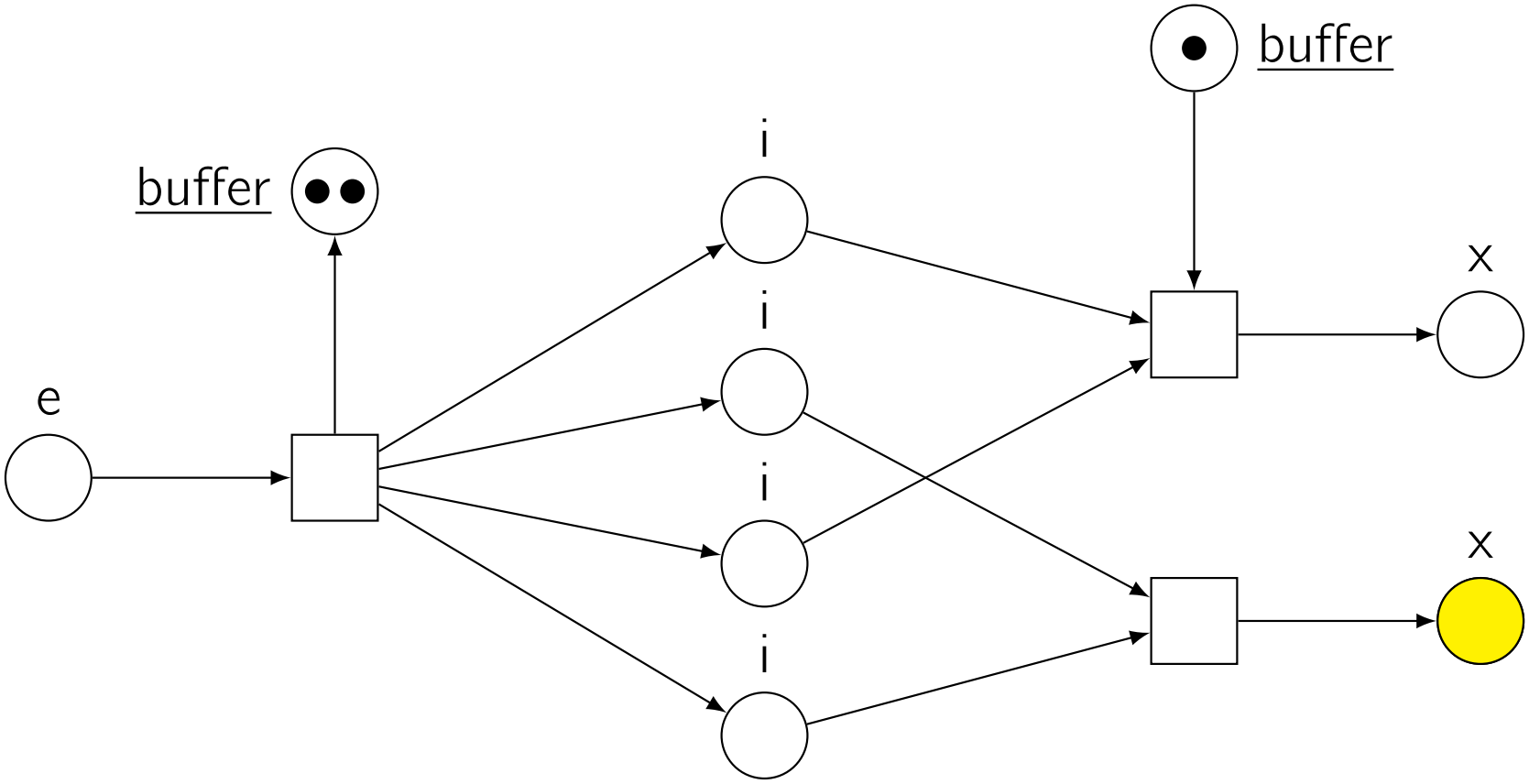


Composing nets

Example: sequential composition

- gluing phase

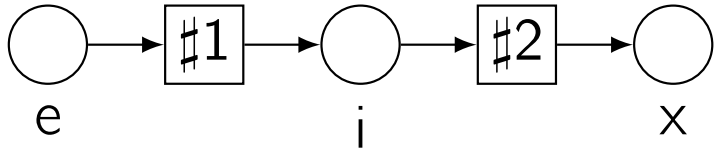


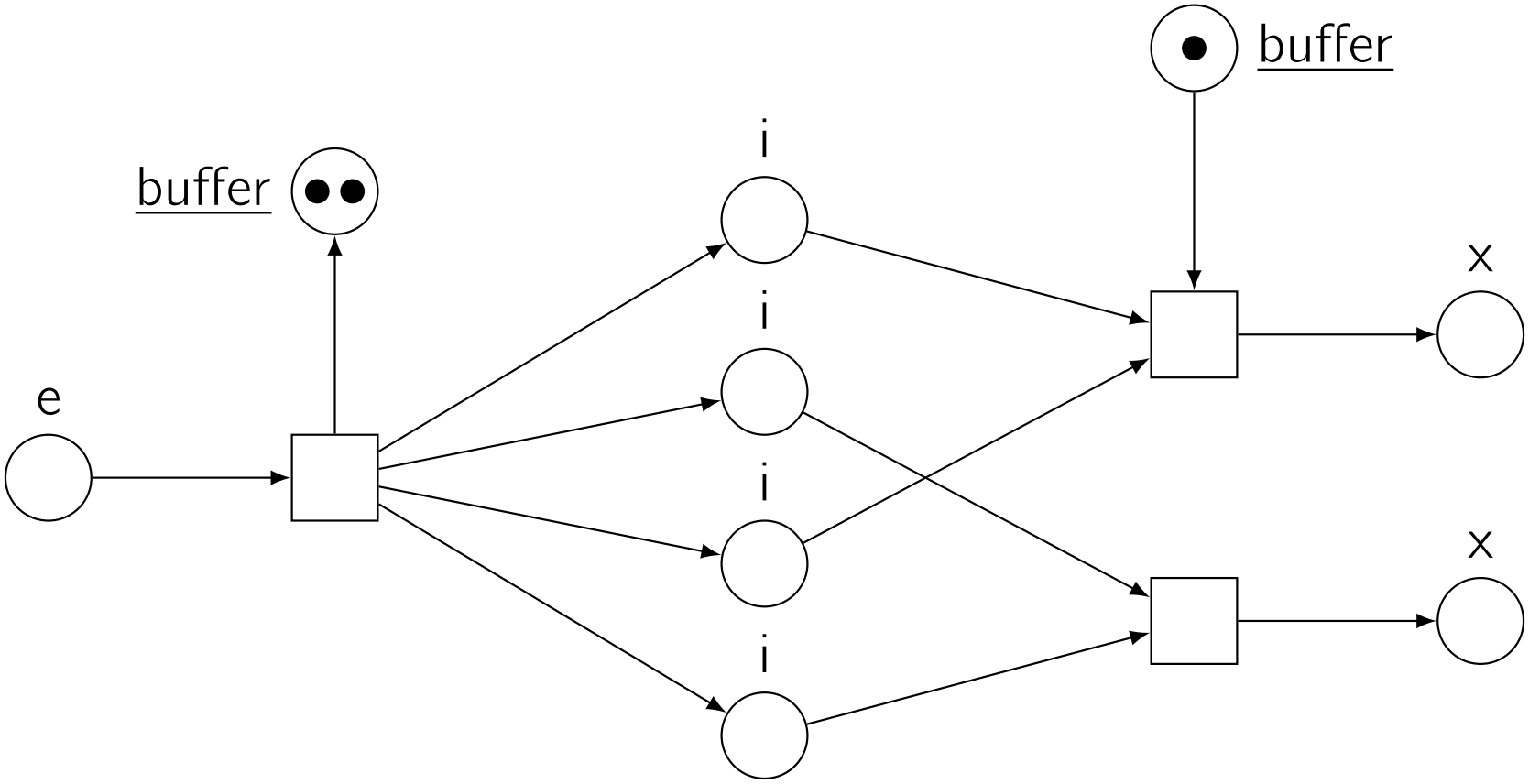


Composing nets

Example: sequential composition

- gluing phase

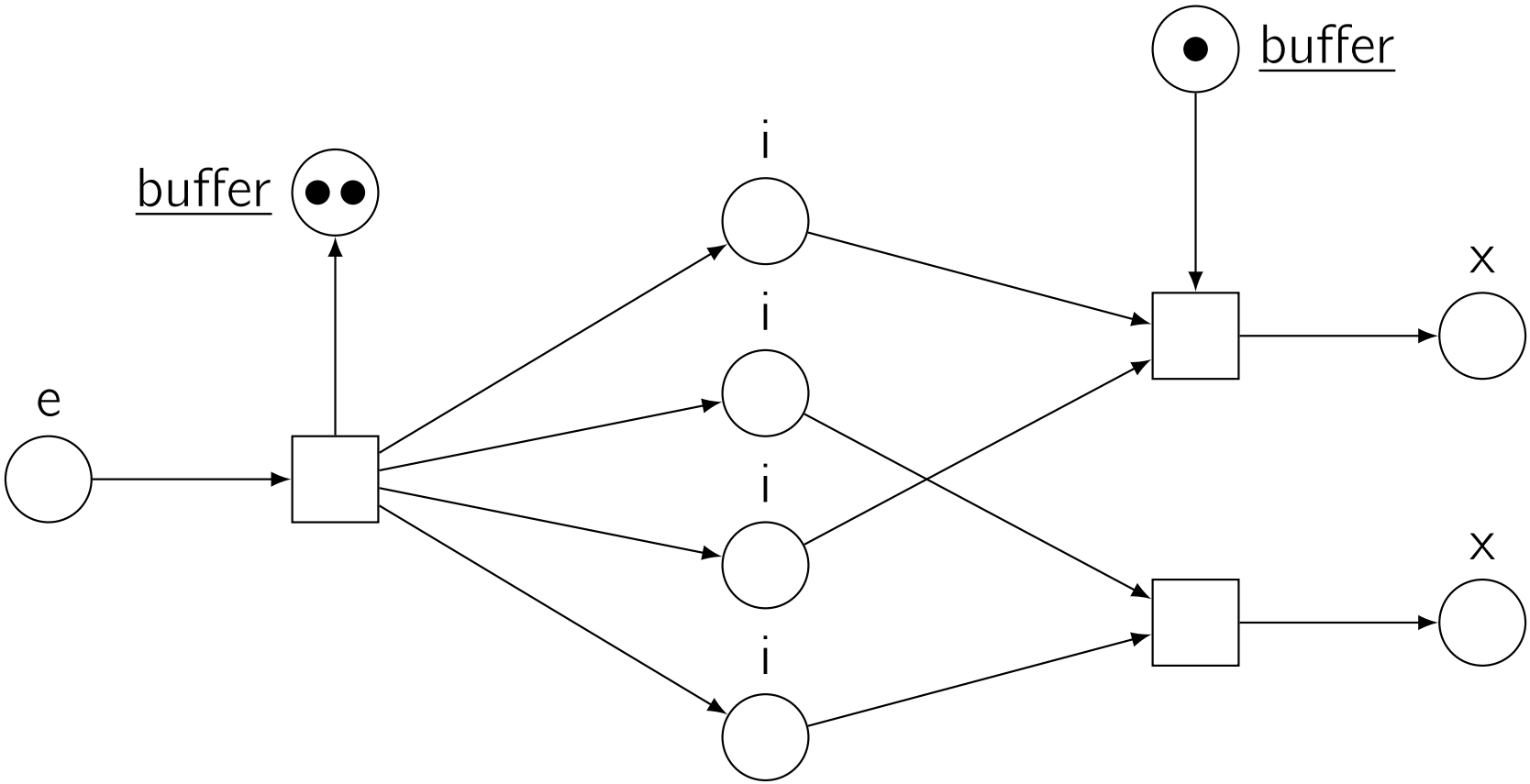




Composing nets

Example: sequential composition

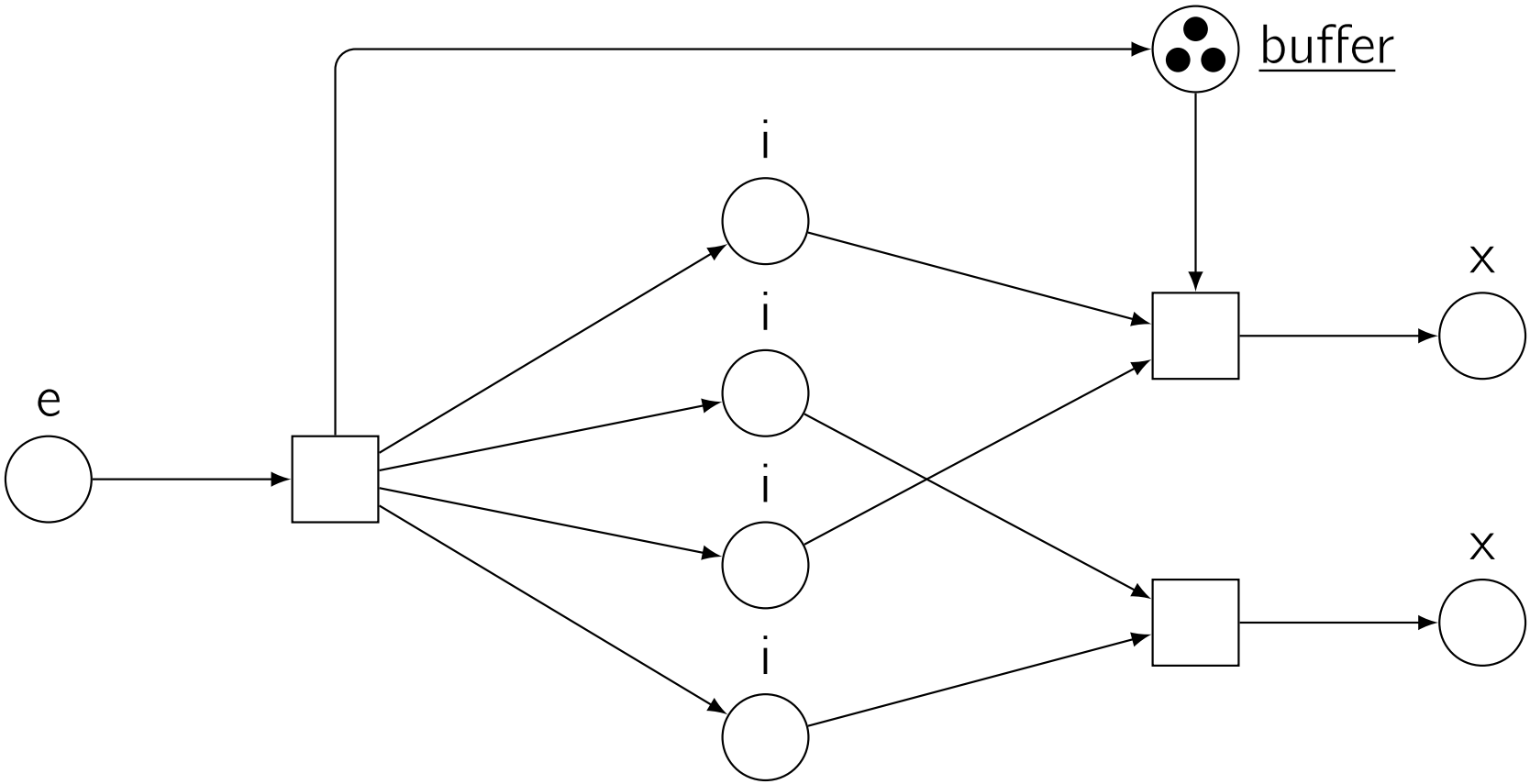
- gluing phase
- merging phase



Composing nets

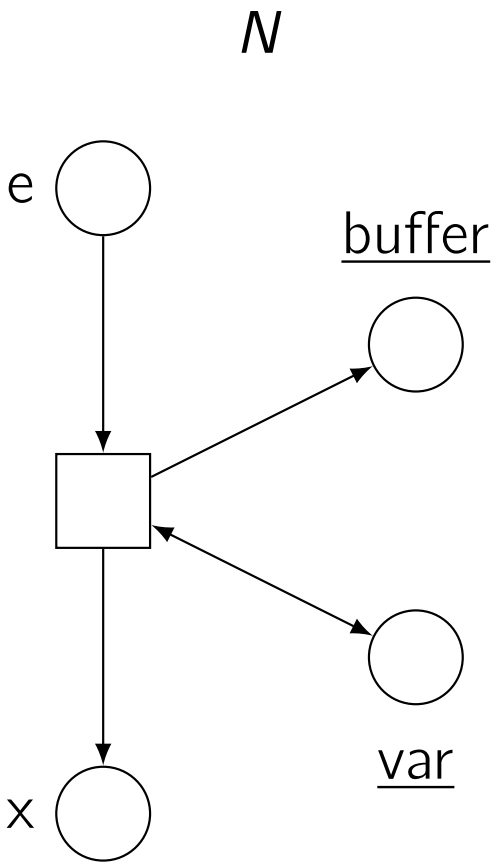
Example: sequential composition

- gluing phase
- merging phase



Status renaming and name hiding

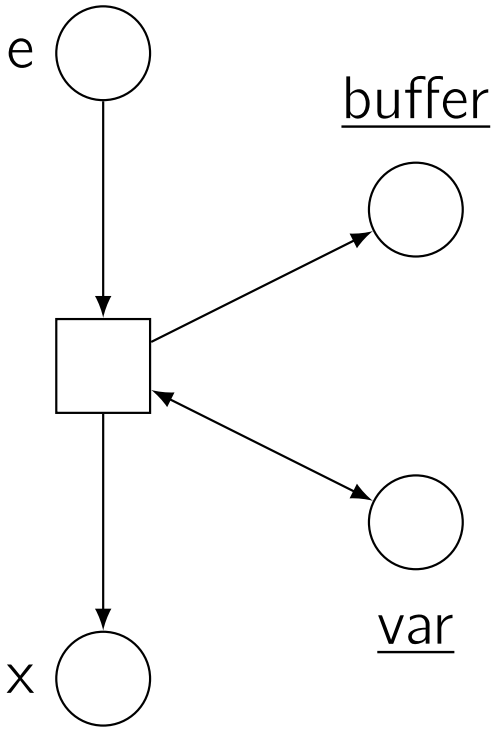
- partial function $\varrho : \mathbb{S} \setminus \{\mathsf{e}, \mathsf{i}, \mathsf{x}, \varepsilon\} \to \mathbb{S} \setminus \{\mathsf{e}, \mathsf{i}, \mathsf{x}\}$



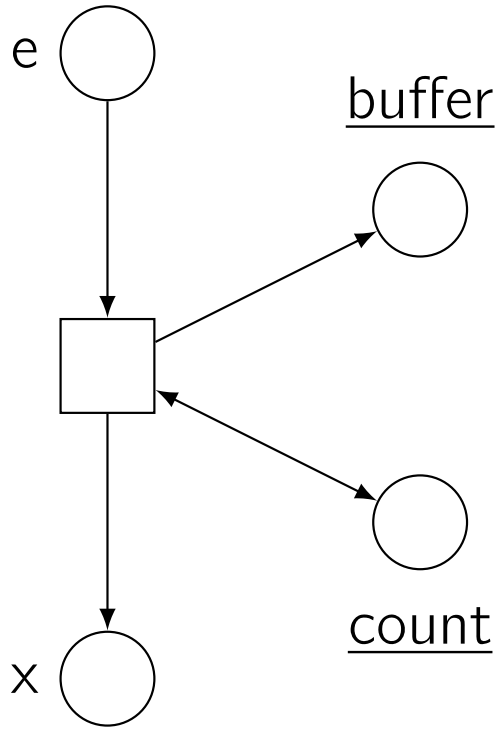
Status renaming and name hiding

- partial function $\varrho : \mathbb{S} \setminus \{\mathsf{e}, \mathsf{i}, \mathsf{x}, \varepsilon\} \to \mathbb{S} \setminus \{\mathsf{e}, \mathsf{i}, \mathsf{x}\}$
- renaming $N[\varrho]$

N



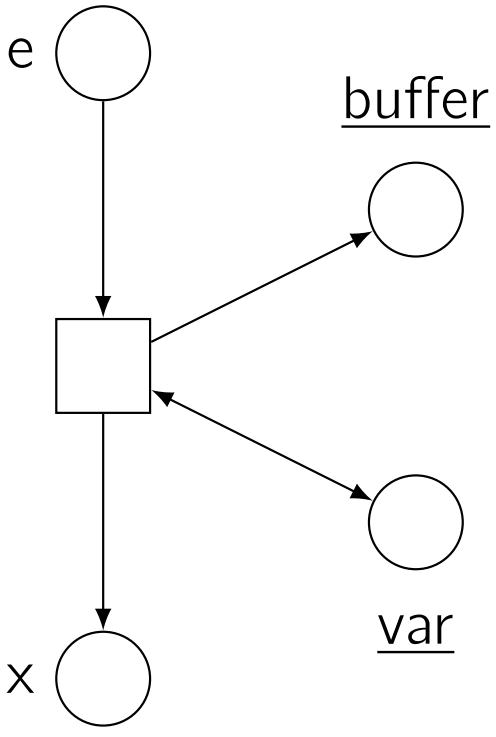
$N[\underline{\text{var}} \mapsto \underline{\text{count}}]$



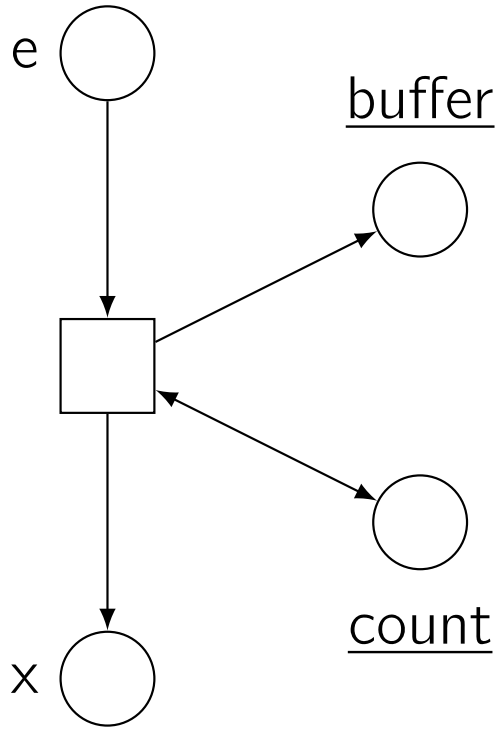
Status renaming and name hiding

- partial function $\varrho : \mathbb{S} \setminus \{\mathsf{e}, \mathsf{i}, \mathsf{x}, \varepsilon\} \to \mathbb{S} \setminus \{\mathsf{e}, \mathsf{i}, \mathsf{x}\}$
- renaming $N[\varrho]$
- name hiding $N/\varsigma \stackrel{\text{df}}{=} N[\varsigma \mapsto \varepsilon]$

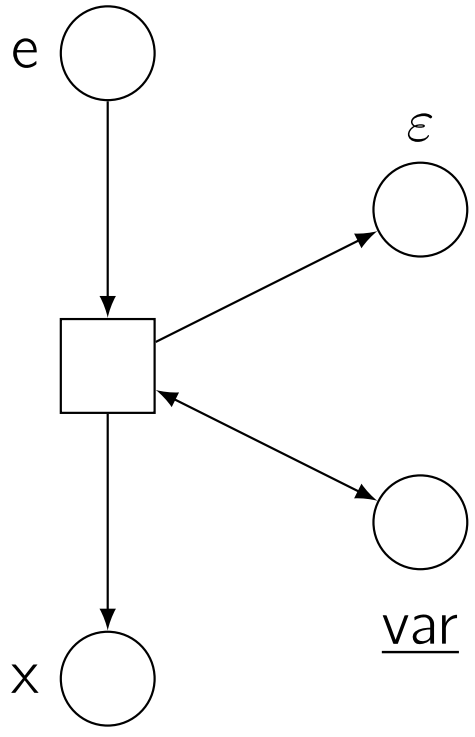
N



$N[\underline{\text{var}} \mapsto \underline{\text{count}}]$



$N/\underline{\text{buffer}}$



Outline

A modular framework
- Coloured Petri nets
- Synchronous communication
- Control flow
- Exceptions
- Threads

ABCD of modelling

More than simulation

Three application domains

Verification issues

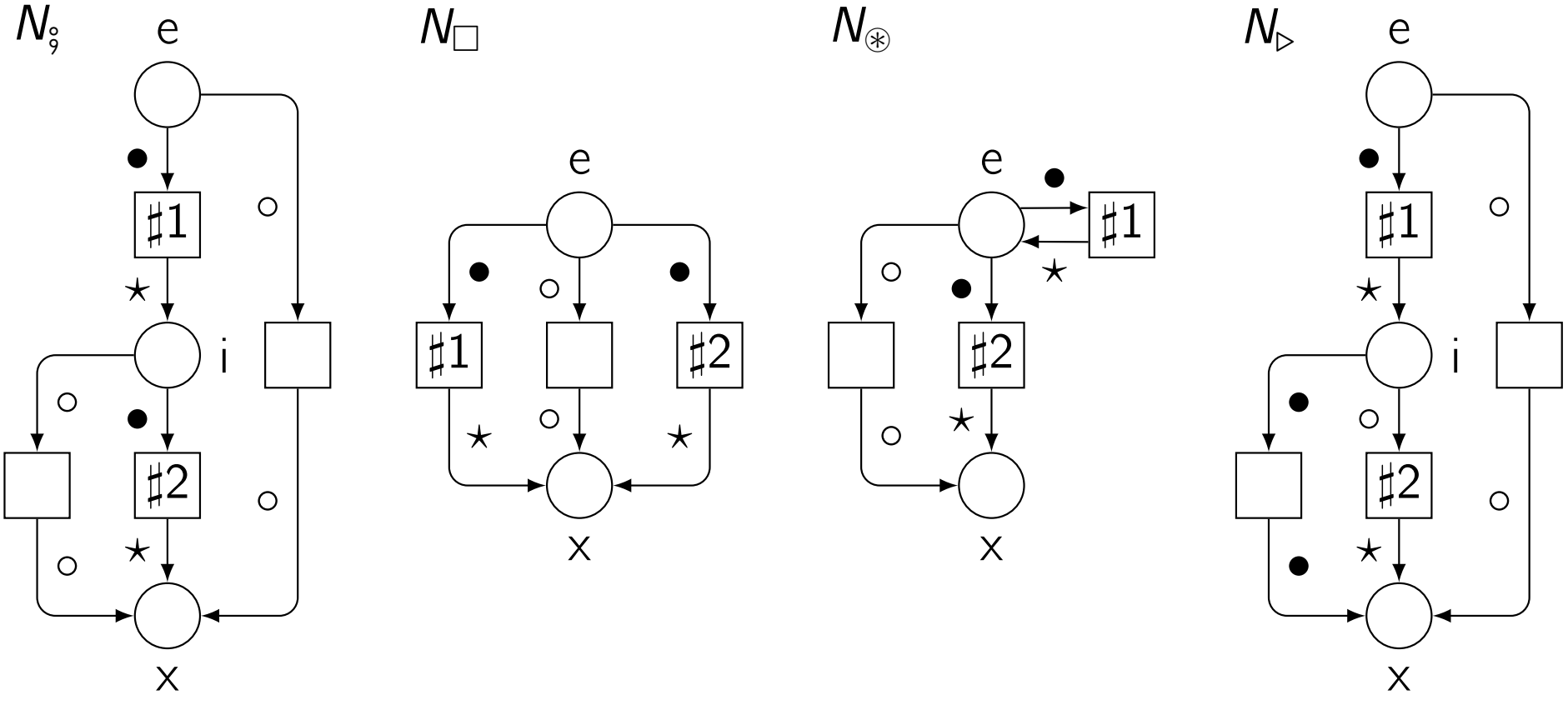
Conclusion

Sequential dual control flow

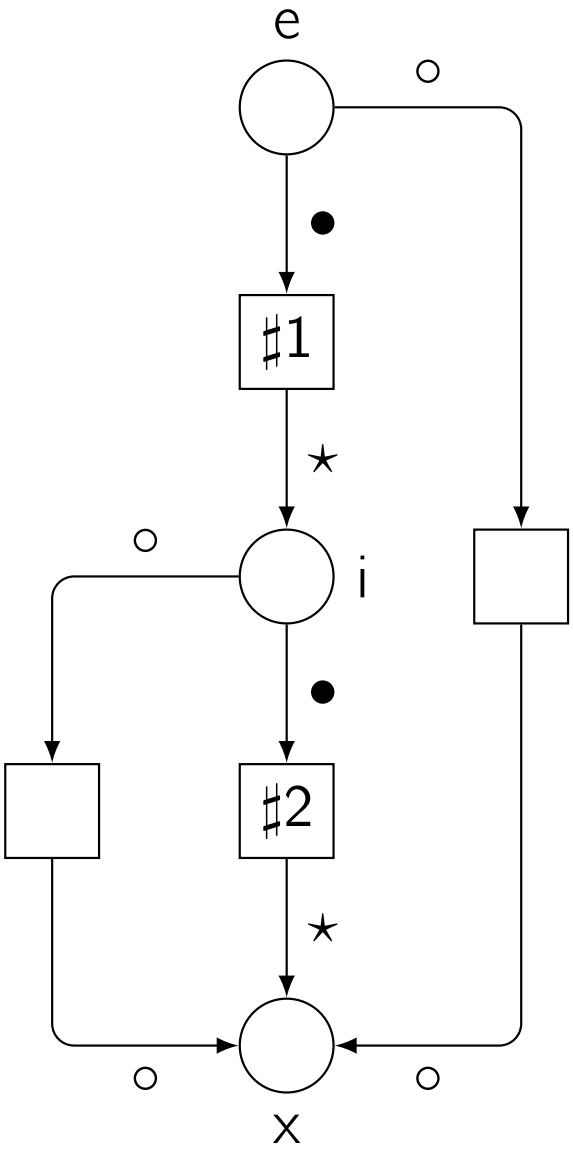
- control flow tokens: regular •, error ○
- syntactical restriction: exactly one entry and one exit place

Sequential dual control flow

- control flow tokens: regular •, error ○
- syntactical restriction: exactly one entry and one exit place
- operators: ‖ discarded, ▷ added (trap)

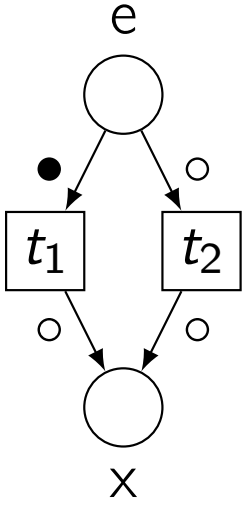


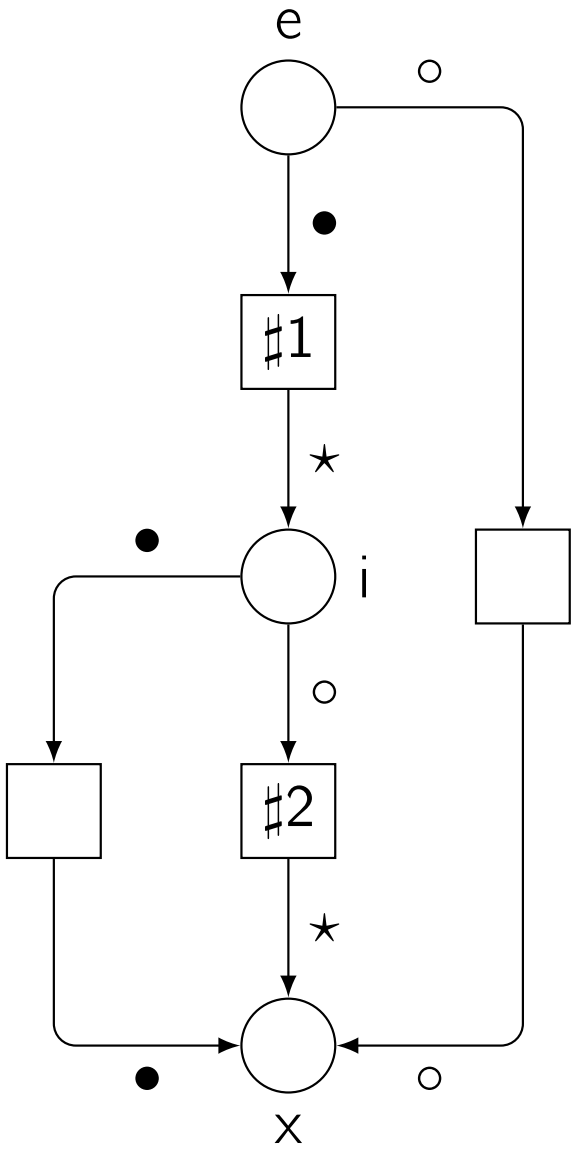
Composing nets with exceptions



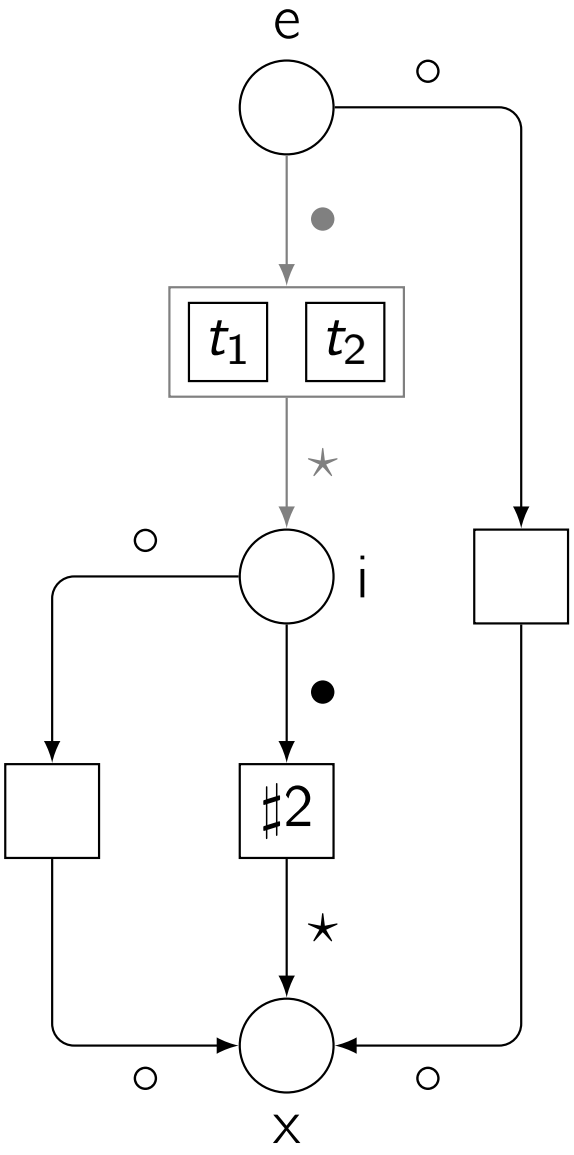
operator

operand	●	○	⋆	∅
●	●	○	●	∅
○	⋆	⋆	○	∅
∅	∅	∅	∅	∅
m	⋆	⋆	m	∅



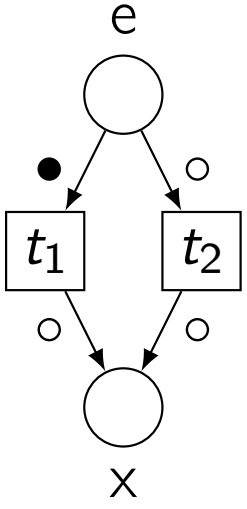


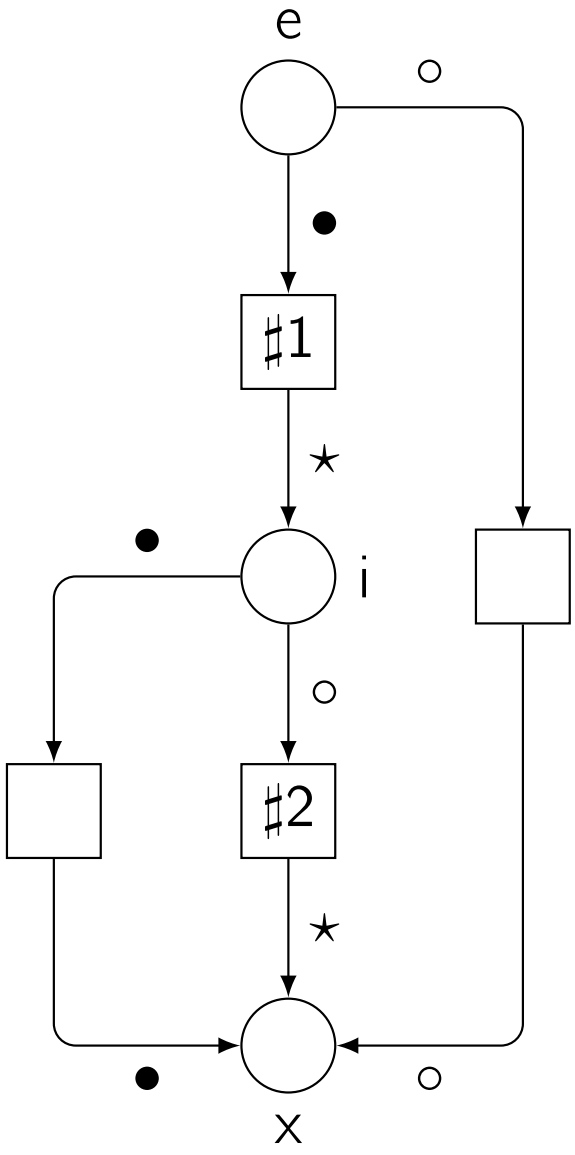
Composing nets with exceptions



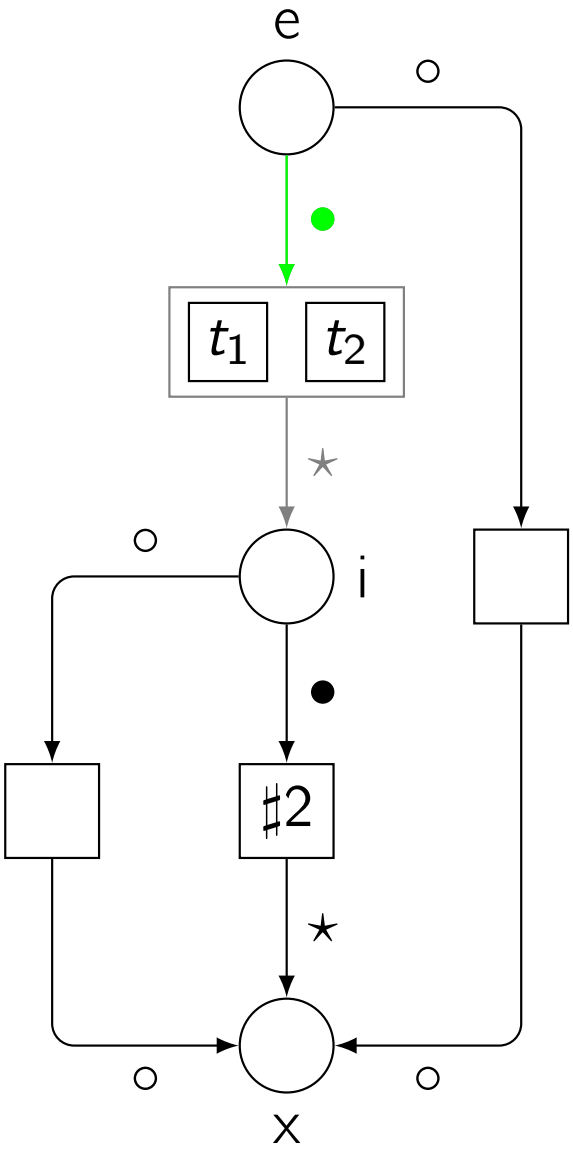
operator

operand	●	○	⋆	∅
●	●	○	●	∅
○	⋆	⋆	○	∅
∅	∅	∅	∅	∅
m	⋆	⋆	m	∅



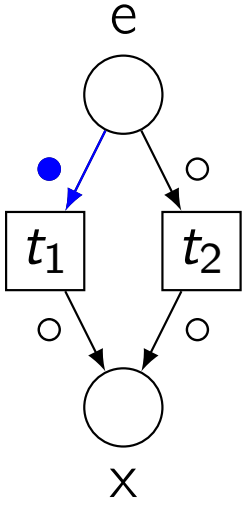


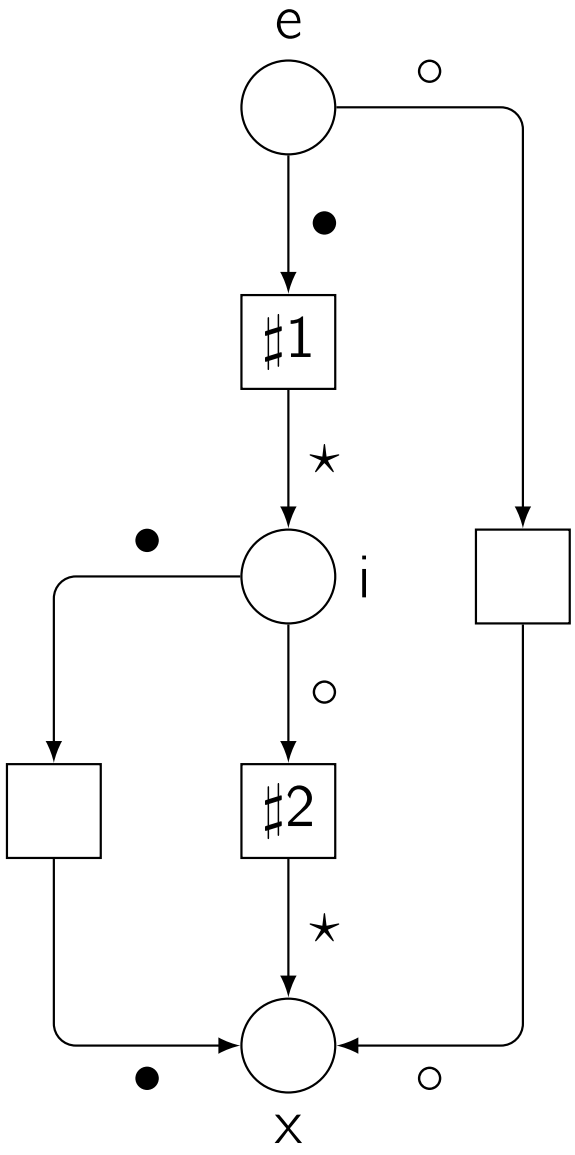
Composing nets with exceptions



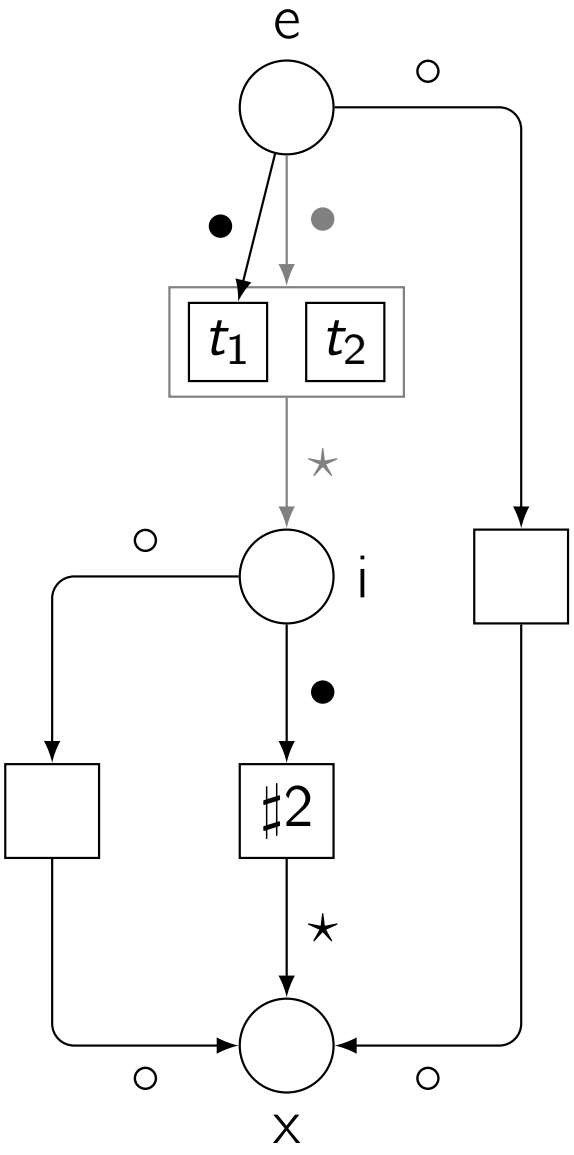
operator

operand	●	○	⋆	∅
●	●	○	●	∅
○	⋆	⋆	○	∅
∅	∅	∅	∅	∅
m	⋆	⋆	m	∅



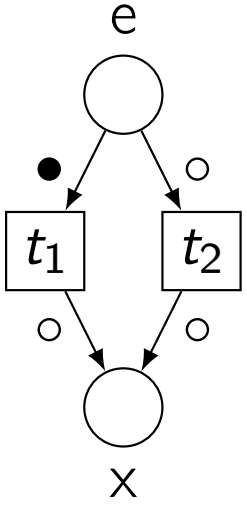


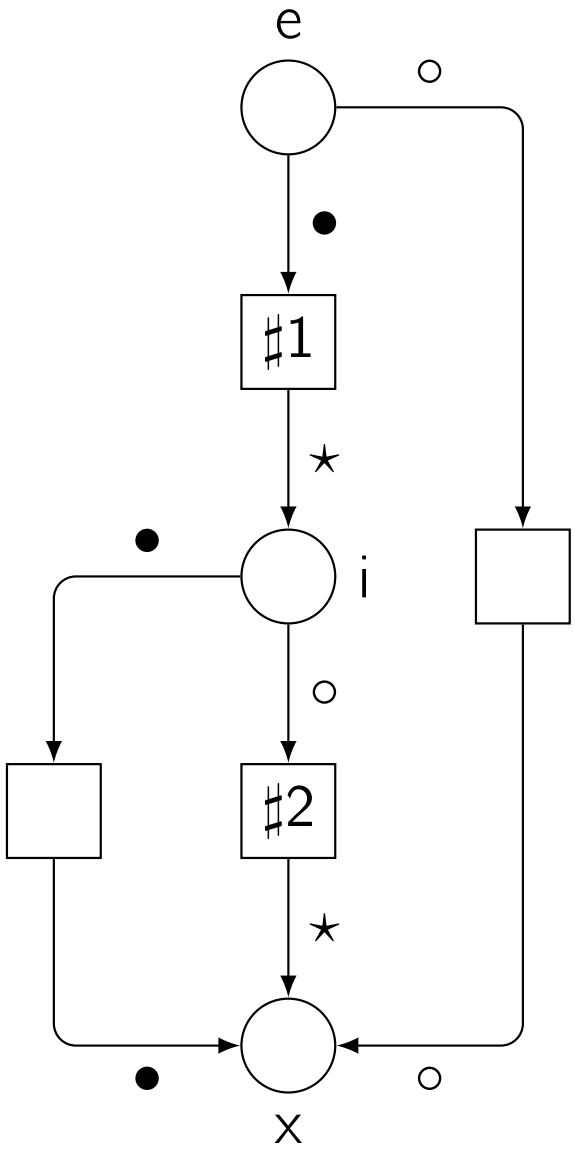
Composing nets with exceptions



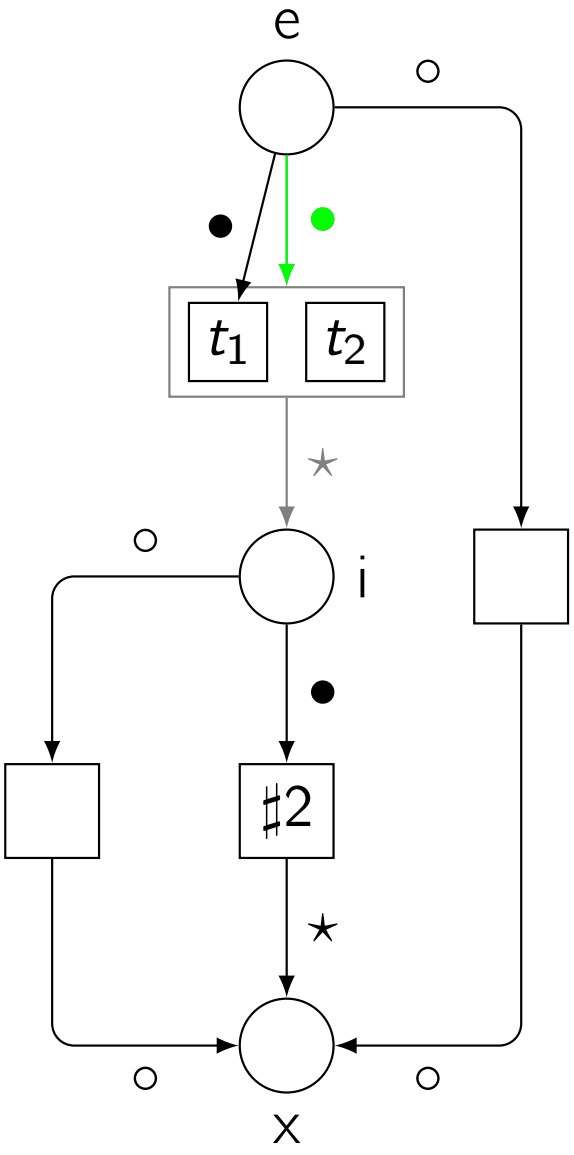
operator

operand	●	○	⋆	∅
●	●	○	●	∅
○	⋆	⋆	○	∅
∅	∅	∅	∅	∅
m	⋆	⋆	m	∅



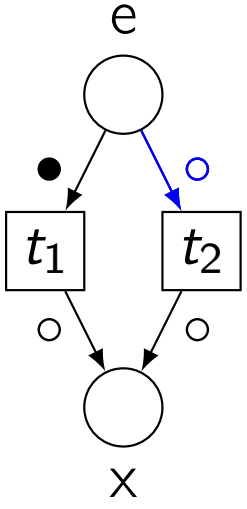


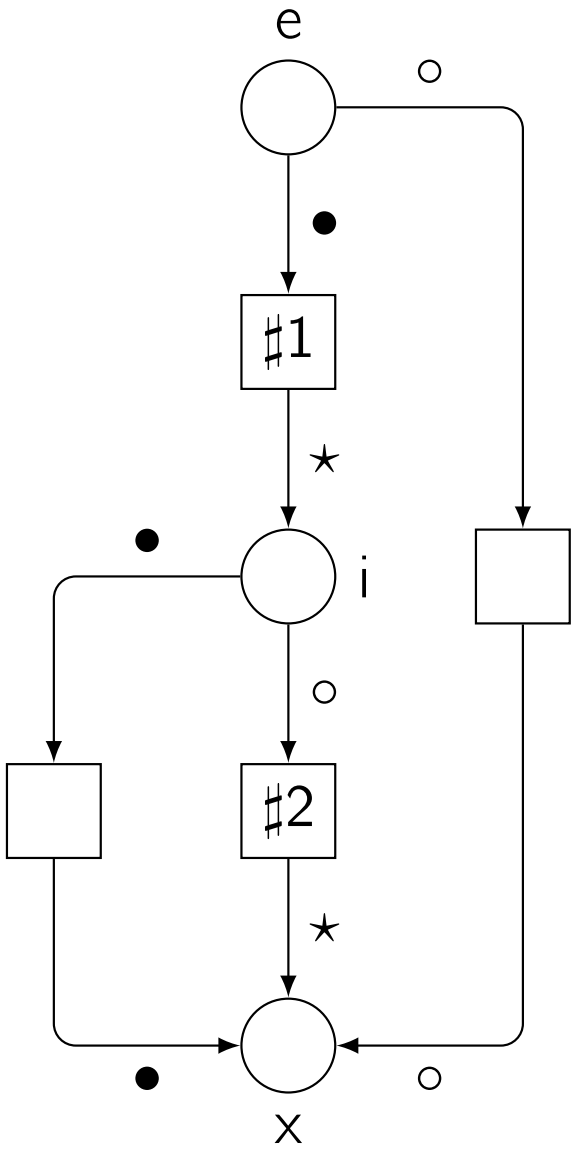
Composing nets with exceptions



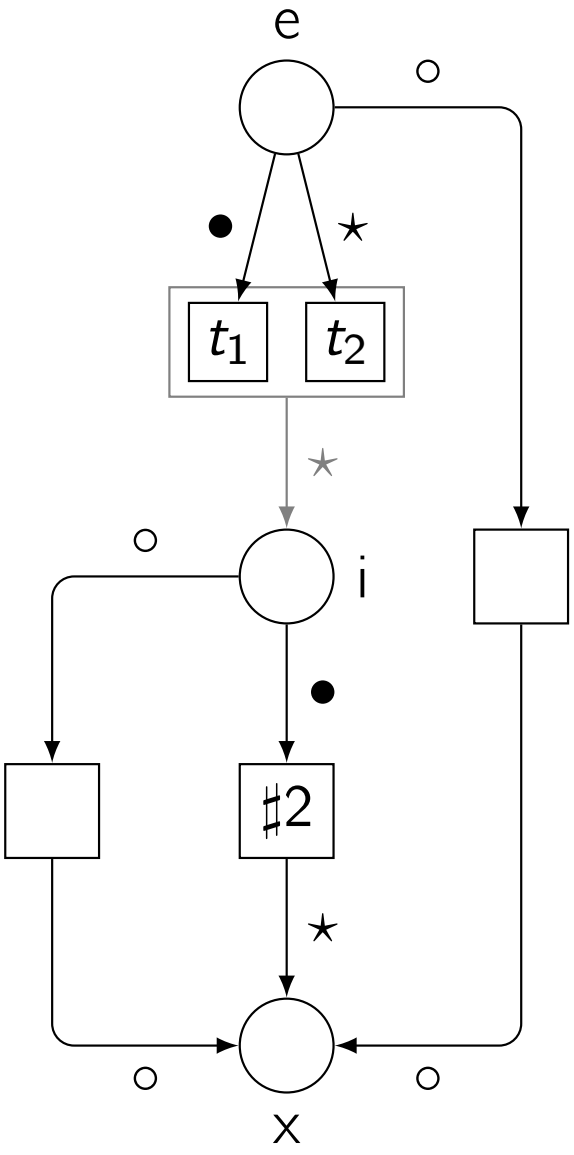
operator

operand	●	○	⋆	∅
●	●	○	●	∅
○	⋆	⋆	○	∅
∅	∅	∅	∅	∅
m	⋆	⋆	m	∅



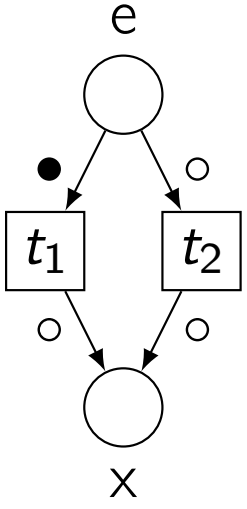


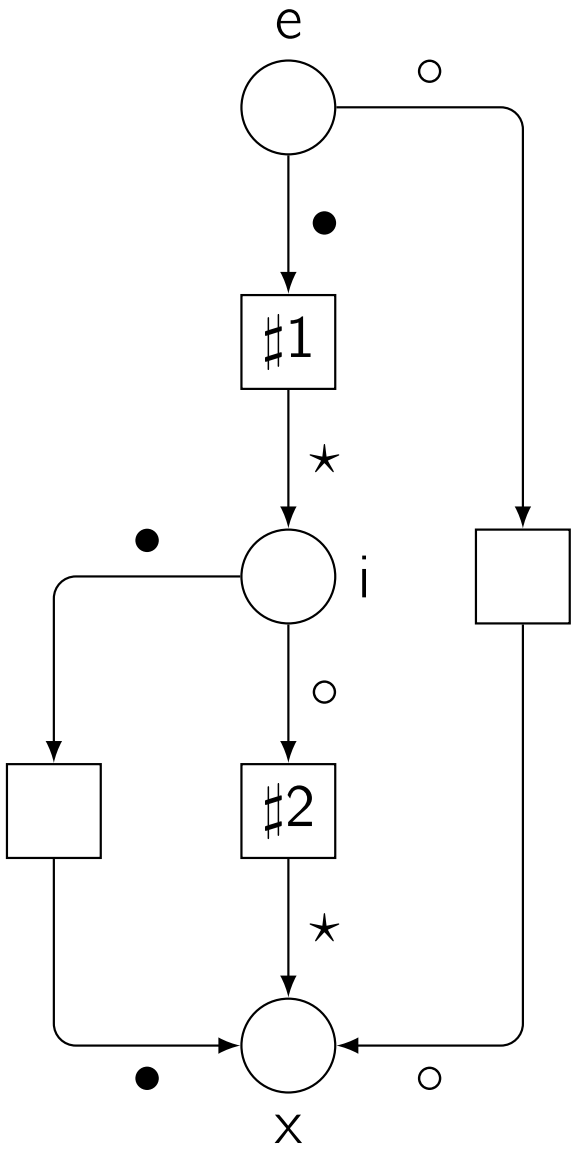
Composing nets with exceptions



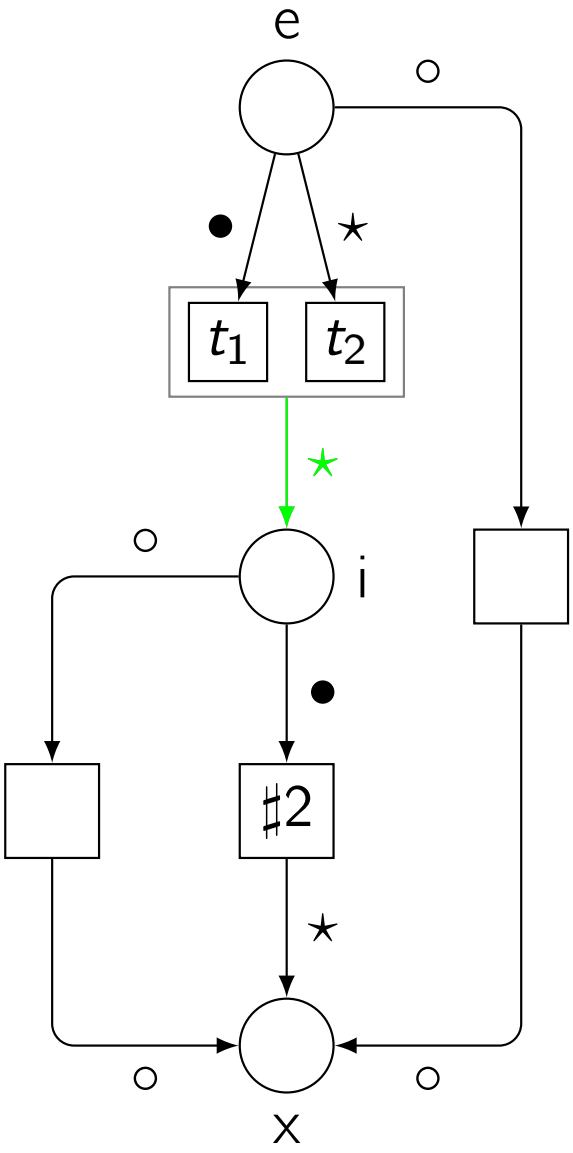
operator

operand	●	○	⋆	∅
●	●	○	●	∅
○	⋆	⋆	○	∅
∅	∅	∅	∅	∅
m	⋆	⋆	m	∅



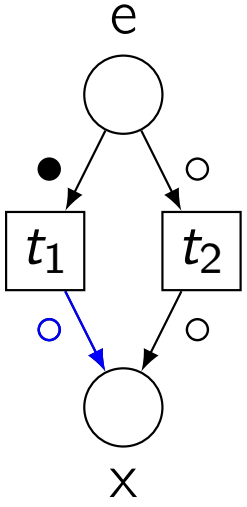


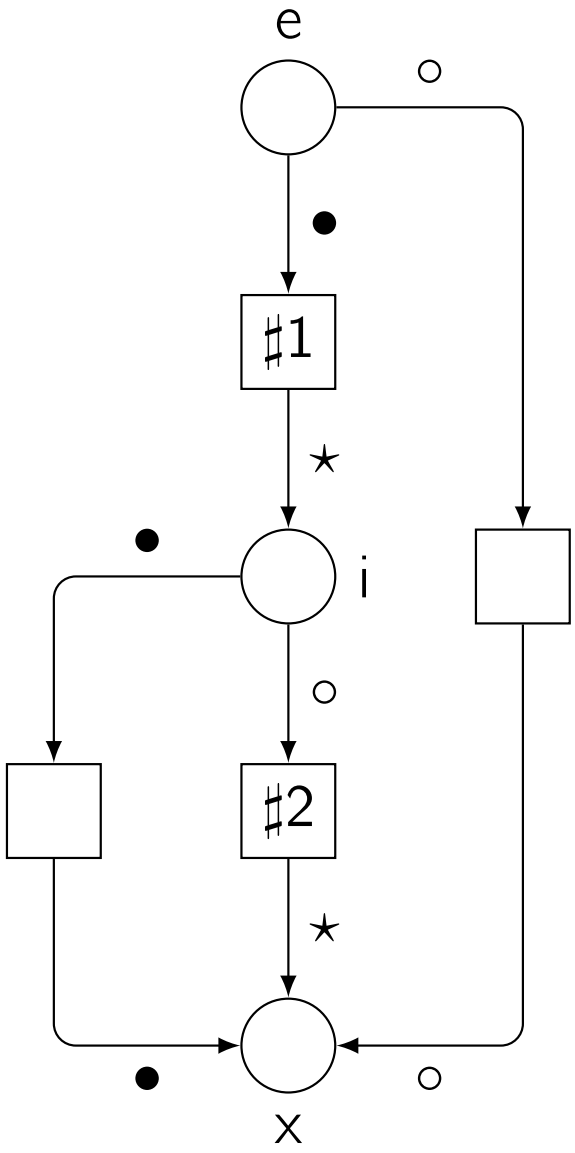
Composing nets with exceptions



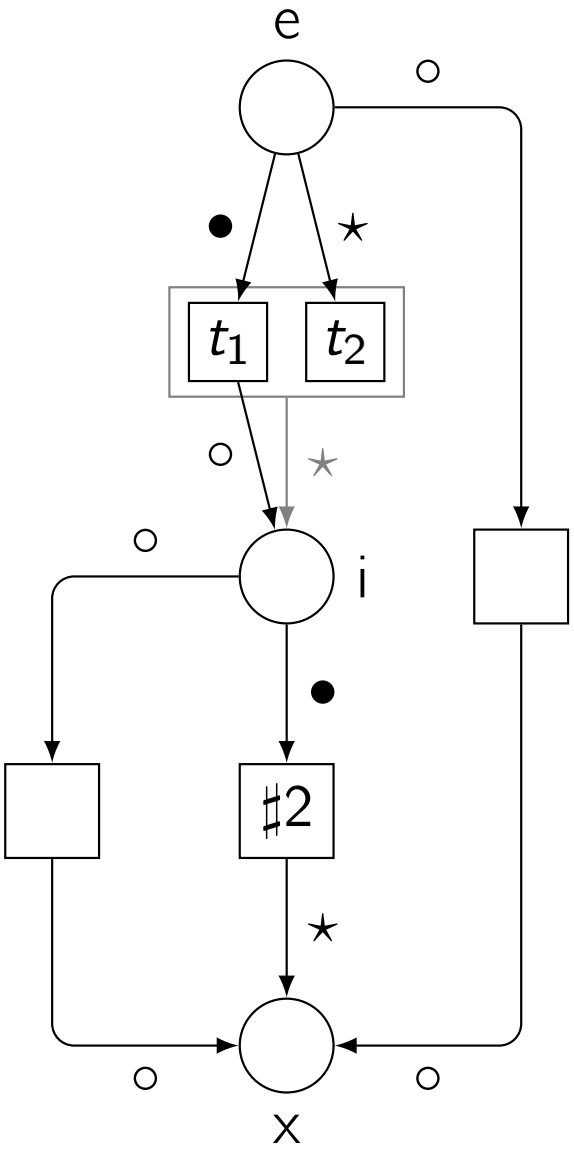
operator

operand	●	○	⋆	∅
●	●	○	●	∅
○	⋆	⋆	○	∅
∅	∅	∅	∅	∅
m	⋆	⋆	m	∅



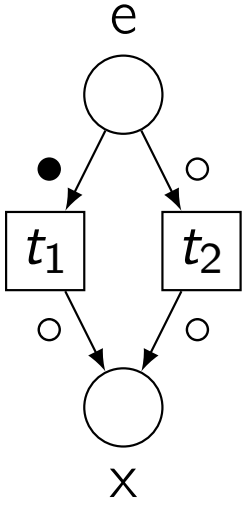


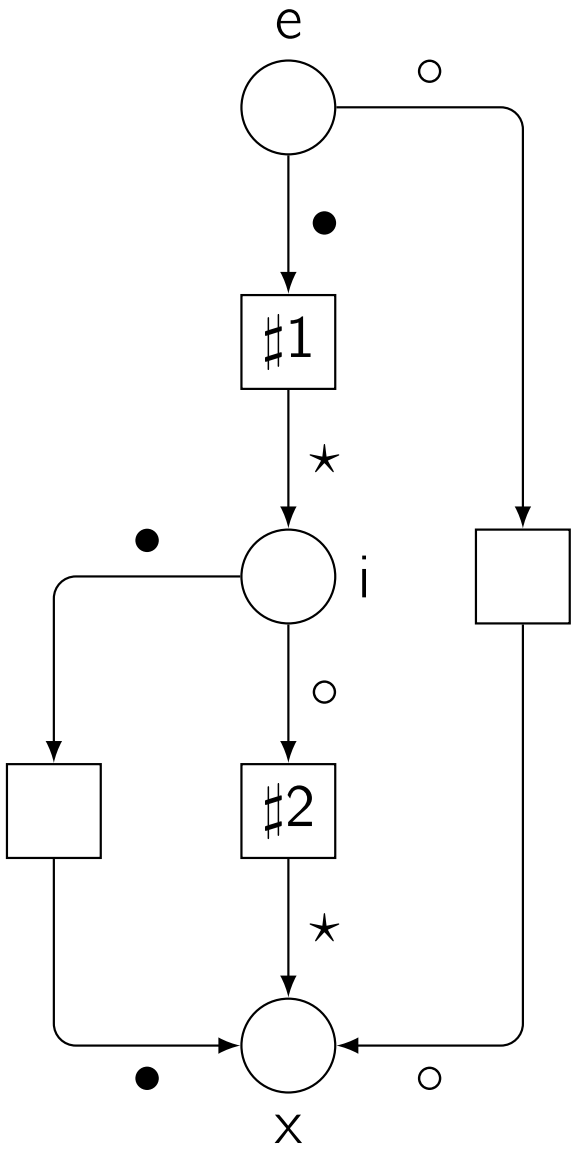
Composing nets with exceptions



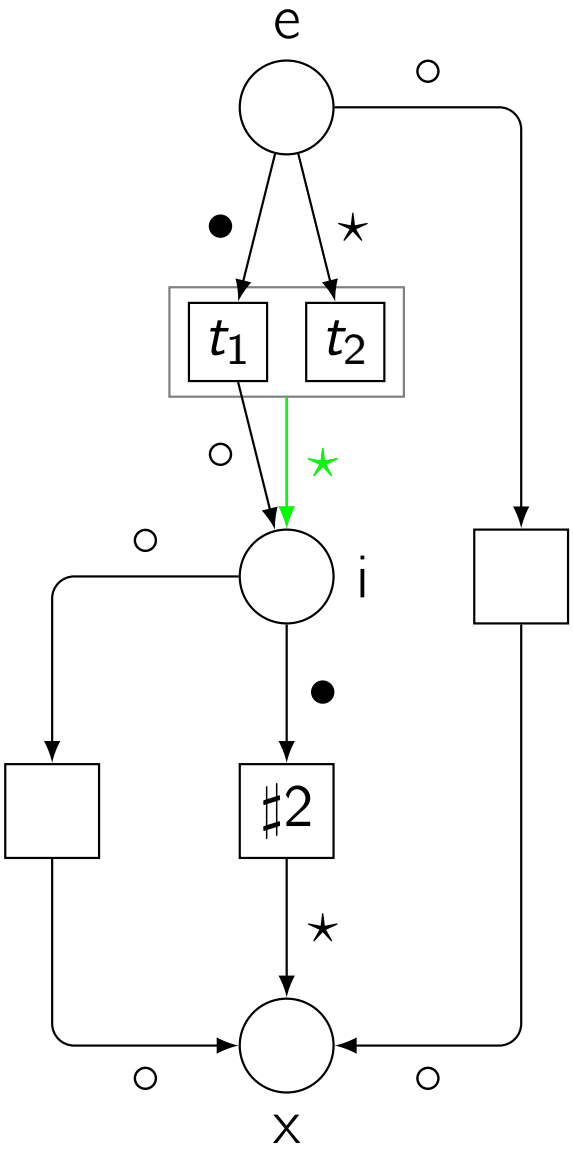
operator

operand	●	○	⋆	∅
●	●	○	●	∅
○	⋆	⋆	○	∅
∅	∅	∅	∅	∅
m	⋆	⋆	m	∅



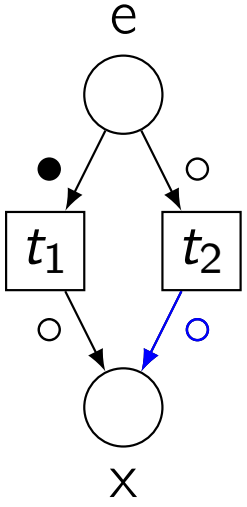


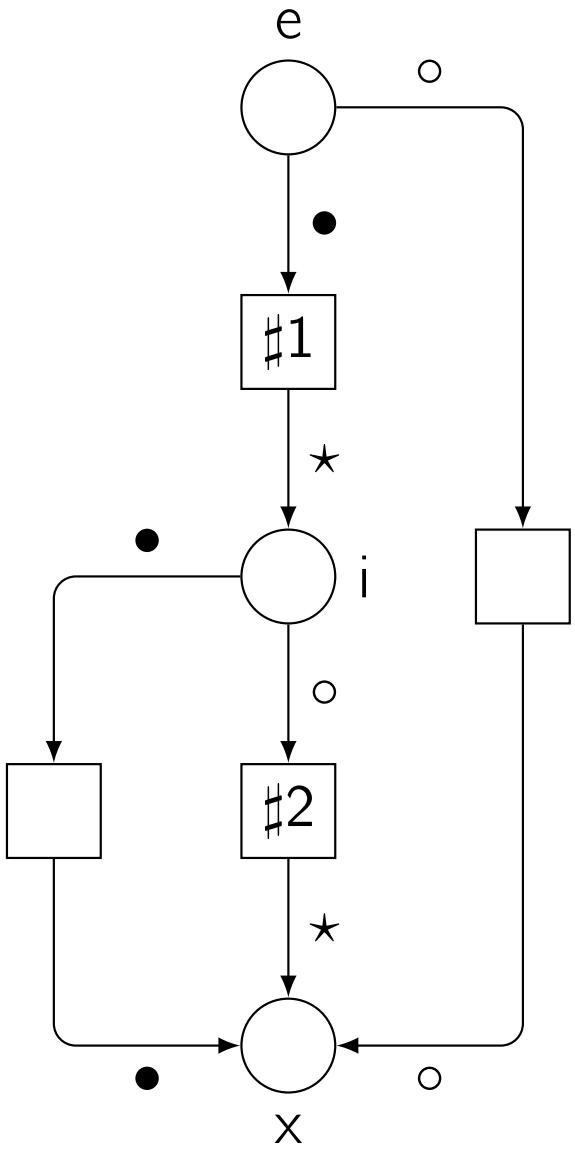
Composing nets with exceptions



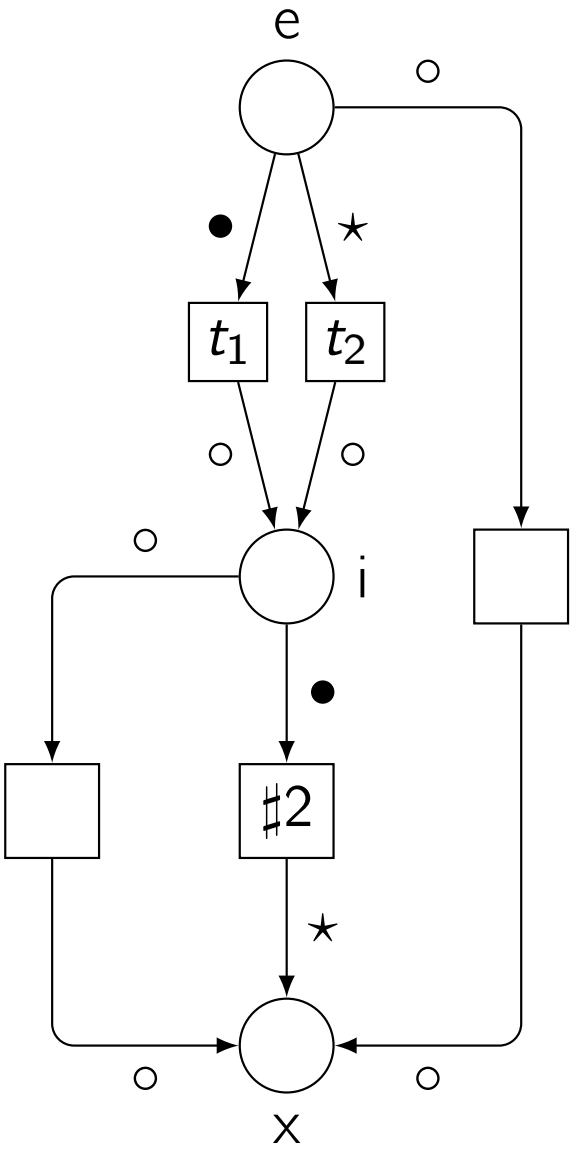
operator

operand	●	○	⋆	∅
●	●	○	●	∅
○	⋆	⋆	○	∅
∅	∅	∅	∅	∅
m	⋆	⋆	m	∅



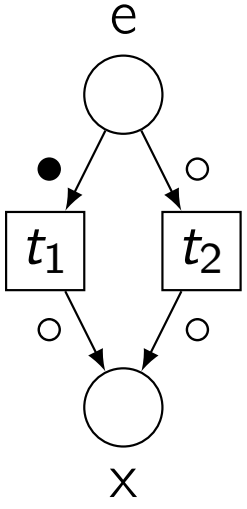


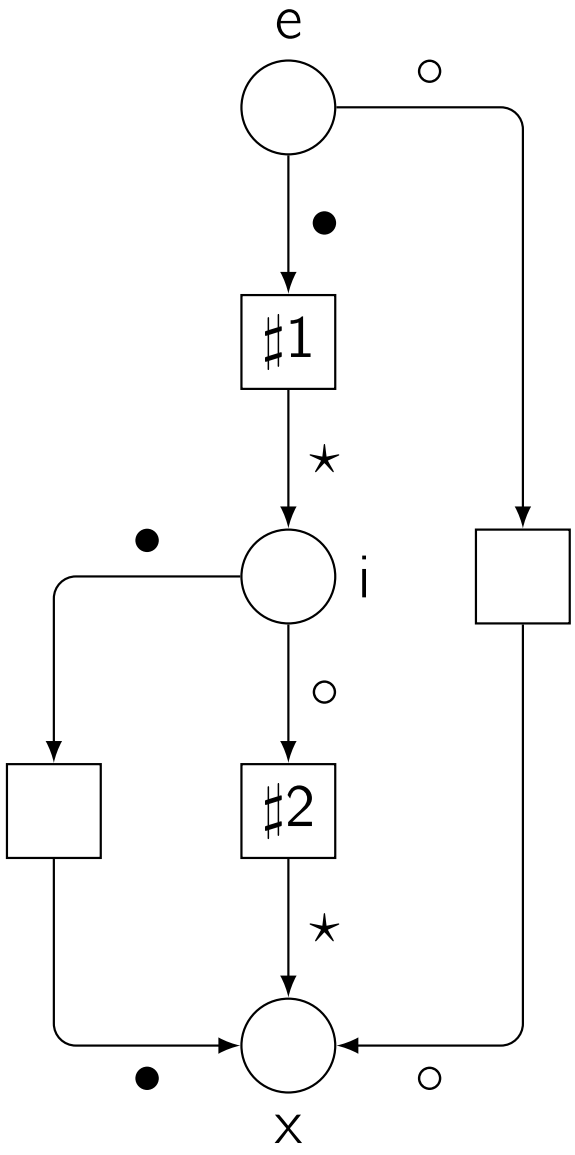
Composing nets with exceptions



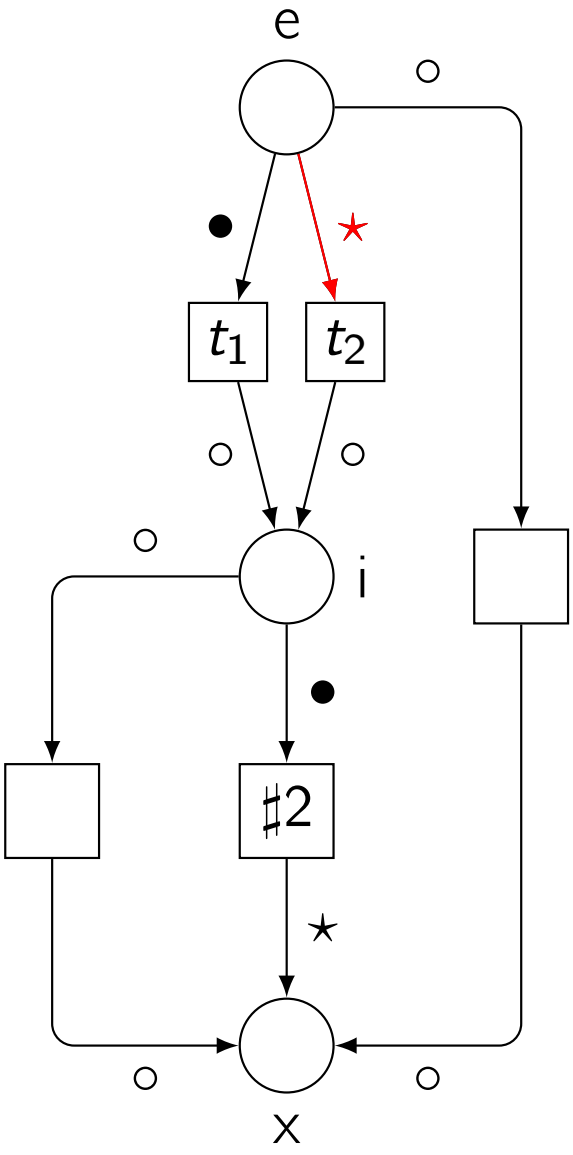
operator

operand	●	○	⋆	∅
●	●	○	●	∅
○	⋆	⋆	○	∅
∅	∅	∅	∅	∅
m	⋆	⋆	m	∅



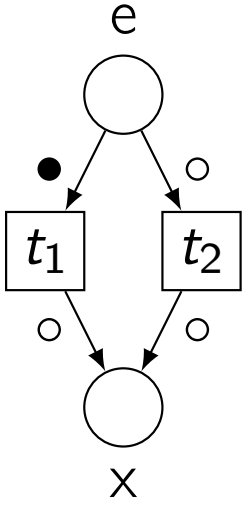


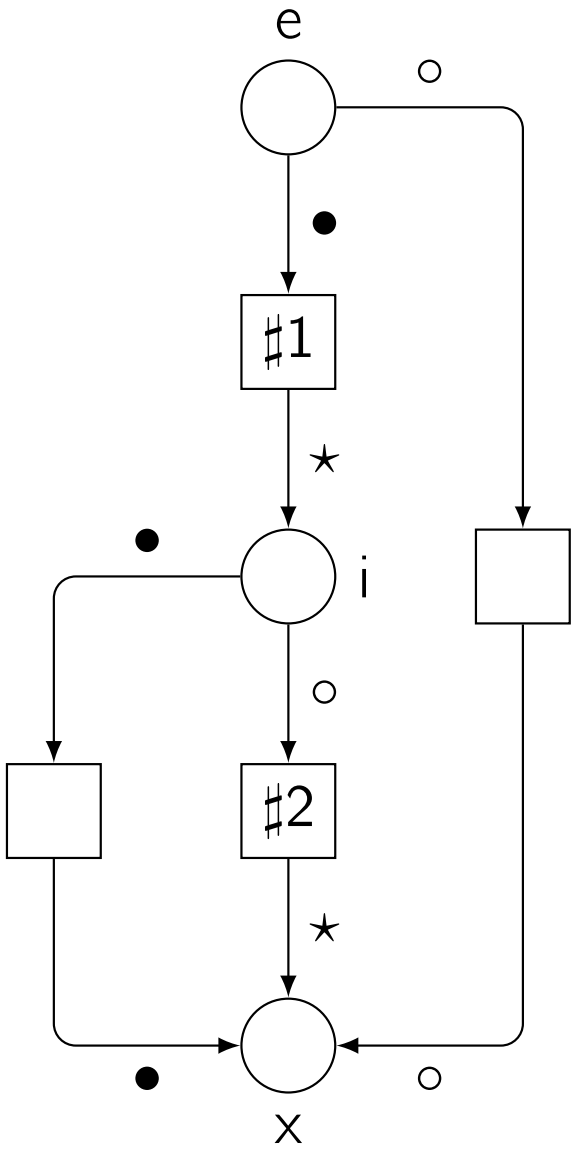
Composing nets with exceptions



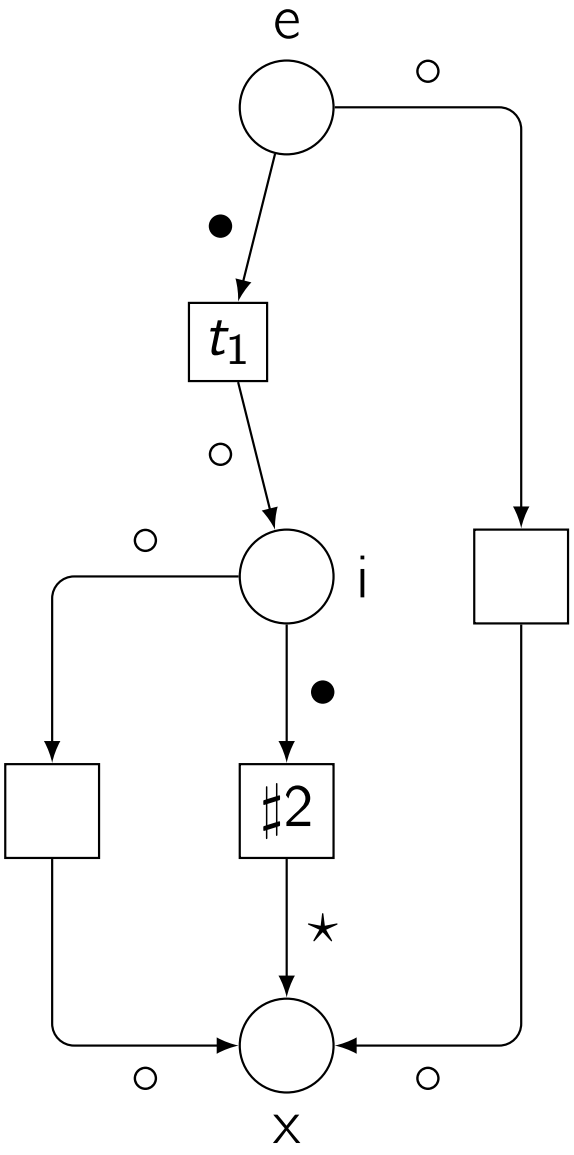
operator

operand	●	○	⋆	∅
●	●	○	●	∅
○	⋆	⋆	○	∅
∅	∅	∅	∅	∅
m	⋆	⋆	m	∅



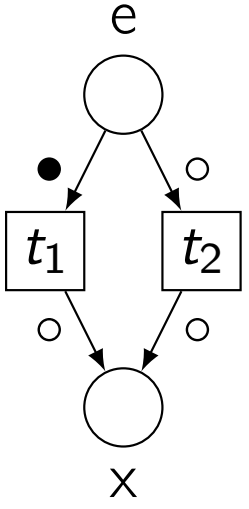


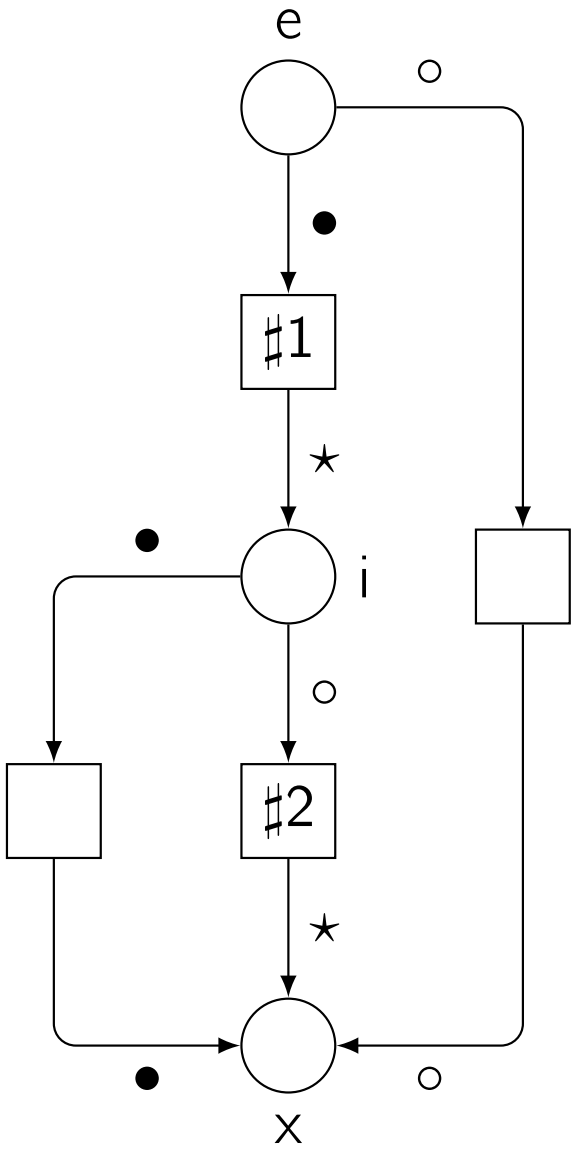
Composing nets with exceptions



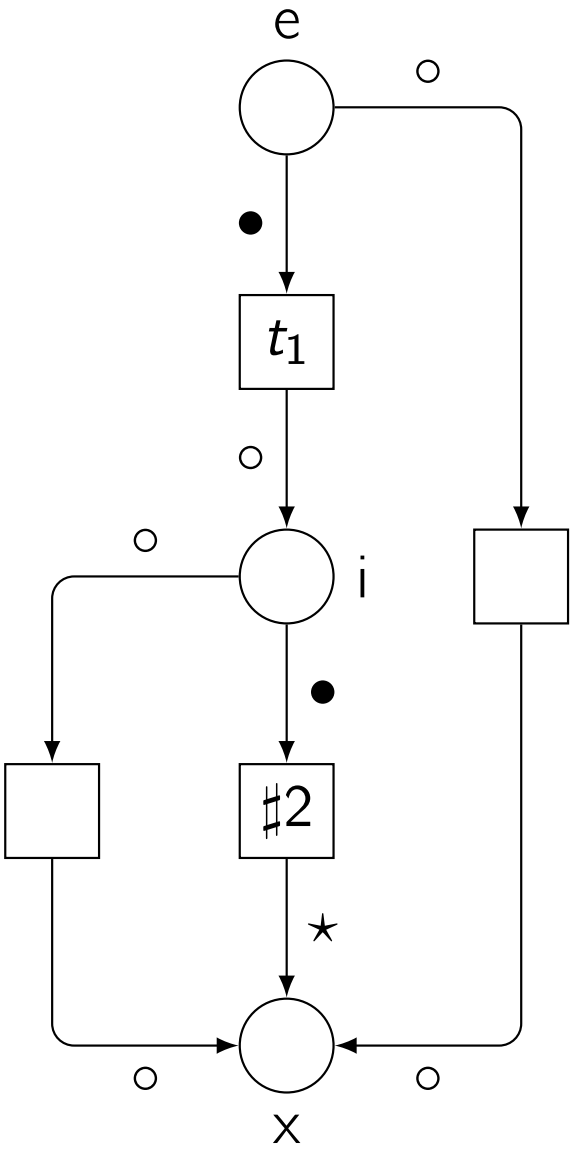
operator

operand	●	○	⋆	∅
●	●	○	●	∅
○	⋆	⋆	○	∅
∅	∅	∅	∅	∅
m	⋆	⋆	m	∅



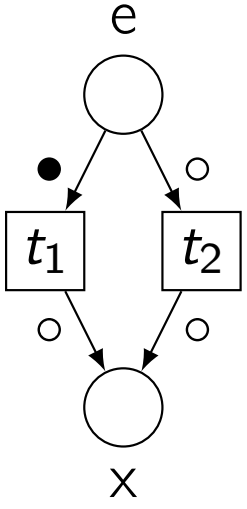


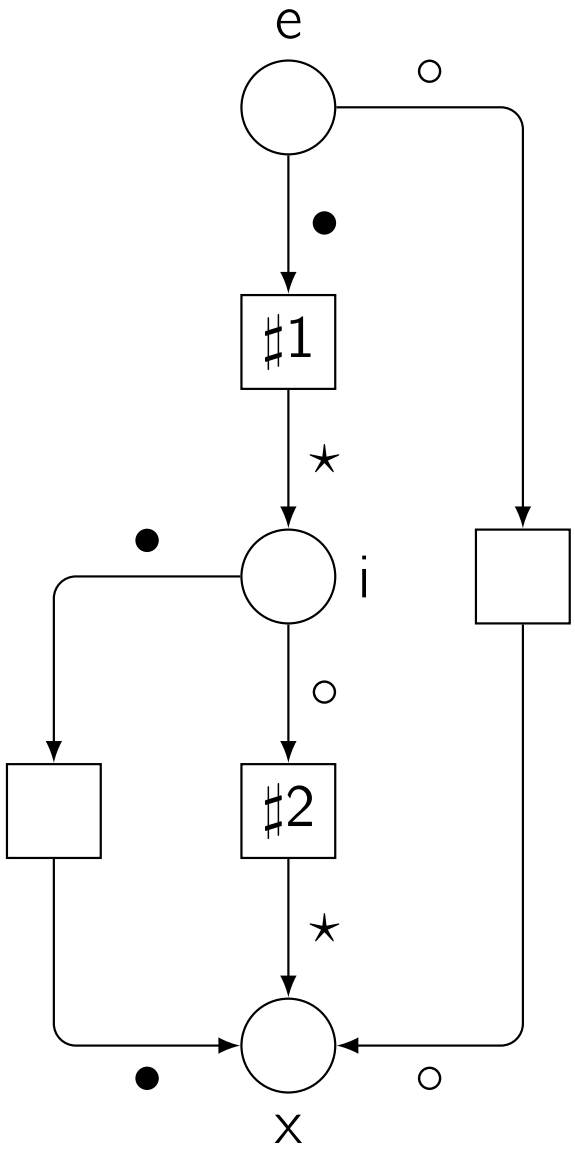
Composing nets with exceptions



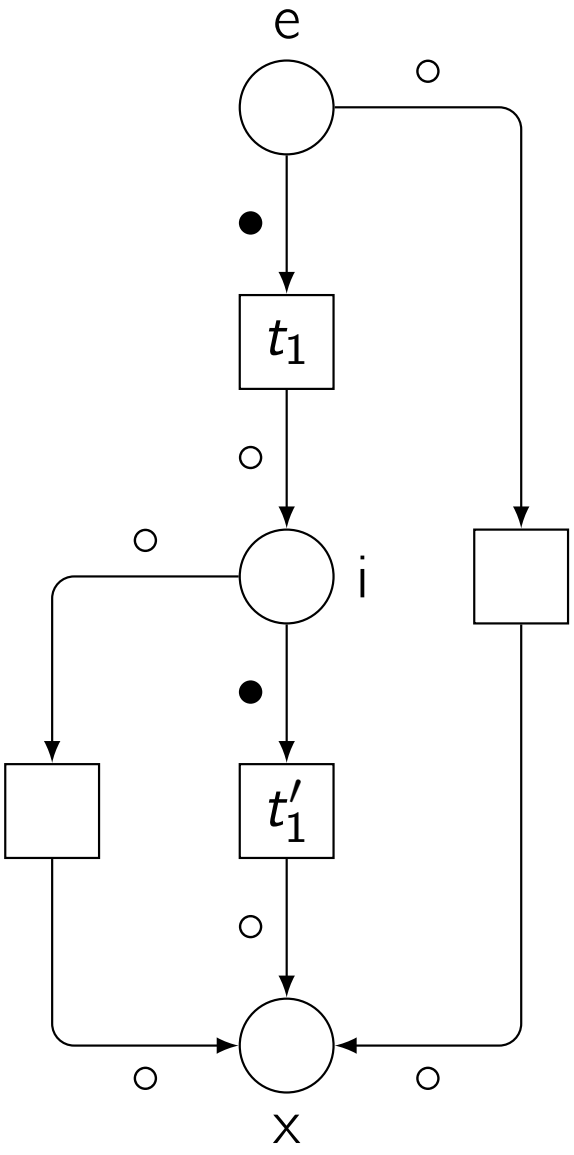
operator

operand	●	○	⋆	∅
●	●	○	●	∅
○	⋆	⋆	○	∅
∅	∅	∅	∅	∅
m	⋆	⋆	m	∅



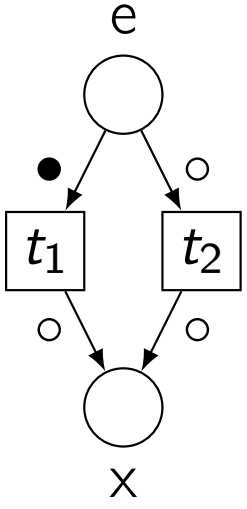


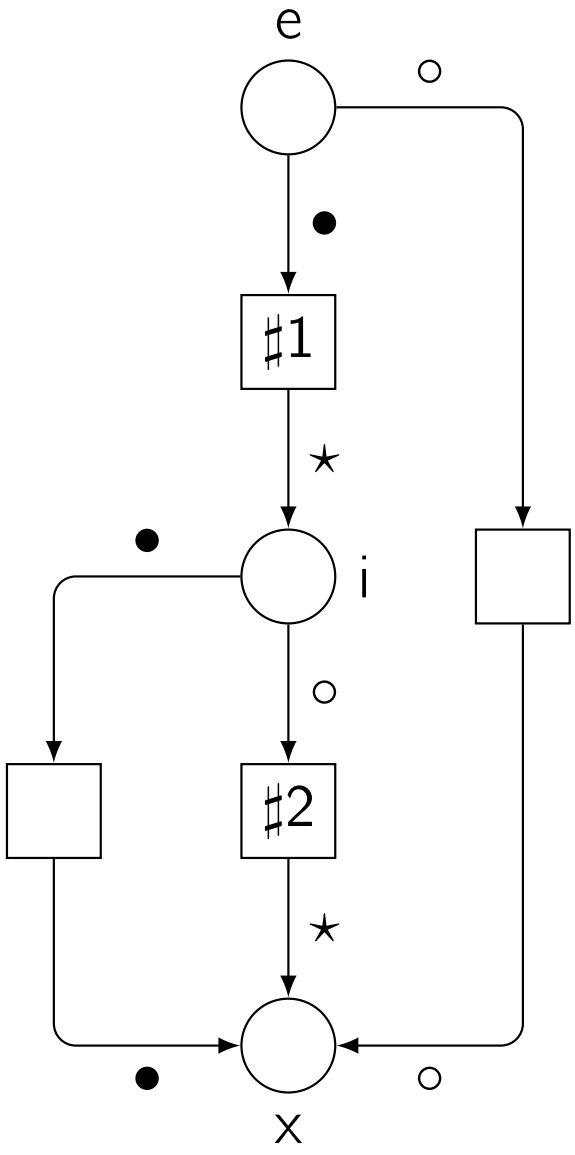
Composing nets with exceptions



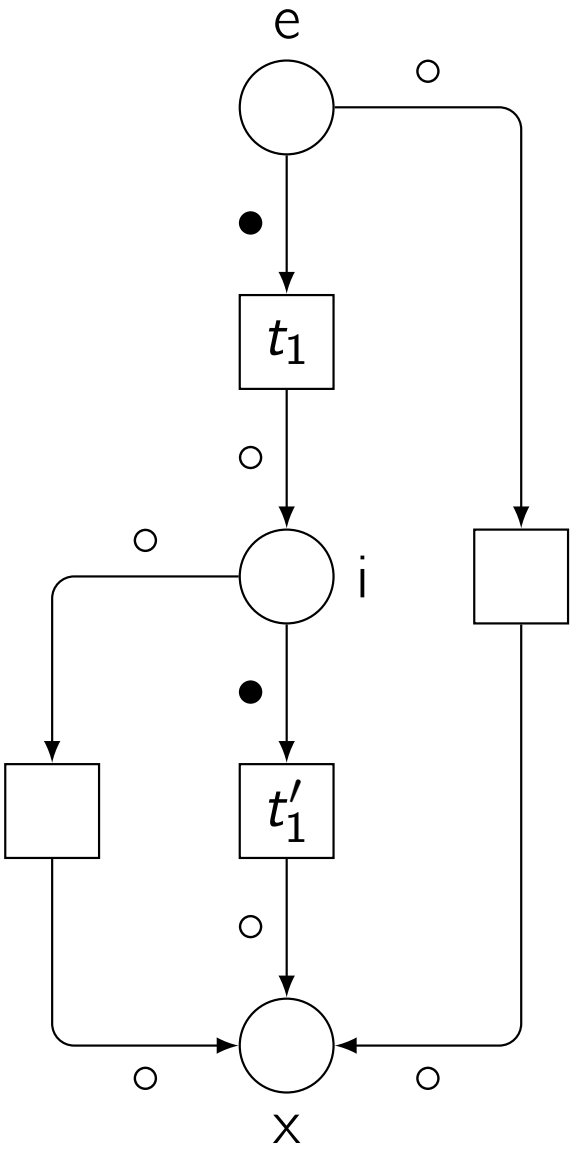
operator

operand	●	○	⋆	∅
●	●	○	●	∅
○	⋆	⋆	○	∅
∅	∅	∅	∅	∅
m	⋆	⋆	m	∅



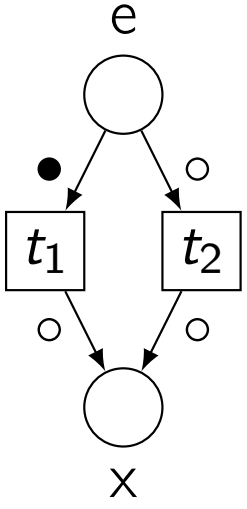


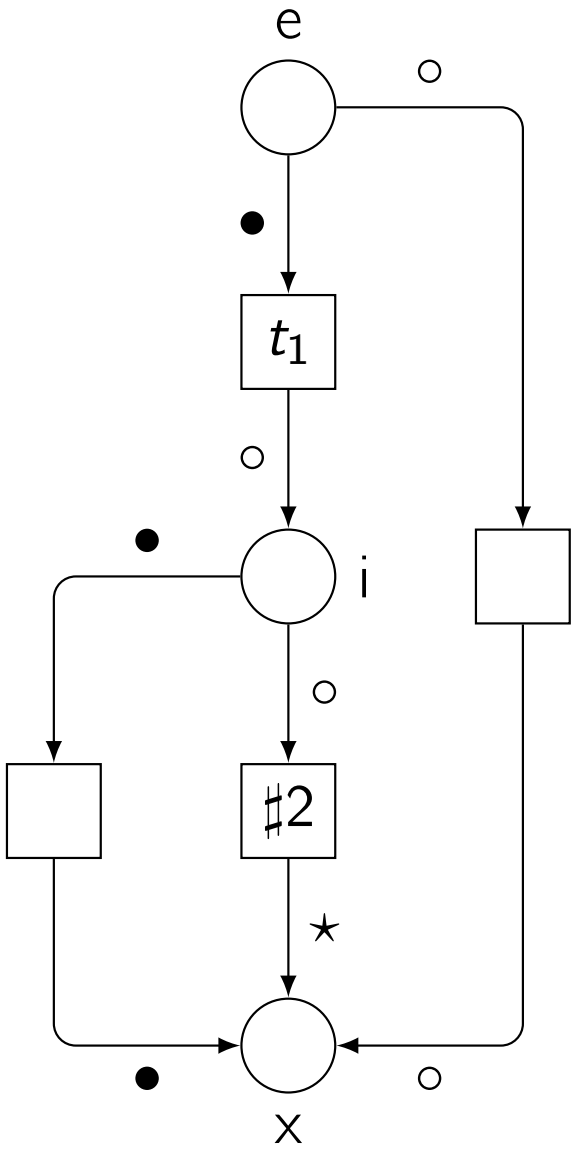
Composing nets with exceptions



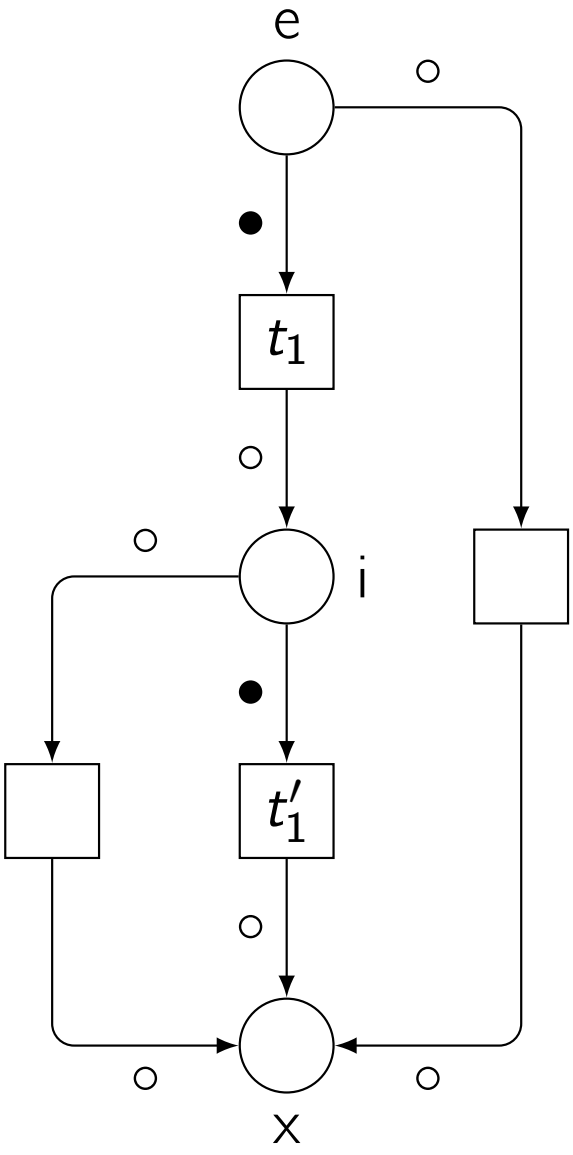
operator

operand	●	○	⋆	∅
●	●	○	●	∅
○	⋆	⋆	○	∅
∅	∅	∅	∅	∅
m	⋆	⋆	m	∅



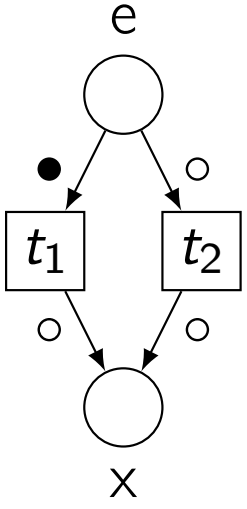


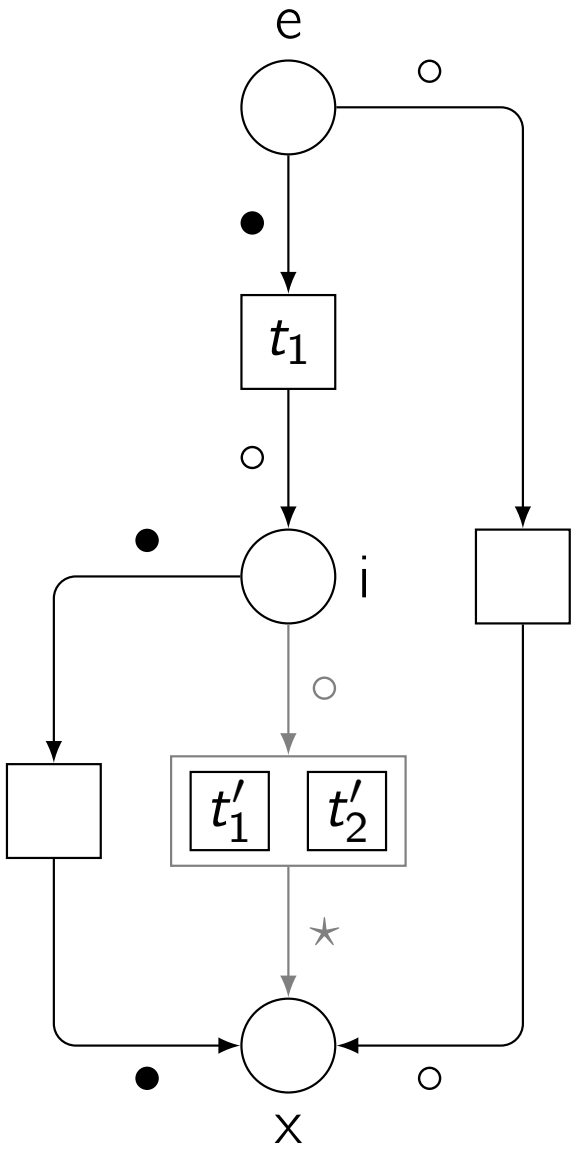
Composing nets with exceptions



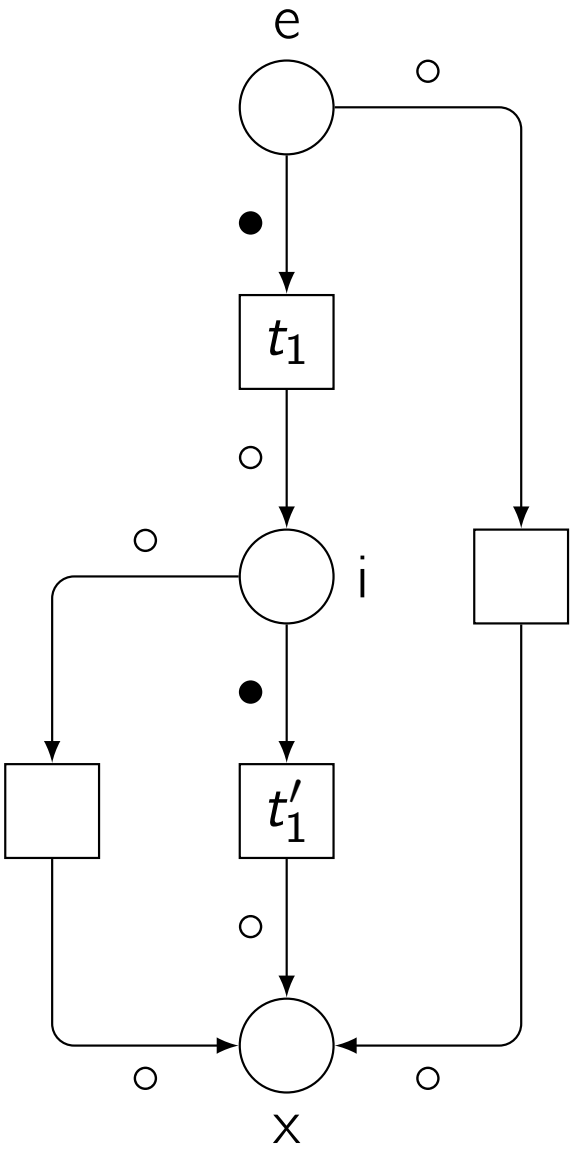
operator

operand	●	○	⋆	∅
●	●	○	●	∅
○	⋆	⋆	○	∅
∅	∅	∅	∅	∅
m	⋆	⋆	m	∅



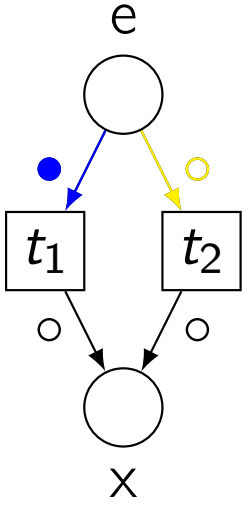


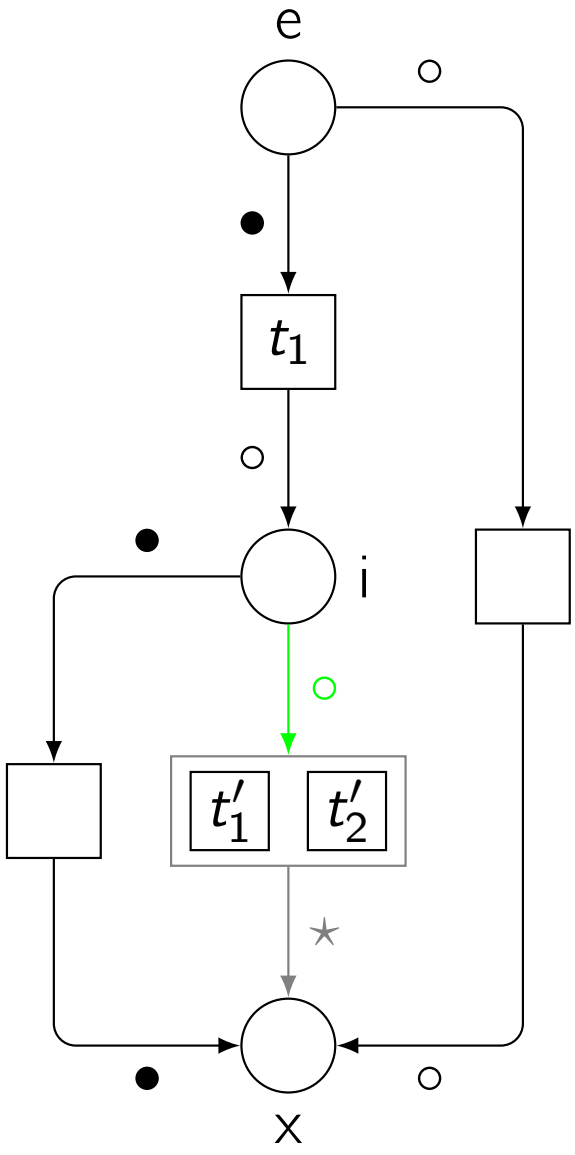
Composing nets with exceptions



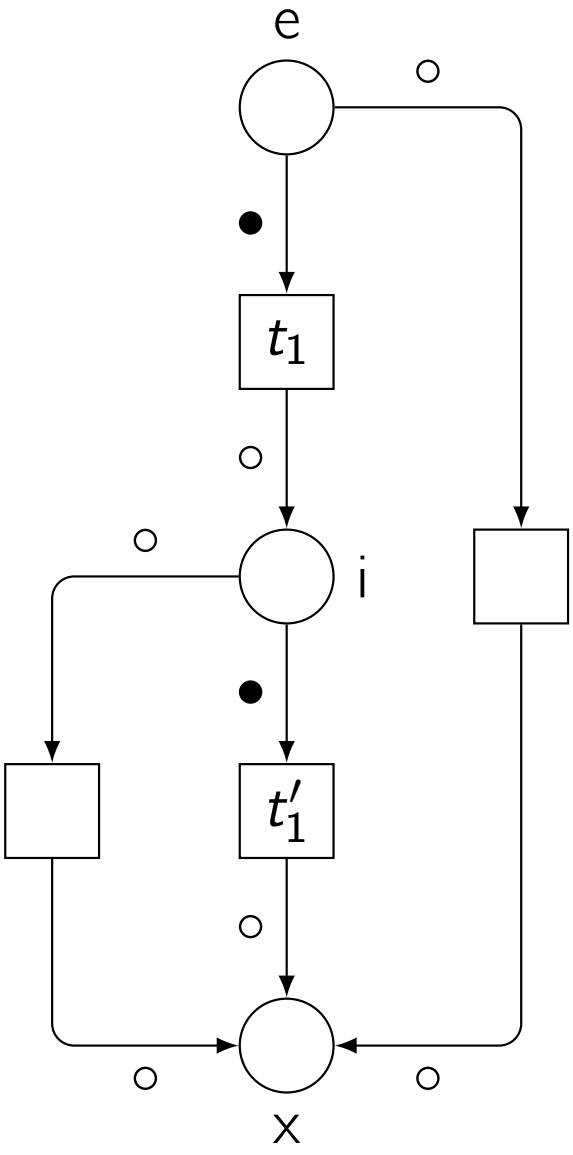
operator

operand	•	○	⋆	∅
•	•	○	•	∅
○	⋆	⋆	○	∅
∅	∅	∅	∅	∅
m	⋆	⋆	m	∅



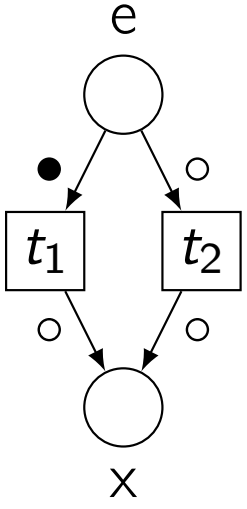


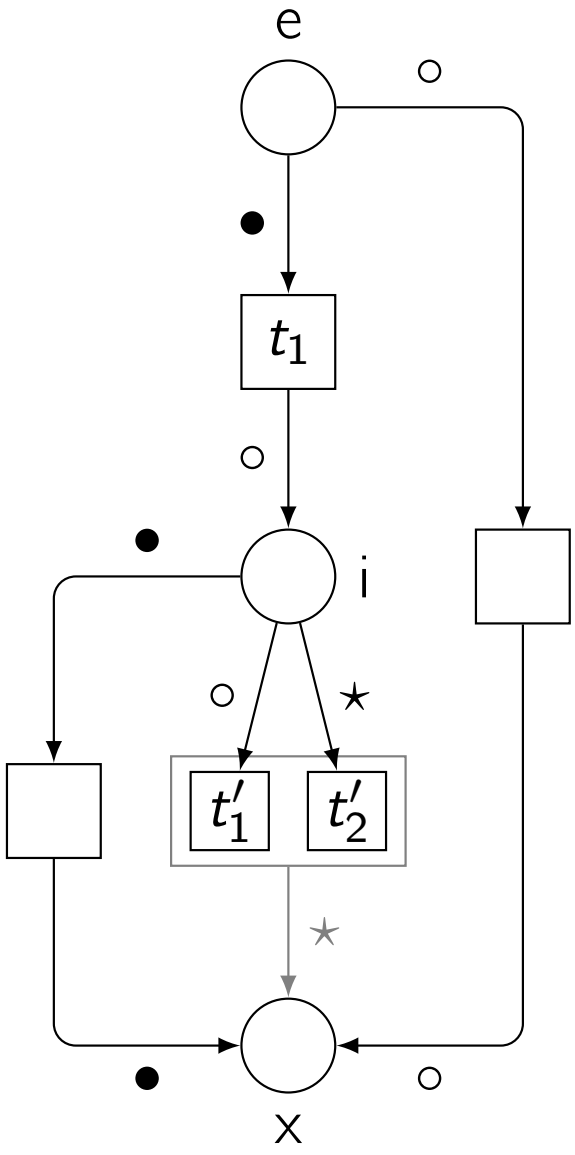
Composing nets with exceptions



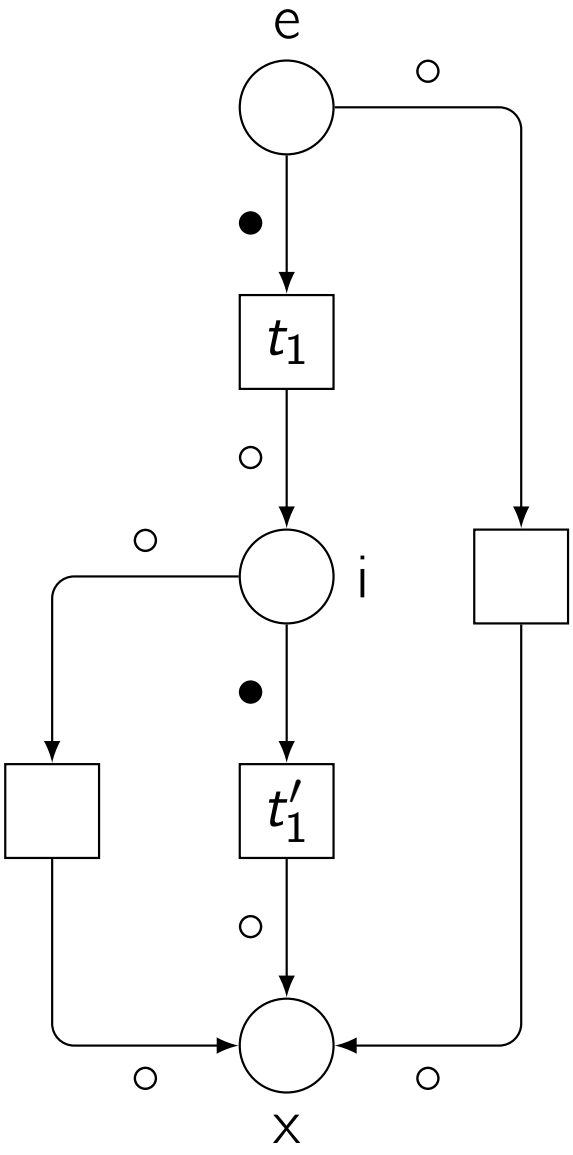
operator

operand	•	○	⋆	∅
•	•	○	•	∅
○	⋆	⋆	○	∅
∅	∅	∅	∅	∅
m	⋆	⋆	m	∅



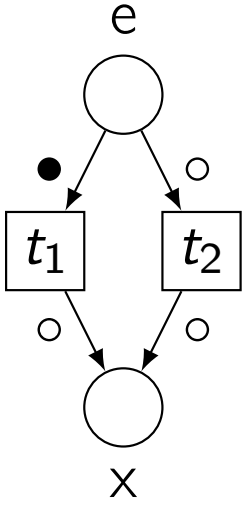


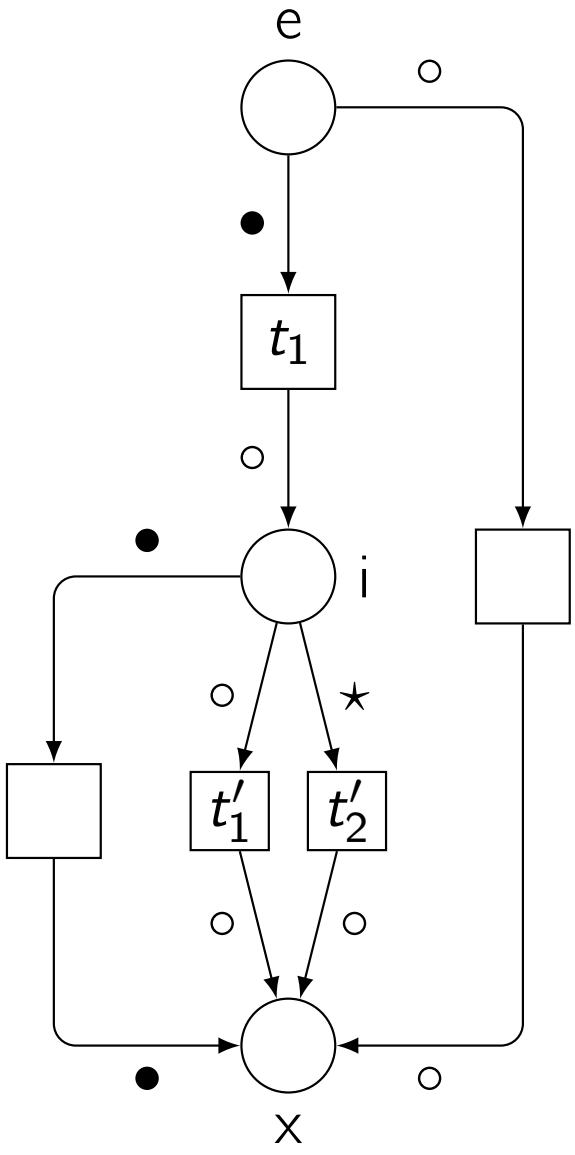
Composing nets with exceptions



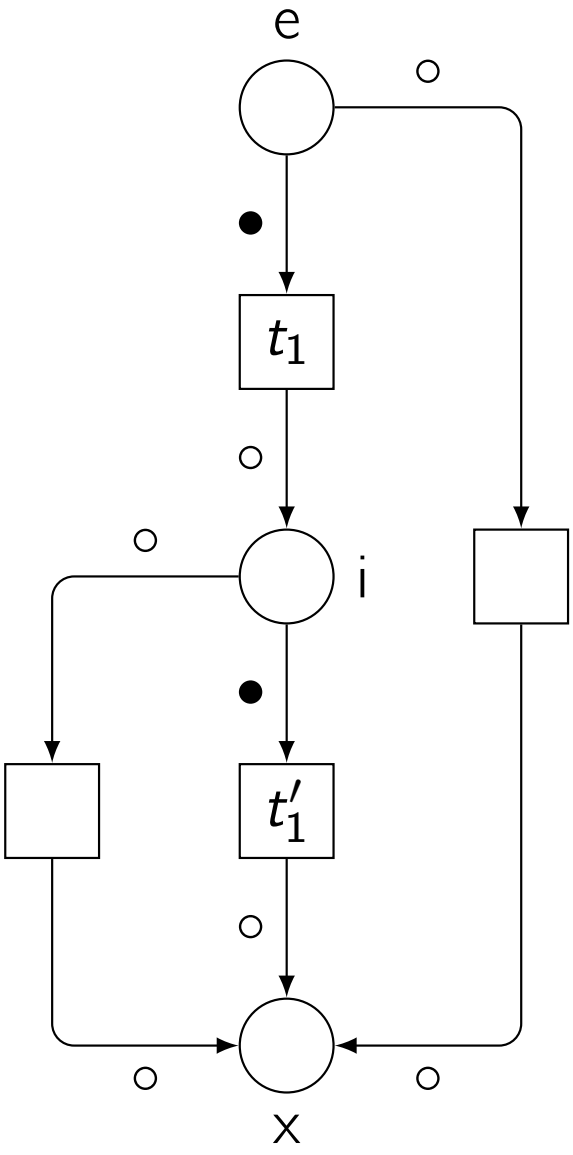
operator

operand	●	○	⋆	∅
●	●	○	●	∅
○	⋆	⋆	○	∅
∅	∅	∅	∅	∅
m	⋆	⋆	m	∅



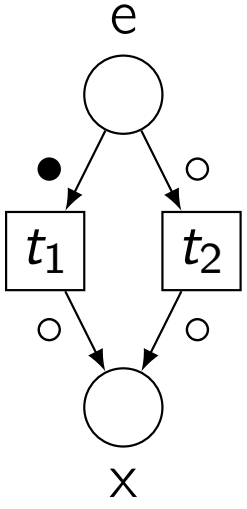


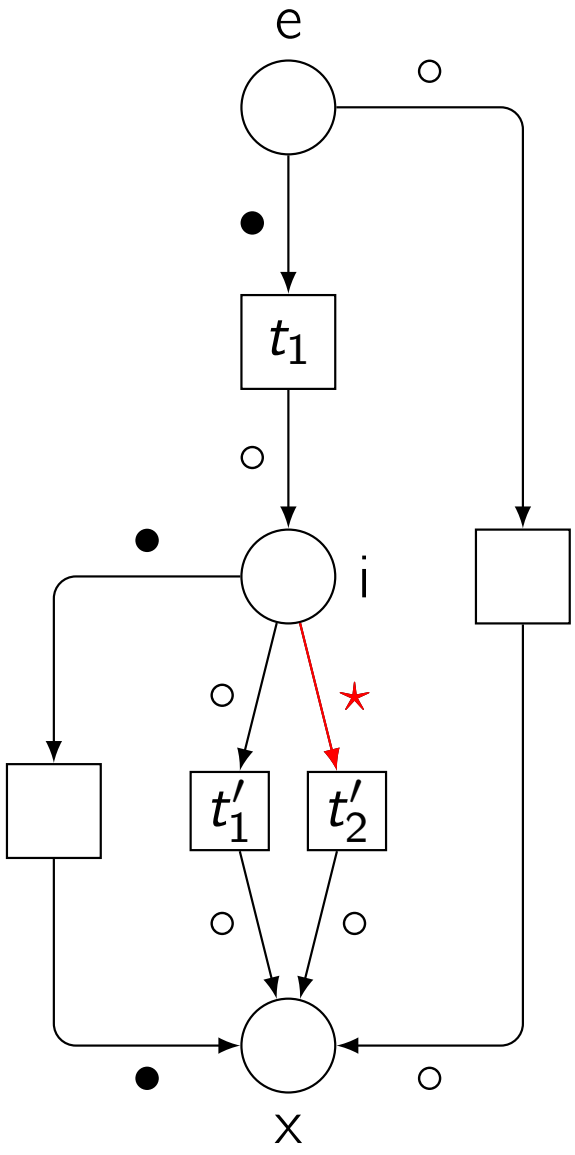
Composing nets with exceptions



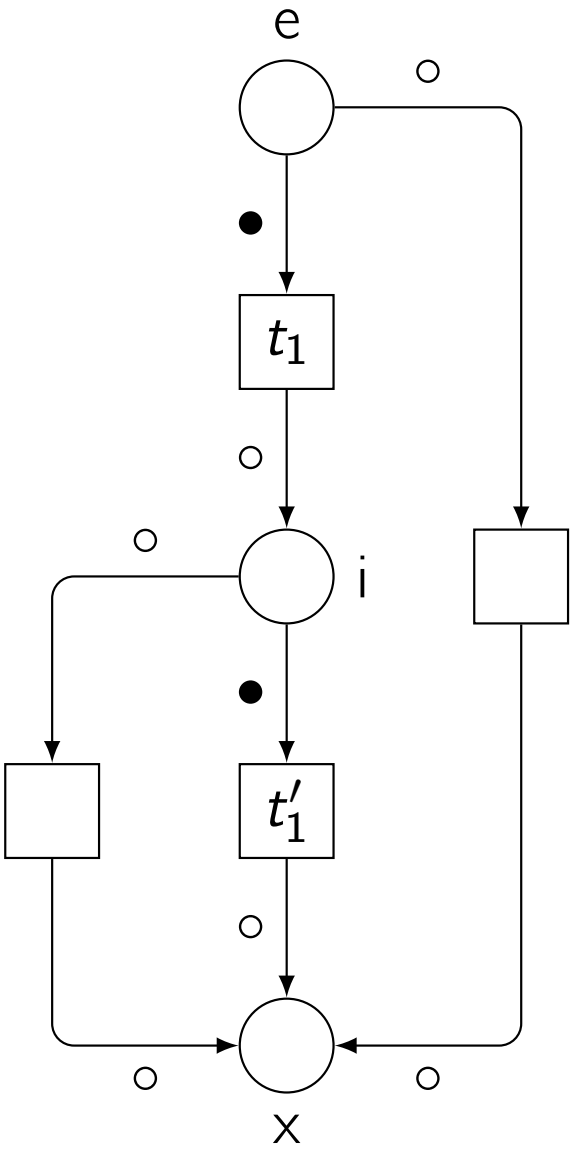
operator

operand	●	○	⋆	∅
●	●	○	●	∅
○	⋆	⋆	○	∅
∅	∅	∅	∅	∅
m	⋆	⋆	m	∅



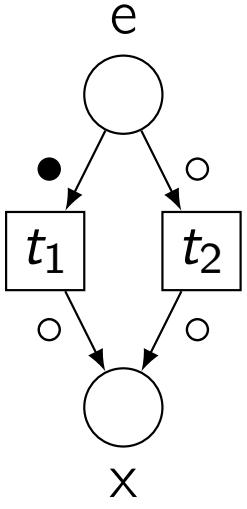


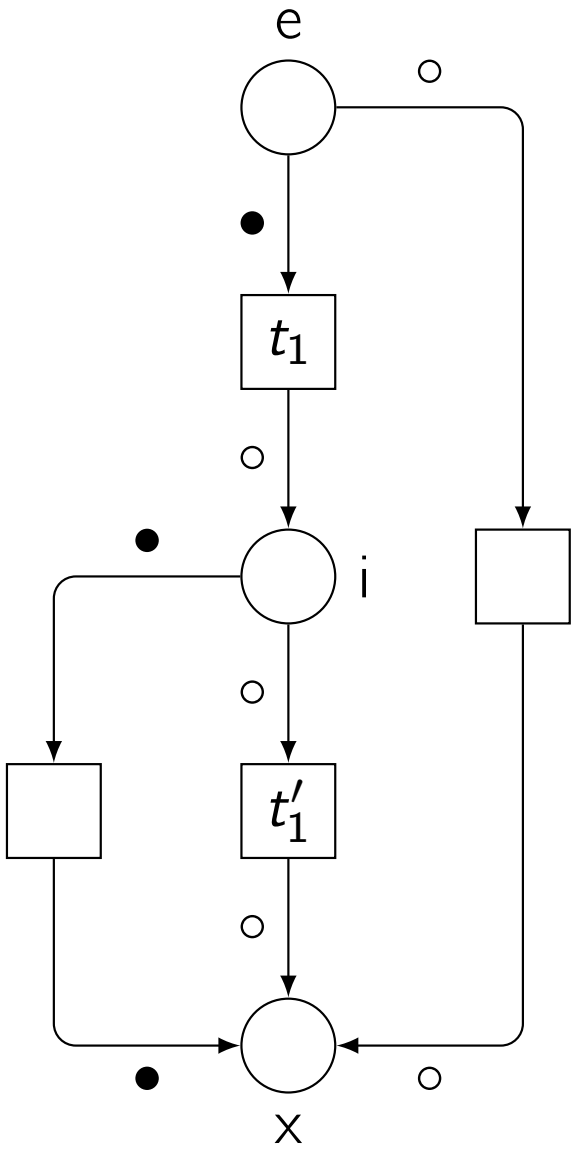
Composing nets with exceptions



operator

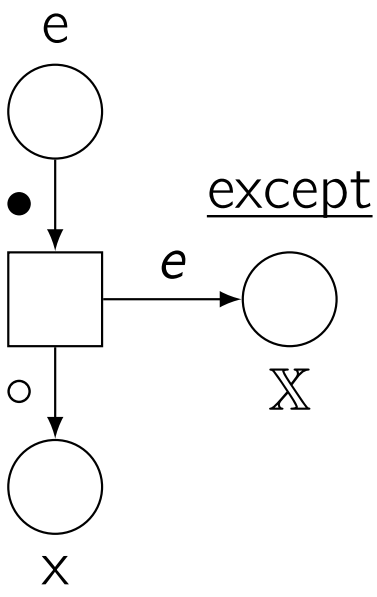
operand	●	○	⋆	∅
●	●	○	●	∅
○	⋆	⋆	○	∅
∅	∅	∅	∅	∅
m	⋆	⋆	m	∅



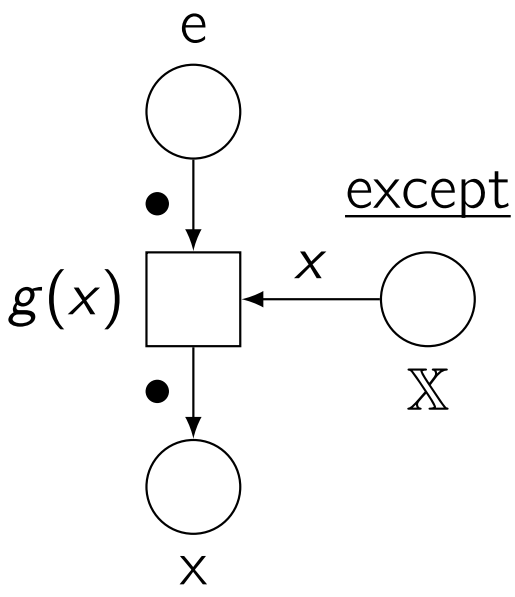


Modelling exceptions

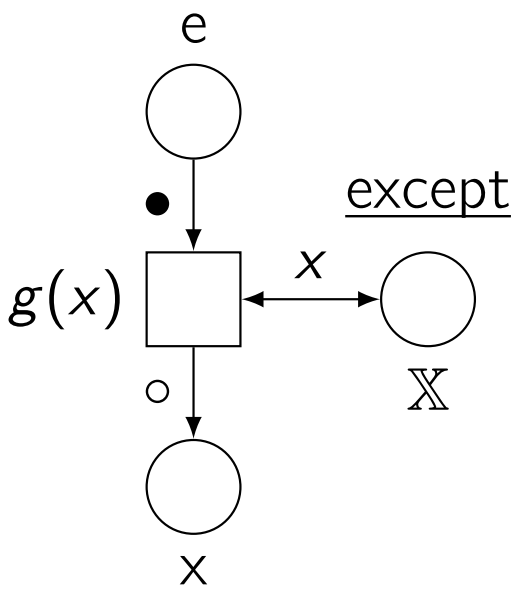
R_e: raise e



C_g: catch



P_g: propagate



Catch e_1 and e_2, propagate others

$$N \triangleright \Big((C_{\lambda x : x = e_1} \fatsemi H_1) \,\Box\, (C_{\lambda x : x = e_2} \fatsemi H_2) \,\Box\, P_{\lambda x : x \notin \{e_1, e_2\}} \Big)$$

Outline

A modular framework
- Coloured Petri nets
- Synchronous communication
- Control flow
- Exceptions
- Threads

ABCD of modelling

More than simulation

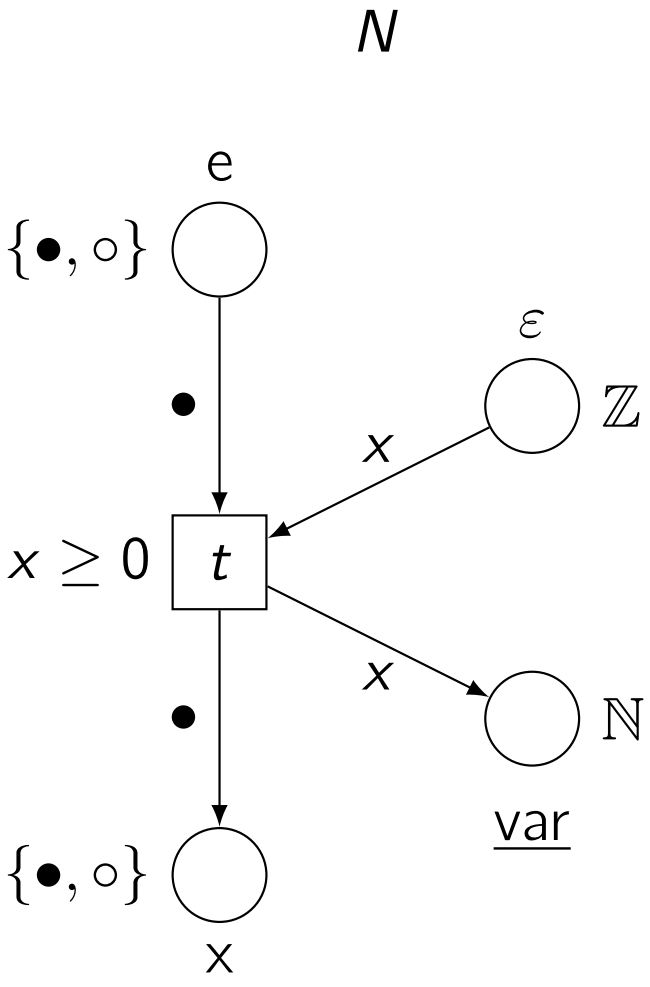
Three application domains

Verification issues

Conclusion

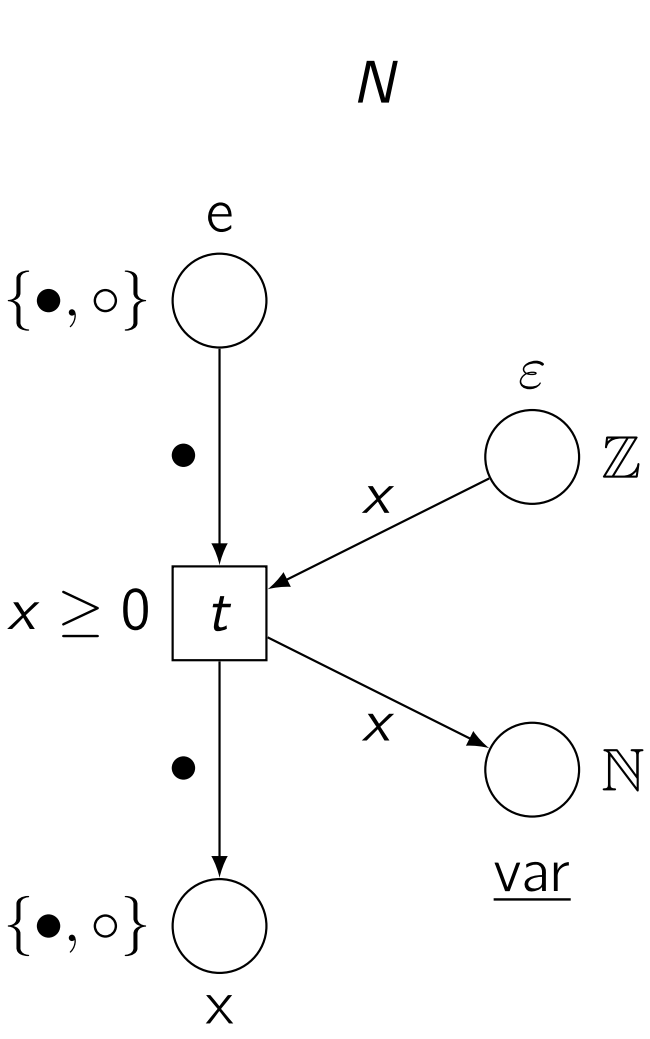
Building tasks

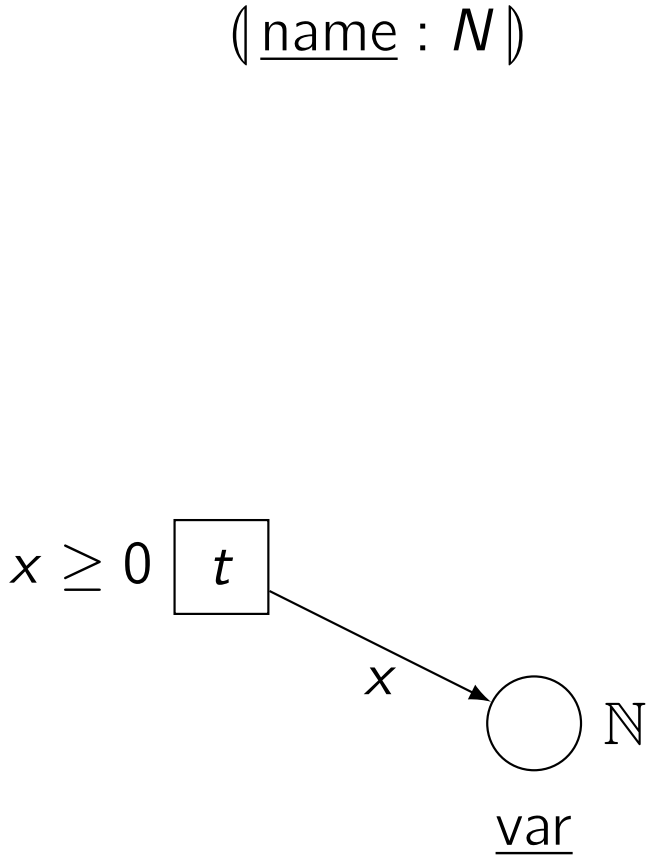
From Petri nets with exceptions to Petri nets with threads



Building tasks

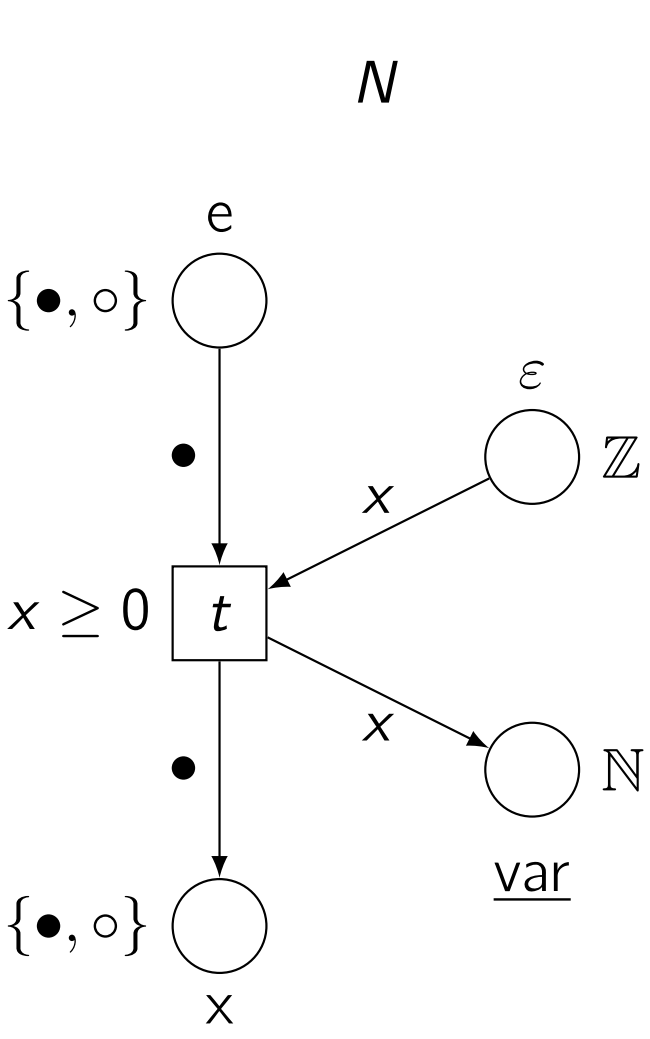
From Petri nets with exceptions to Petri nets with threads

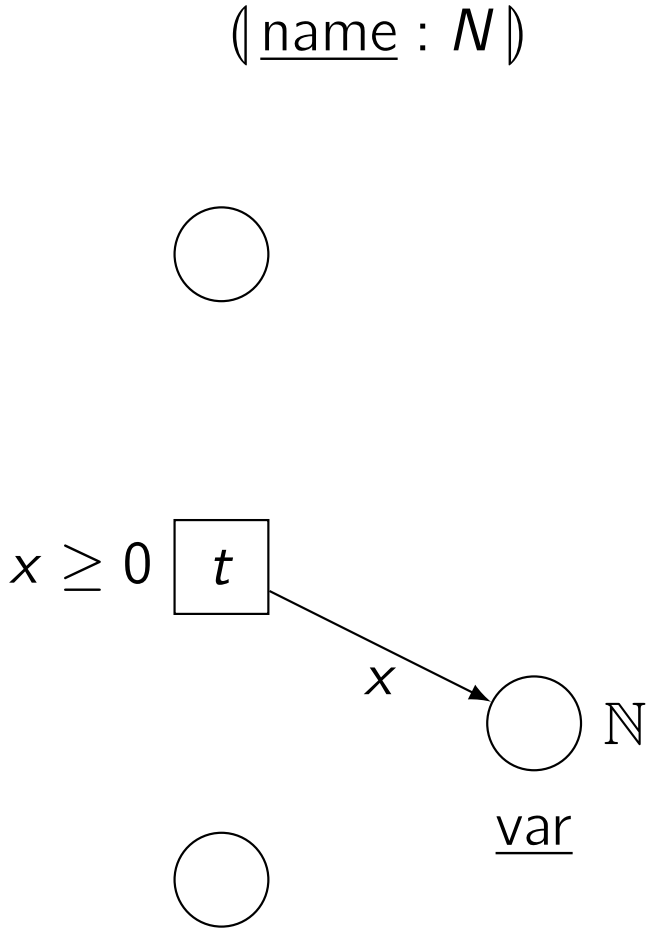




Building tasks

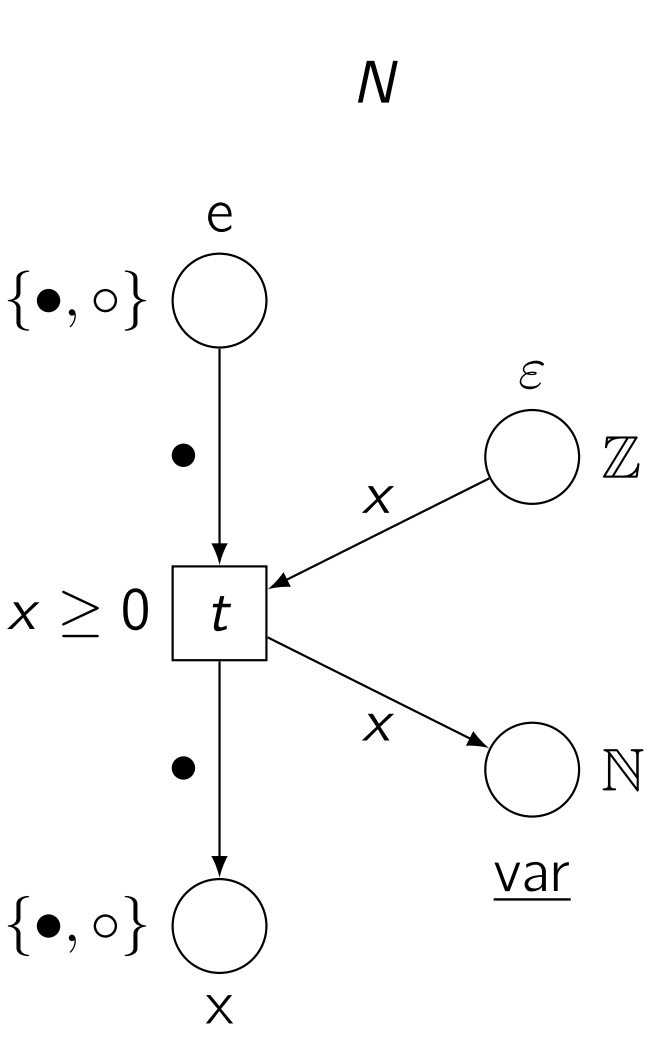
From Petri nets with exceptions to Petri nets with threads

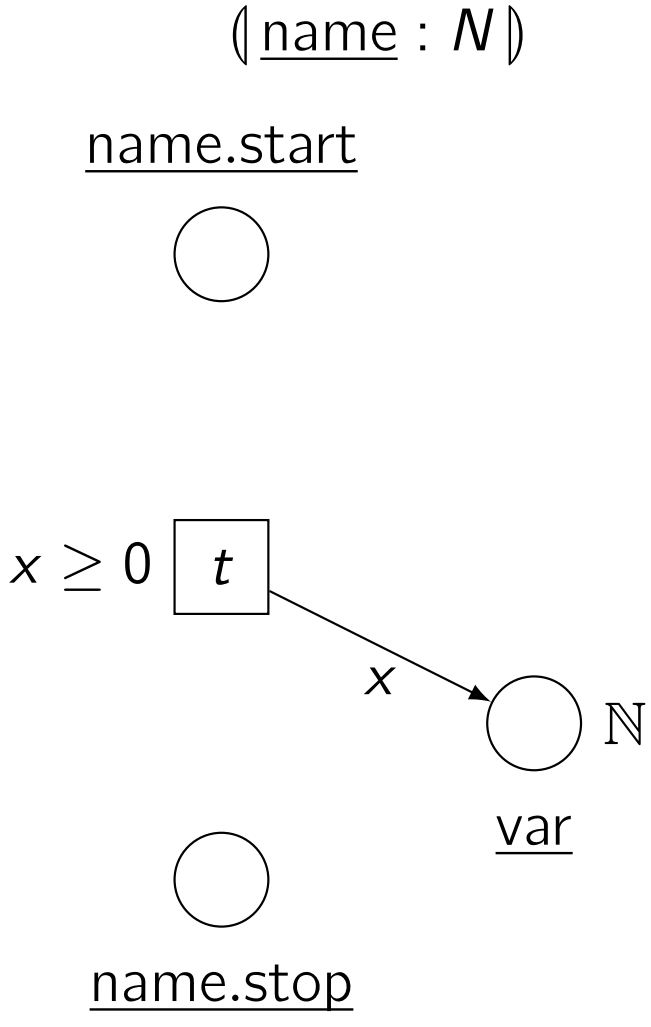




Building tasks

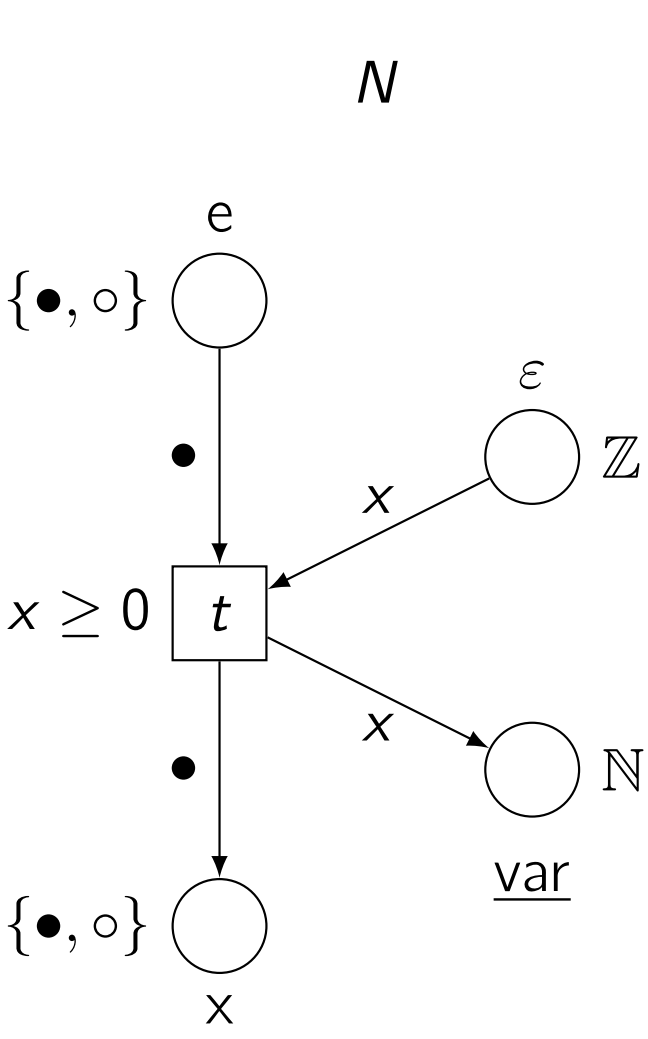
From Petri nets with exceptions to Petri nets with threads

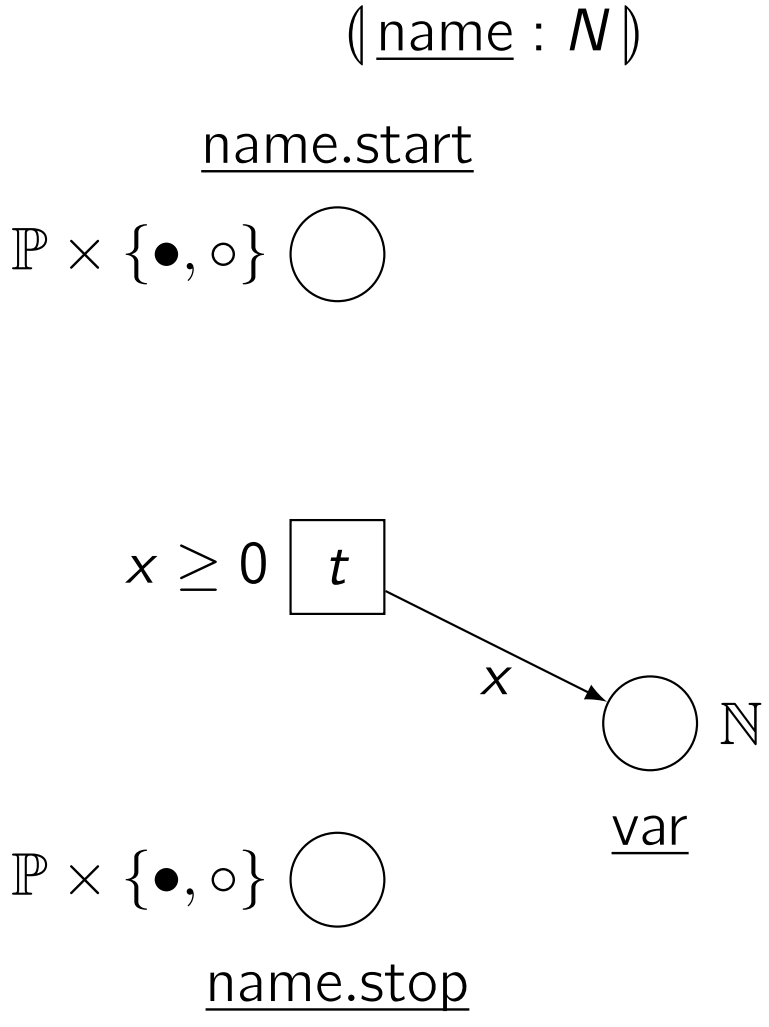




Building tasks

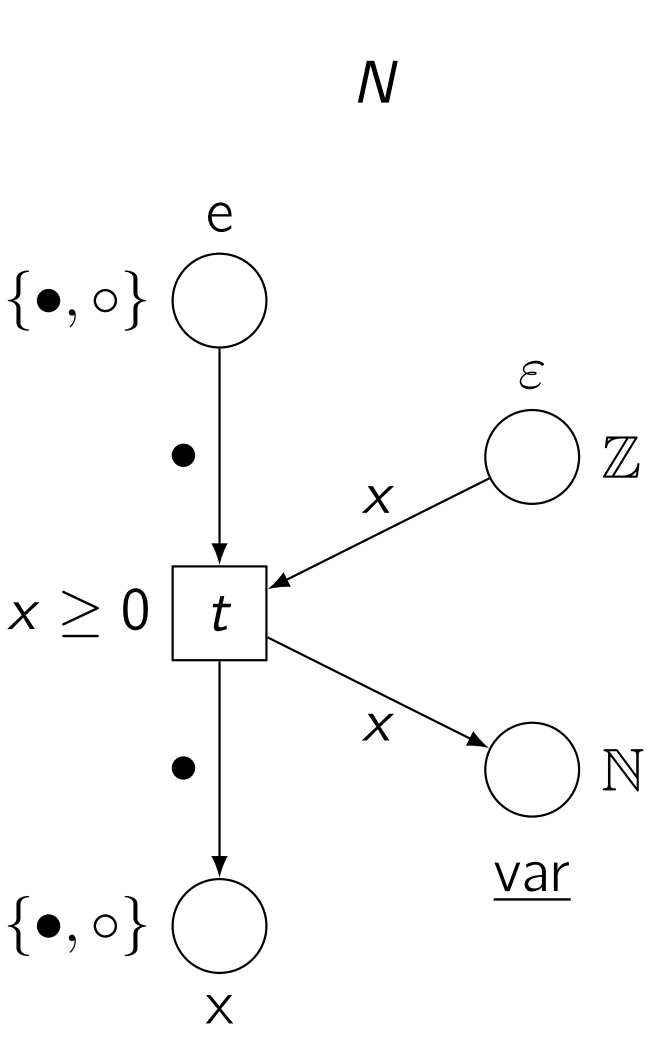
From Petri nets with exceptions to Petri nets with threads

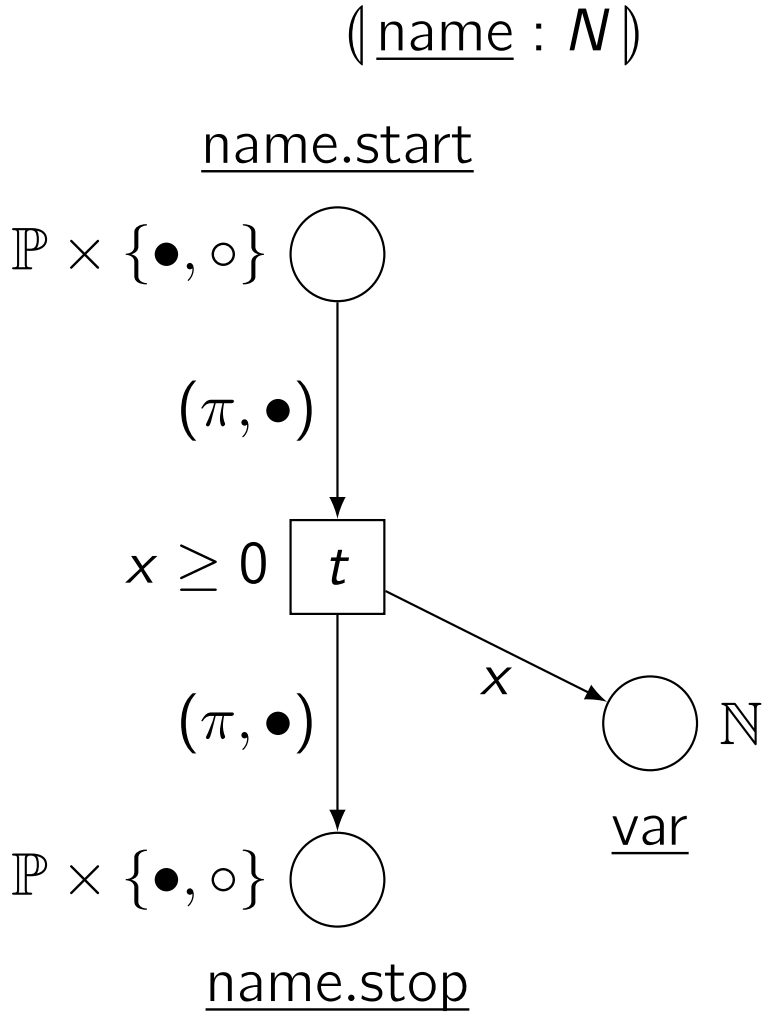




Building tasks

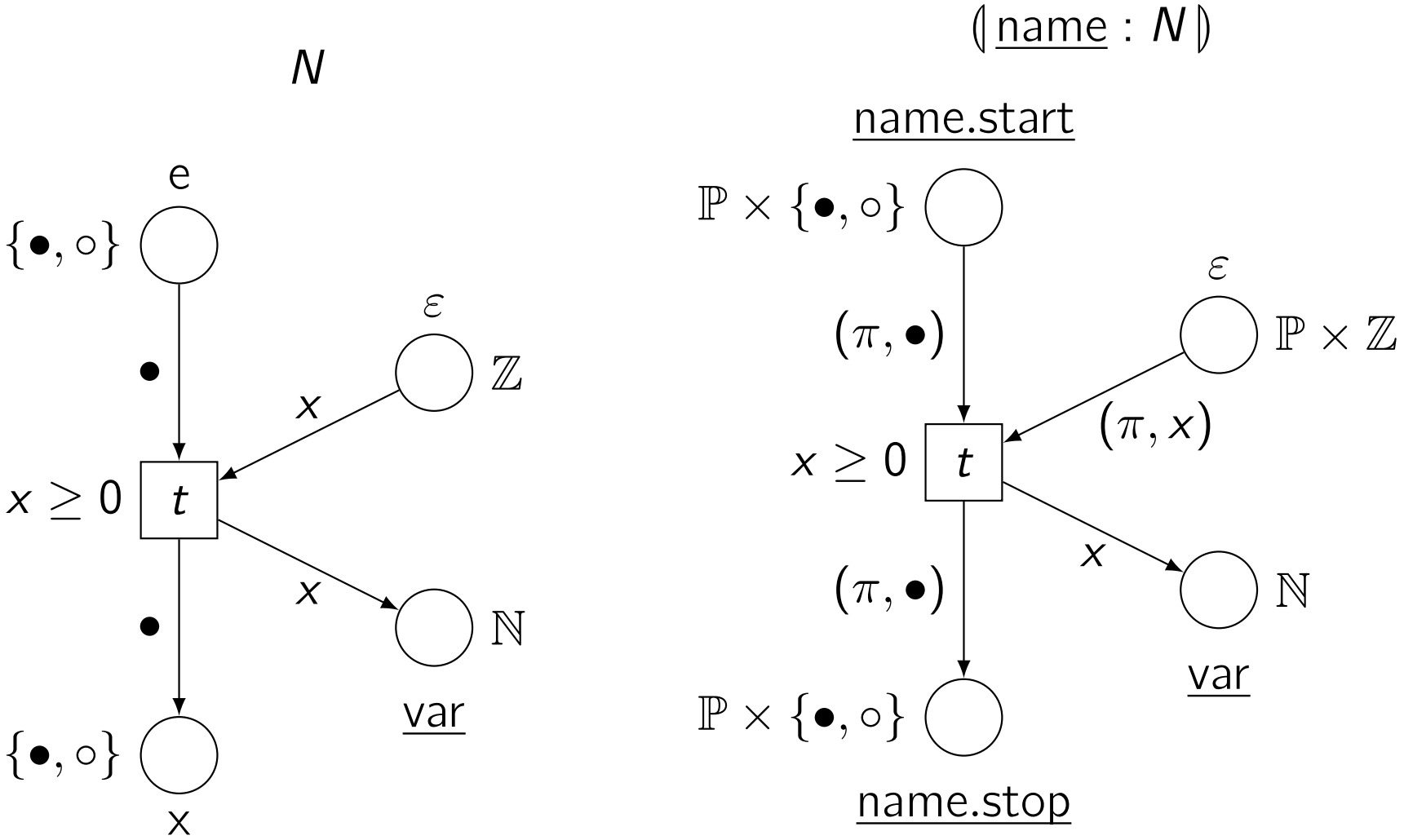
From Petri nets with exceptions to Petri nets with threads





Building tasks

From Petri nets with exceptions to Petri nets with threads

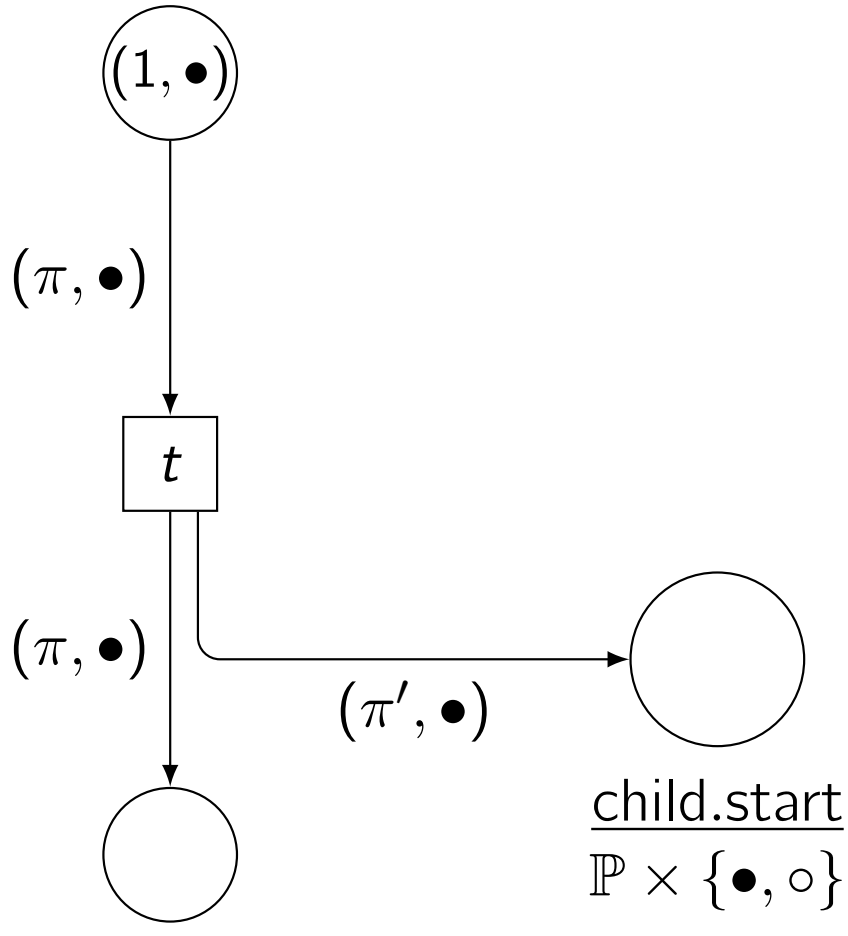


Process identifiers (pids)

- pids from $\mathbb{P}$ as sequences like 1.3.2.1
  - opaque values for the system
- relations on pids
  - parent $\pi_1 \lhd_1 \pi_2$ $1.3.2 \lhd_1 1.3.2.1$
  - ancestor $\pi_1 \lhd \pi_2$ $1.3 \lhd 1.3.2.1$
  - immediate sibling $\pi_1 \pitchfork_1 \pi_2$ $1.2 \pitchfork_1 1.3$
  - elder sibling $\pi_1 \pitchfork \pi_2$ $1.1 \pitchfork 1.3$
- advantages
  - distributed and concurrent
  - no pid reuse
  - may be bounded in width and depth
  - $\mathrm{MSO}(\mathbb{P}, \lhd_1, \pitchfork_1) \equiv$ tree automata
  - simple and easy to implement

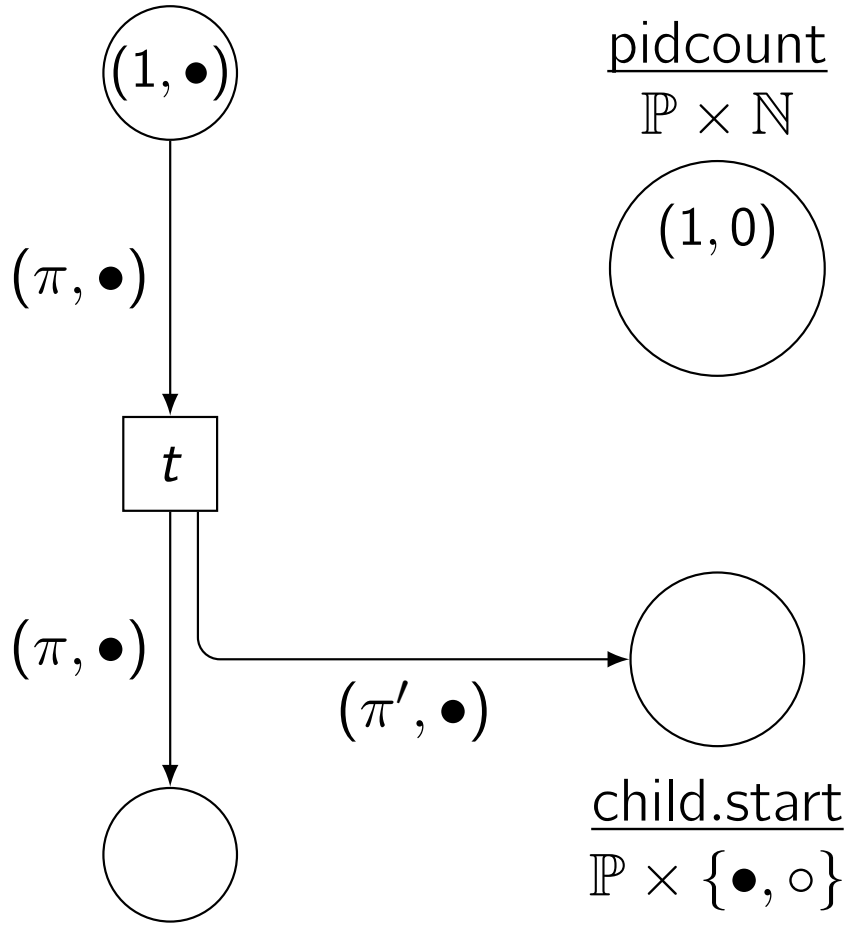
Process identifiers (pids)

- pids from $\mathbb{P}$ as sequences like 1.3.2.1
  - opaque values for the system
- relations on pids
  - parent $\pi_1 \lhd_1 \pi_2$
  - ancestor $\pi_1 \lhd \pi_2$
  - immediate sibling $\pi_1 \pitchfork_1 \pi_2$
  - elder sibling $\pi_1 \pitchfork \pi_2$
- advantages
  - distributed and concurrent
  - no pid reuse
  - may be bounded in width and depth
  - MSO$(\mathbb{P}, \lhd_1, \pitchfork_1) \equiv$ tree automata
  - simple and easy to implement



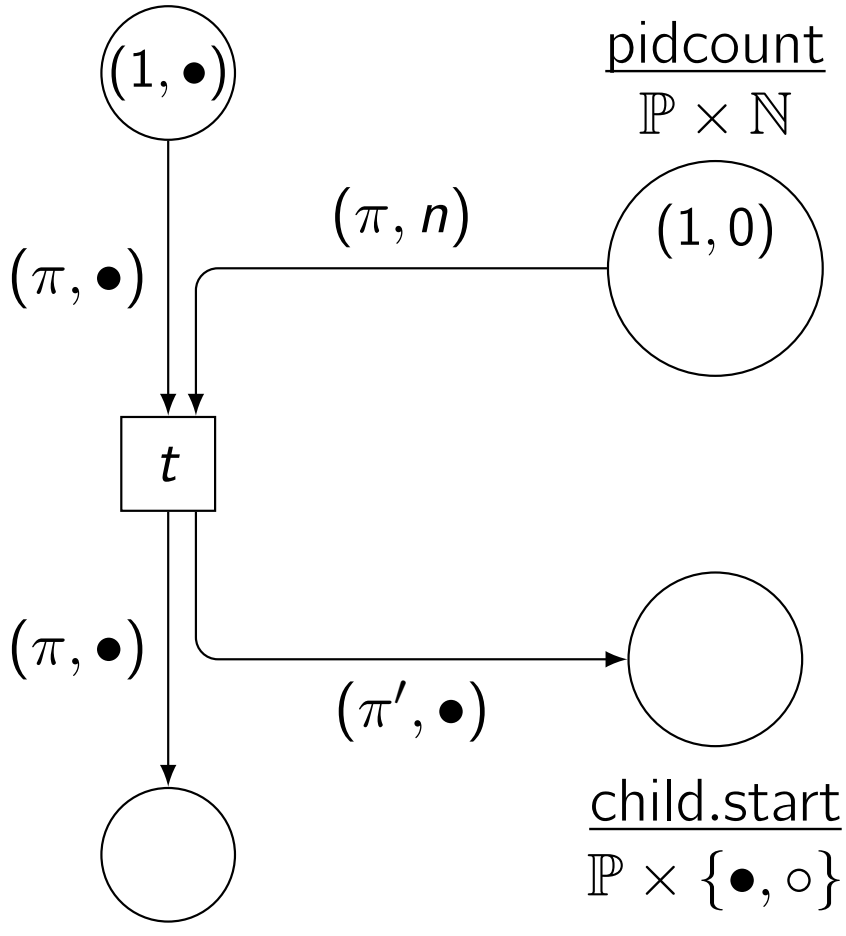
Process identifiers (pids)

- pids from $\mathbb{P}$ as sequences like 1.3.2.1
  - opaque values for the system
- relations on pids
  - parent $\pi_1 \lhd_1 \pi_2$
  - ancestor $\pi_1 \lhd \pi_2$
  - immediate sibling $\pi_1 \pitchfork_1 \pi_2$
  - elder sibling $\pi_1 \pitchfork \pi_2$
- advantages
  - distributed and concurrent
  - no pid reuse
  - may be bounded in width and depth
  - MSO$(\mathbb{P}, \lhd_1, \pitchfork_1) \equiv$ tree automata
  - simple and easy to implement



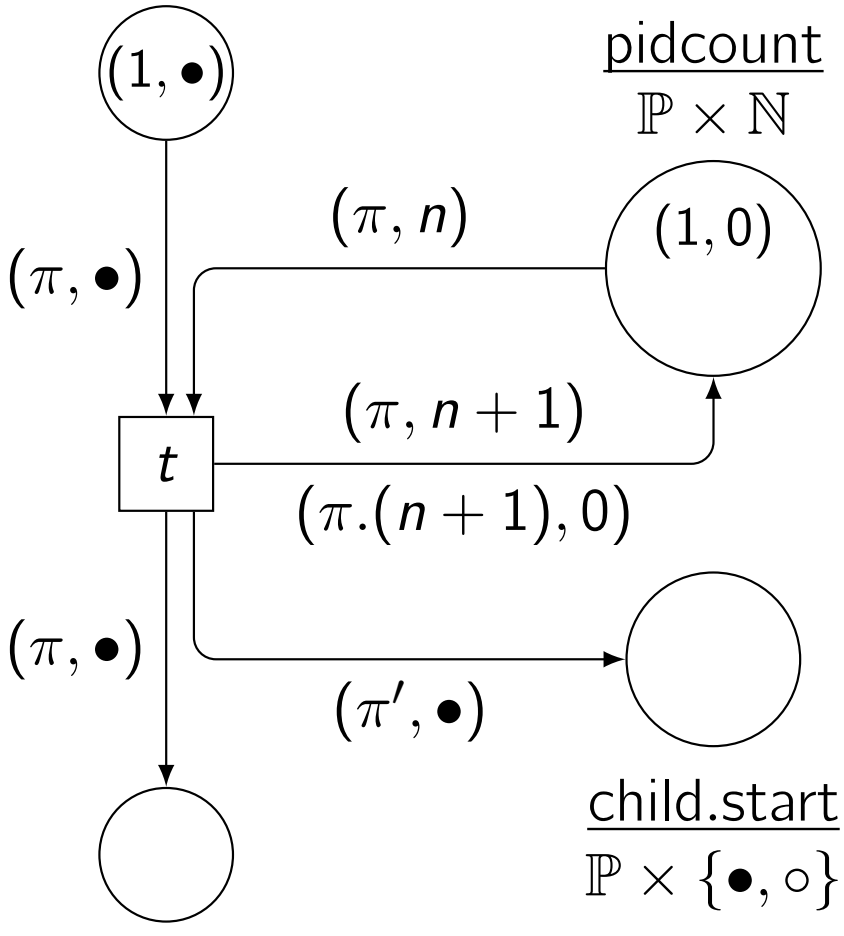
Process identifiers (pids)

- pids from $\mathbb{P}$ as sequences like 1.3.2.1
  - opaque values for the system
- relations on pids
  - parent $\pi_1 \lhd_1 \pi_2$
  - ancestor $\pi_1 \lhd \pi_2$
  - immediate sibling $\pi_1 \pitchfork_1 \pi_2$
  - elder sibling $\pi_1 \pitchfork \pi_2$
- advantages
  - distributed and concurrent
  - no pid reuse
  - may be bounded in width and depth
  - MSO$(\mathbb{P}, \lhd_1, \pitchfork_1) \equiv$ tree automata
  - simple and easy to implement



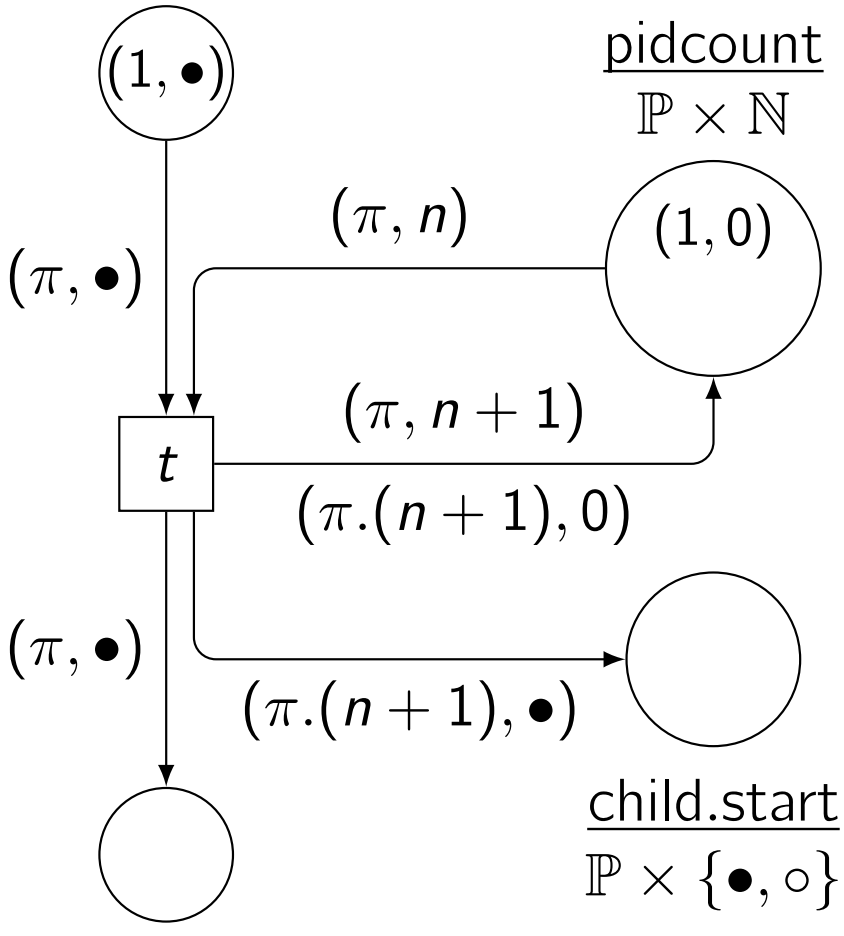
Process identifiers (pids)

- pids from $\mathbb{P}$ as sequences like 1.3.2.1
  - opaque values for the system
- relations on pids
  - parent $\pi_1 \lhd_1 \pi_2$
  - ancestor $\pi_1 \lhd \pi_2$
  - immediate sibling $\pi_1 \pitchfork_1 \pi_2$
  - elder sibling $\pi_1 \pitchfork \pi_2$
- advantages
  - distributed and concurrent
  - no pid reuse
  - may be bounded in width and depth
  - MSO$(\mathbb{P}, \lhd_1, \pitchfork_1) \equiv$ tree automata
  - simple and easy to implement



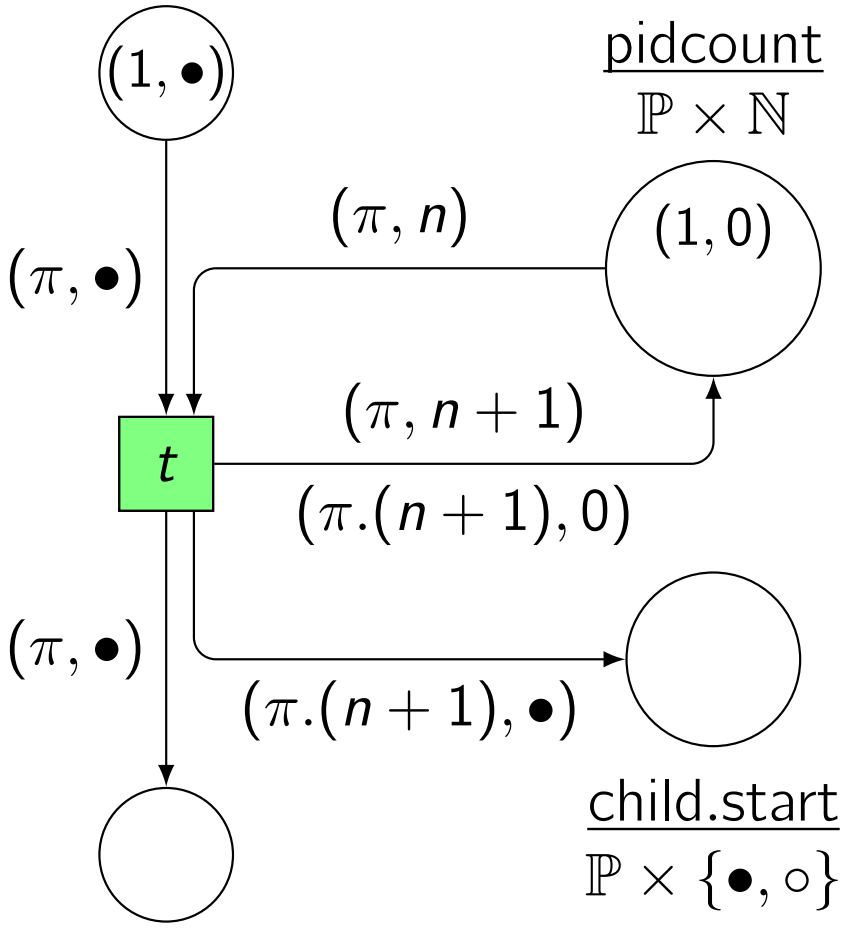
Process identifiers (pids)

- pids from $\mathbb{P}$ as sequences like 1.3.2.1
  - opaque values for the system
- relations on pids
  - parent $\pi_1 \lhd_1 \pi_2$
  - ancestor $\pi_1 \lhd \pi_2$
  - immediate sibling $\pi_1 \pitchfork_1 \pi_2$
  - elder sibling $\pi_1 \pitchfork \pi_2$
- advantages
  - distributed and concurrent
  - no pid reuse
  - may be bounded in width and depth
  - MSO$(\mathbb{P}, \lhd_1, \pitchfork_1) \equiv$ tree automata
  - simple and easy to implement



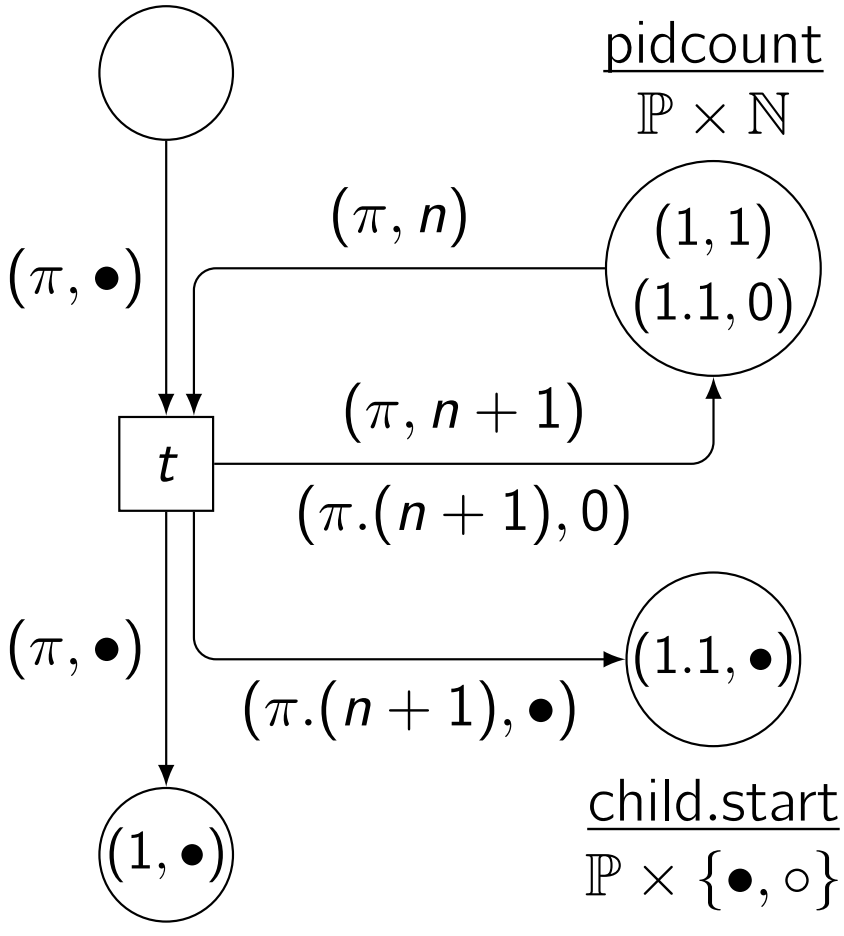
Process identifiers (pids)

- pids from $\mathbb{P}$ as sequences like 1.3.2.1
  - opaque values for the system
- relations on pids
  - parent $\pi_1 \lhd_1 \pi_2$
  - ancestor $\pi_1 \lhd \pi_2$
  - immediate sibling $\pi_1 \pitchfork_1 \pi_2$
  - elder sibling $\pi_1 \pitchfork \pi_2$
- advantages
  - distributed and concurrent
  - no pid reuse
  - may be bounded in width and depth
  - MSO$(\mathbb{P}, \lhd_1, \pitchfork_1) \equiv$ tree automata
  - simple and easy to implement



Process identifiers (pids)

- pids from $\mathbb{P}$ as sequences like 1.3.2.1
  - opaque values for the system
- relations on pids
  - parent $\pi_1 \lhd_1 \pi_2$
  - ancestor $\pi_1 \lhd \pi_2$
  - immediate sibling $\pi_1 \pitchfork_1 \pi_2$
  - elder sibling $\pi_1 \pitchfork \pi_2$
- advantages
  - distributed and concurrent
  - no pid reuse
  - may be bounded in width and depth
  - MSO$(\mathbb{P}, \lhd_1, \pitchfork_1) \equiv$ tree automata
  - simple and easy to implement

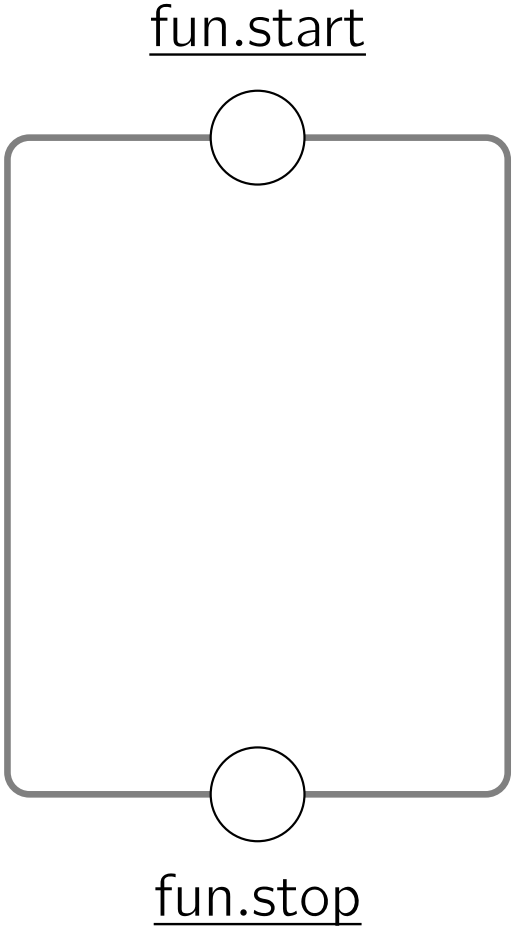


Modelling threads and sub-programs

$$(\!|\, \underline{\mathsf{fun}} : N_1 \,|\!) \;\|\; (\!|\, \underline{\mathsf{task}} : N_2 \,|\!) \;\|\; (\!|\, \underline{\mathsf{main}} : N_3 \,|\!)$$

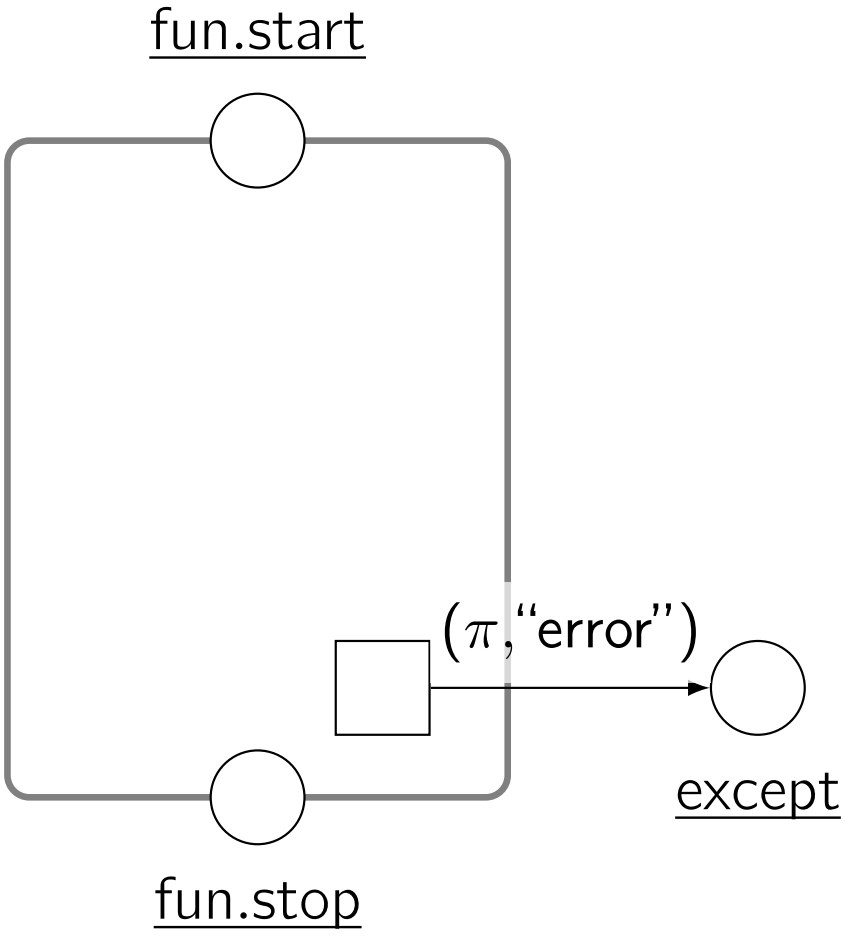
Modelling threads and sub-programs

$$(\!|\, \underline{\mathsf{fun}} : N_1 \,|\!) \;\|\; (\!|\, \underline{\mathsf{task}} : N_2 \,|\!) \;\|\; (\!|\, \underline{\mathsf{main}} : N_3 \,|\!)$$



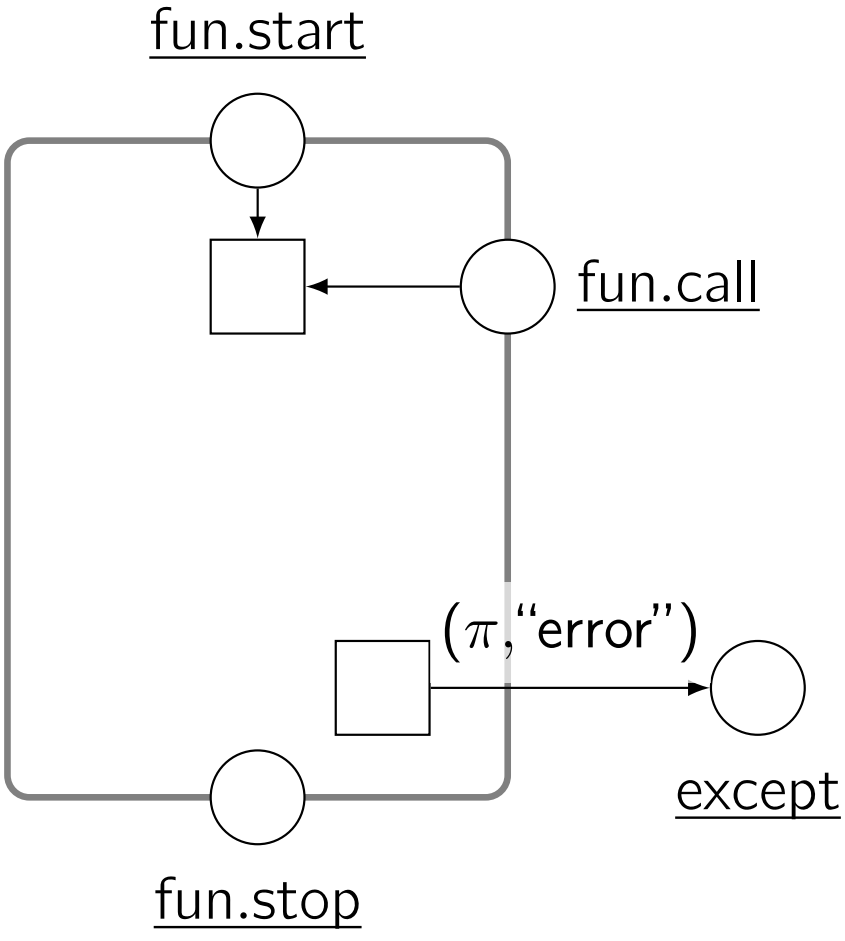
Modelling threads and sub-programs

$$(\!|\, \underline{\text{fun}} : N_1 \,|\!) \;\|\; (\!|\, \underline{\text{task}} : N_2 \,|\!) \;\|\; (\!|\, \underline{\text{main}} : N_3 \,|\!)$$



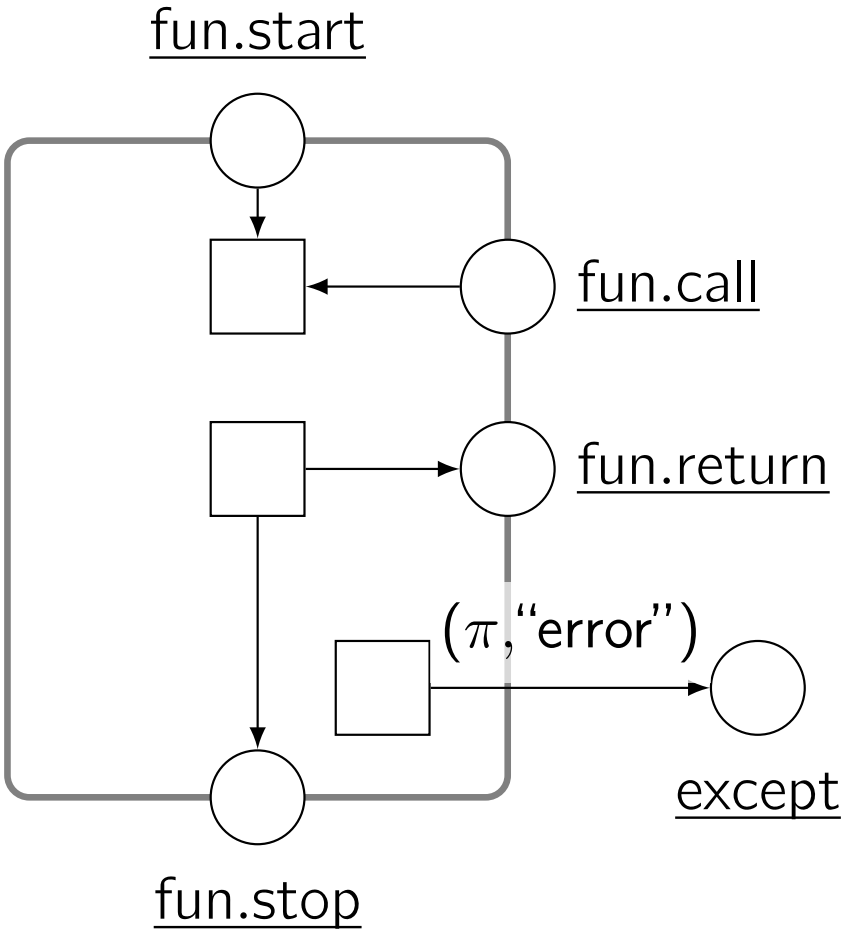
Modelling threads and sub-programs

$$(\!|\, \underline{\text{fun}} : N_1 \,|\!) \;||\; (\!|\, \underline{\text{task}} : N_2 \,|\!) \;||\; (\!|\, \underline{\text{main}} : N_3 \,|\!)$$



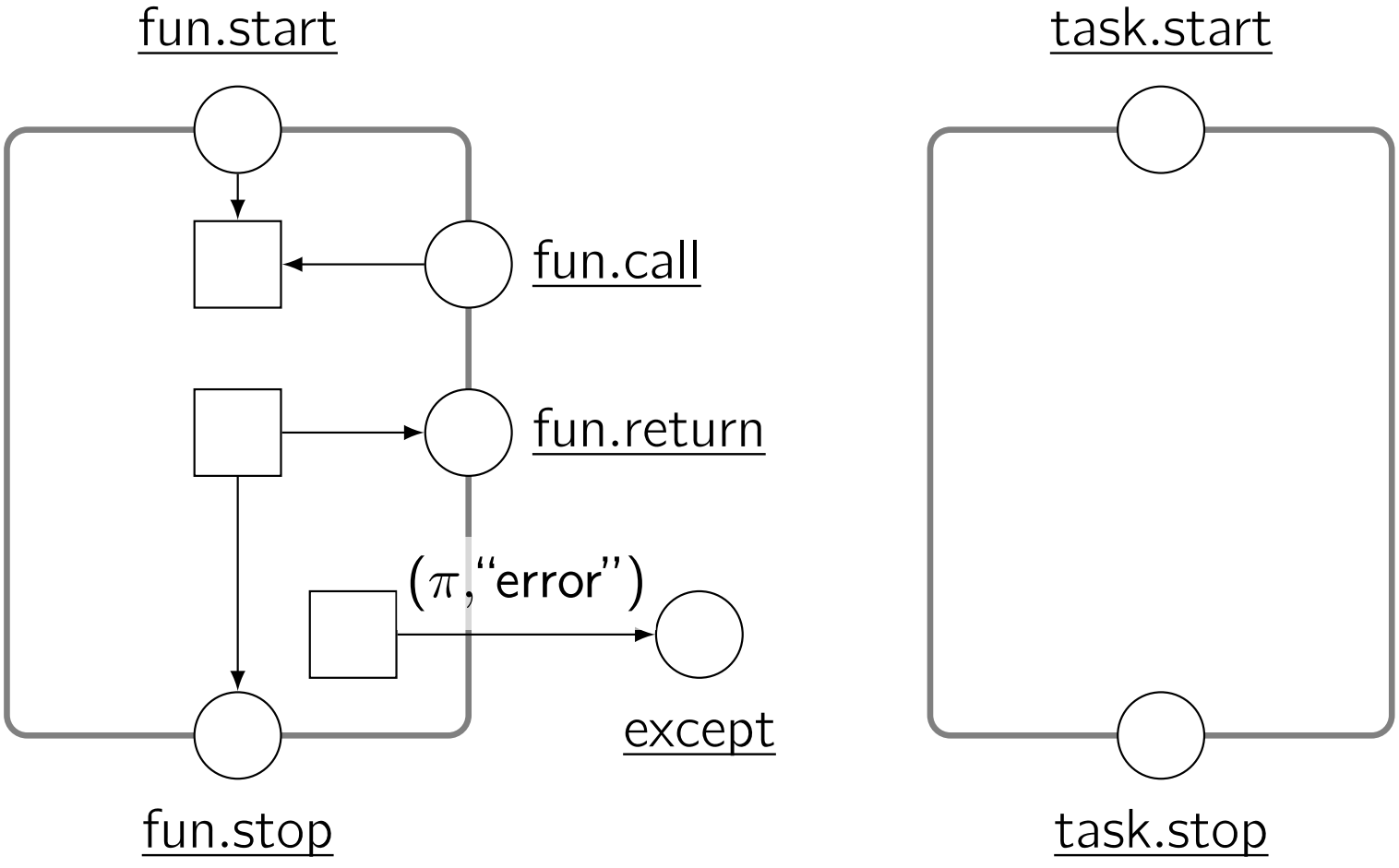
Modelling threads and sub-programs

$$(\!|\, \underline{\text{fun}} : N_1 \,|\!) \;||\; (\!|\, \underline{\text{task}} : N_2 \,|\!) \;||\; (\!|\, \underline{\text{main}} : N_3 \,|\!)$$



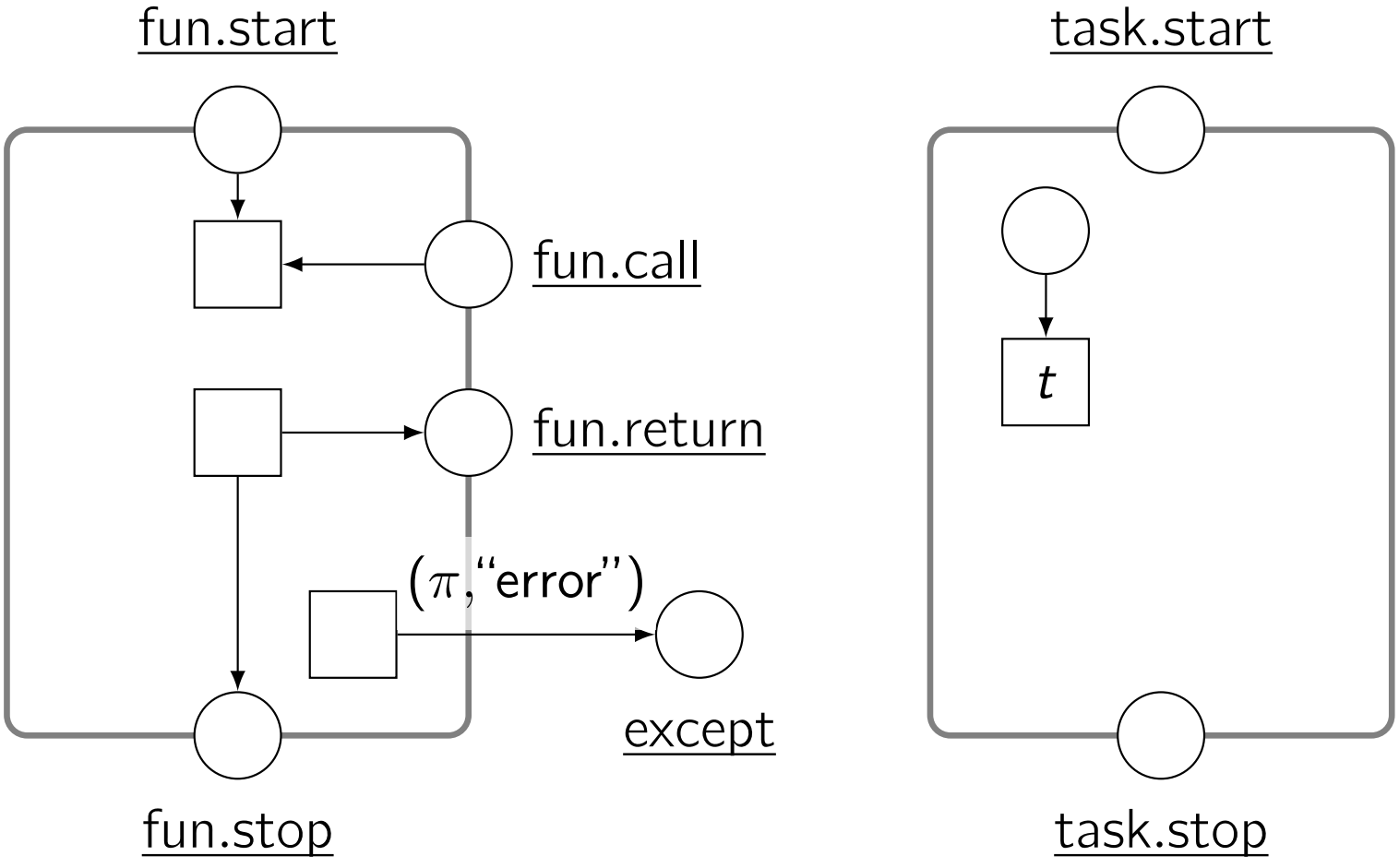
Modelling threads and sub-programs

$$(\!|\, \underline{\text{fun}} : N_1 \,|\!) \;\|\; (\!|\, \underline{\text{task}} : N_2 \,|\!) \;\|\; (\!|\, \underline{\text{main}} : N_3 \,|\!)$$



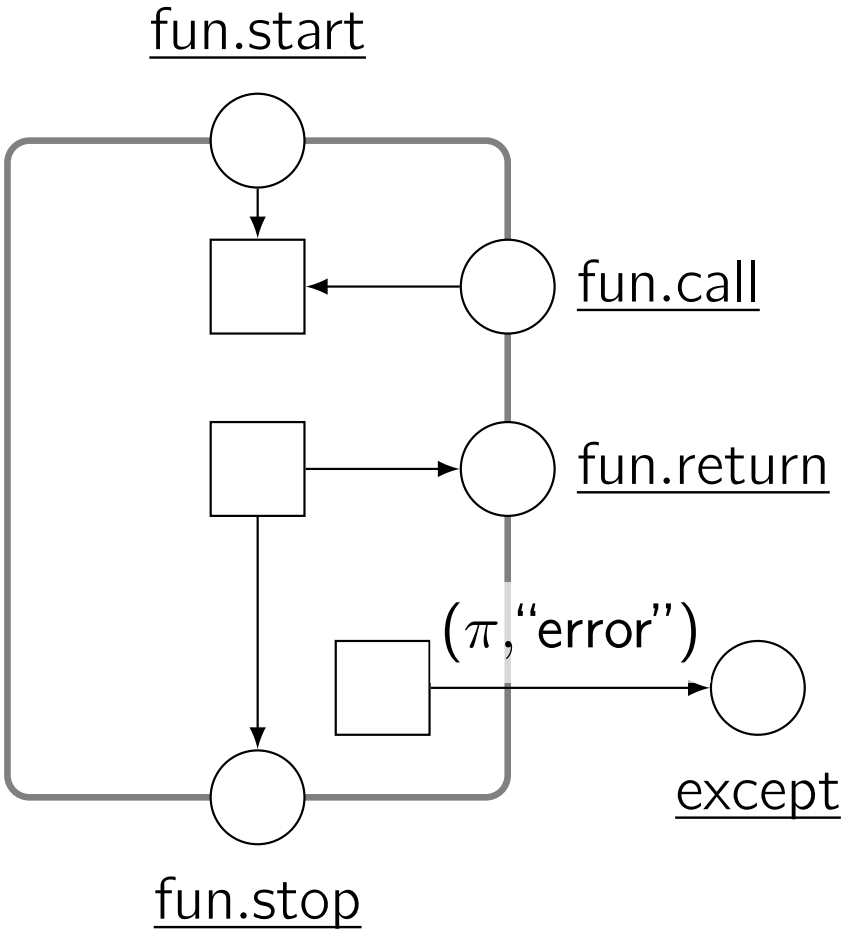
Modelling threads and sub-programs

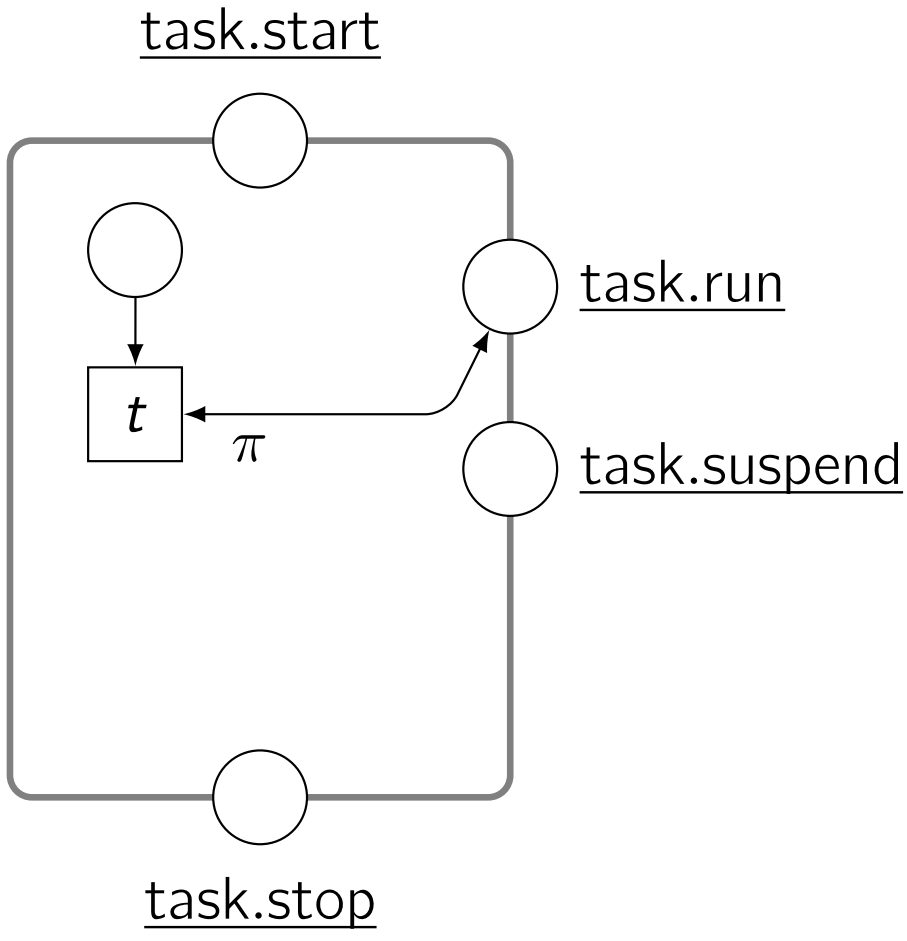
$$(\!|\, \underline{\text{fun}} : N_1 \,|\!) \;||\; (\!|\, \underline{\text{task}} : N_2 \,|\!) \;||\; (\!|\, \underline{\text{main}} : N_3 \,|\!)$$



Modelling threads and sub-programs

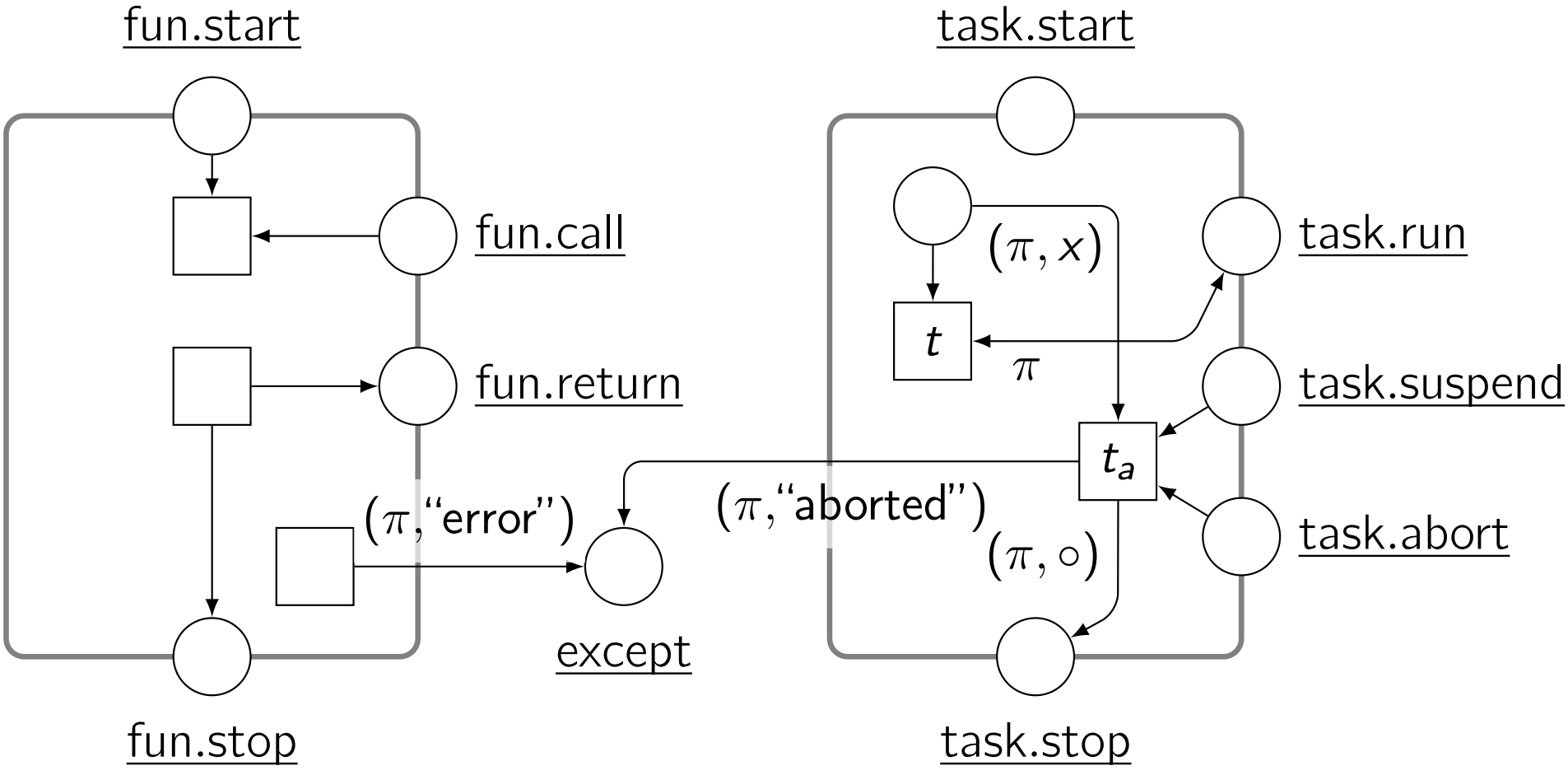
$$(\!|\, \underline{\text{fun}} : N_1 \,|\!) \;||\; (\!|\, \underline{\text{task}} : N_2 \,|\!) \;||\; (\!|\, \underline{\text{main}} : N_3 \,|\!)$$





Modelling threads and sub-programs

$$(\!|\, \underline{\text{fun}} : N_1 \,|\!) \;||\; (\!|\, \underline{\text{task}} : N_2 \,|\!) \;||\; (\!|\, \underline{\text{main}} : N_3 \,|\!)$$



Outline

A modular framework

ABCD of modelling
- Syntax and intuitive semantics
- Petri nets semantics

More than simulation

Three application domains

Verification issues

Conclusion

ABCD: Asynchronous Box Calculus with Data

- a concrete syntax for coloured Petri nets
  - with control flow
  - with a rich colour language Python
- a specification language
- a compiler: `spec.abcd` $\mapsto$ Petri net
- an analyser simulation, reachability, CTL* model-checking (soon)
- all included in the SNAKES toolkit
  - free software
  - works on your computer

Outline

A modular framework

ABCD of modelling
- Syntax and intuitive semantics
- Petri nets semantics

More than simulation

Three application domains

Verification issues

Conclusion

Hello world

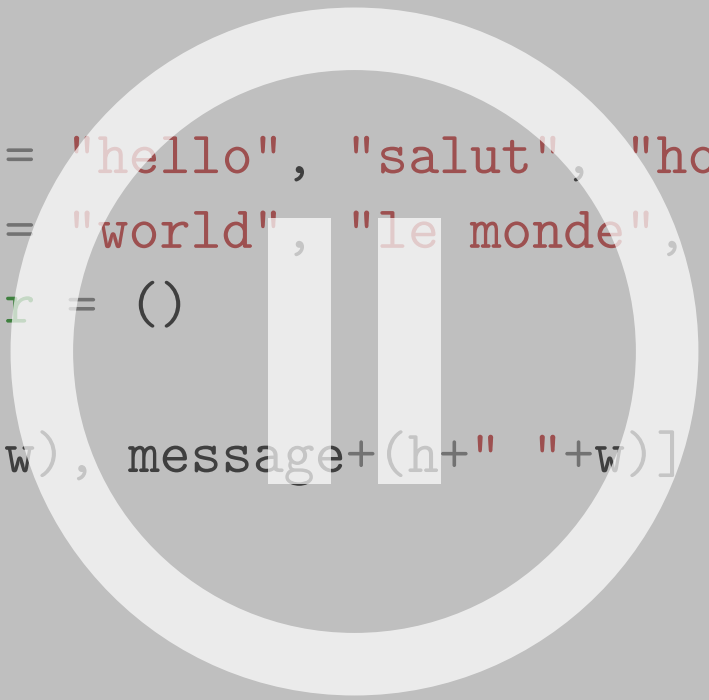
```
buffer hello : str = "hello", "salut", "hola", "ciao"
buffer world : str = "world", "le monde", "mundo", "mondo"
buffer message : str = ()

[hello-(h), world-(w), message+(h+" "+w)]
```

Hello world

```
buffer hello : str = "hello", "salut", "hola", "ciao"
buffer world : str = "world", "le monde", "mundo", "mondo"
buffer message : str = ()

[hello-(h), world-(w), message+(h+" "+w)]
```



hello.abcd

Hello world

```
buffer hello : str = "hello", "salut", "hola", "ciao"
buffer world : str = "world", "le monde", "mundo", "mondo"
buffer message : str = ()

[hello-(h), world-(w), message+(h+" "+w)]
```

Hello world

```
buffer hello : str = "hello", "salut", "hola", "ciao"
buffer world : str = "world", "le monde", "mundo", "mondo"
buffer message : str = ()

[hello-(h), world-(w), message+(h+" "+w)]
* [False]
```

Hello world

```
buffer hello : str = "hello", "salut", "hola", "ciao"
buffer world : str = "world", "le monde", "mundo", "mondo"
buffer message : str = ()

[hello-(h), world-(w), message+(h+" "+w)]
* [False]
```

Hello world

```
buffer hello : str = "hello", "salut", "hola", "ciao"
buffer world : str = "world", "le monde", "mundo", "mondo"
buffer message : str = ()

[hello-(h), world-(w), message+(h+" "+w)]
* [False]
```

Control flow (and comments)

```
# sequence
[foo+(1)] ; [foo-(x)]

# choice
[foo+(2)] + [foo+(3)]

# iteration
[foo+(4)] * [foo-(y)]

# parallel
[foo+(5)] | [foo+(6)]
```

Warning

no priorities $\Rightarrow$ use parentheses

Atomic actions and buffer accesses

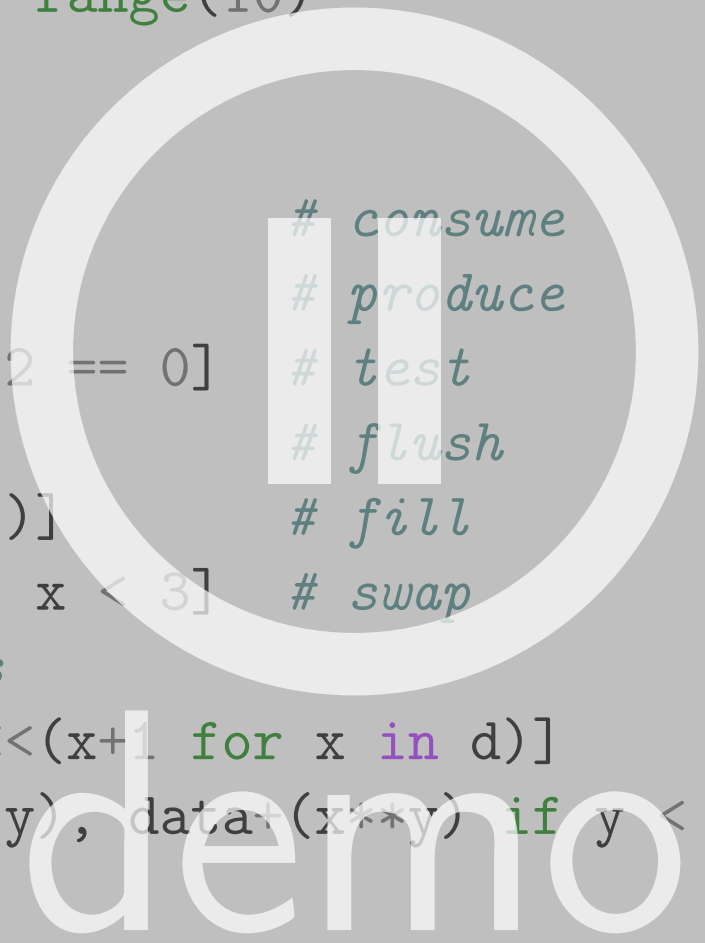
```
buffer data : int = range(10)

# single access
[data-(x) if x > 4]              # consume
+ [data+(42)]                    # produce
+ [data?(x) if x % 2 == 0]       # test
+ [data>>(d)]                    # flush
+ [data<<(range(10))]            # fill
+ [data<>(x=x+1) if x < 3]       # swap
# combined accesses
+ [data>>(d), data<<(x+1 for x in d)]
+ [data-(x), data?(y), data+(x**y) if y < x < 4]
```

Atomic actions and buffer accesses

```
buffer data : int = range(10)

# single access
[data-(x) if x > 4]                  # consume
+ [data+(42)]                        # produce
+ [data?(x) if x % 2 == 0]           # test
+ [data>>(d)]                        # flush
+ [data<<(range(10))]                # fill
+ [data<>(x=x+1) if x < 3]           # swap
# combined accesses
+ [data>>(d), data<<(x+1 for x in d)]
+ [data-(x), data?(y), data+(x**y) if y < x < 4]
```



Atomic actions and buffer accesses

```
buffer data : int = range(10)

# single access
[data-(x) if x > 4]              # consume
+ [data+(42)]                    # produce
+ [data?(x) if x % 2 == 0]       # test
+ [data>>(d)]                    # flush
+ [data<<(range(10))]            # fill
+ [data<>(x=x+1) if x < 3]       # swap
# combined accesses
+ [data>>(d), data<<(x+1 for x in d)]
+ [data-(x), data?(y), data+(x**y) if y < x < 4]
```

Sub-processes

```
buffer bag : int = ()

net prod () :
    buffer count : int = 0
    [count-(x), count+(x+1), bag+(x) if x < 10]
    * [count-(x) if x == 10]

net odd () :
    [bag-(x) if (x % 2) == 1] * [False]

net even () :
    [bag-(x) if (x % 2) == 0] * [False]

odd() | even() | prod()
```

Sub-processes

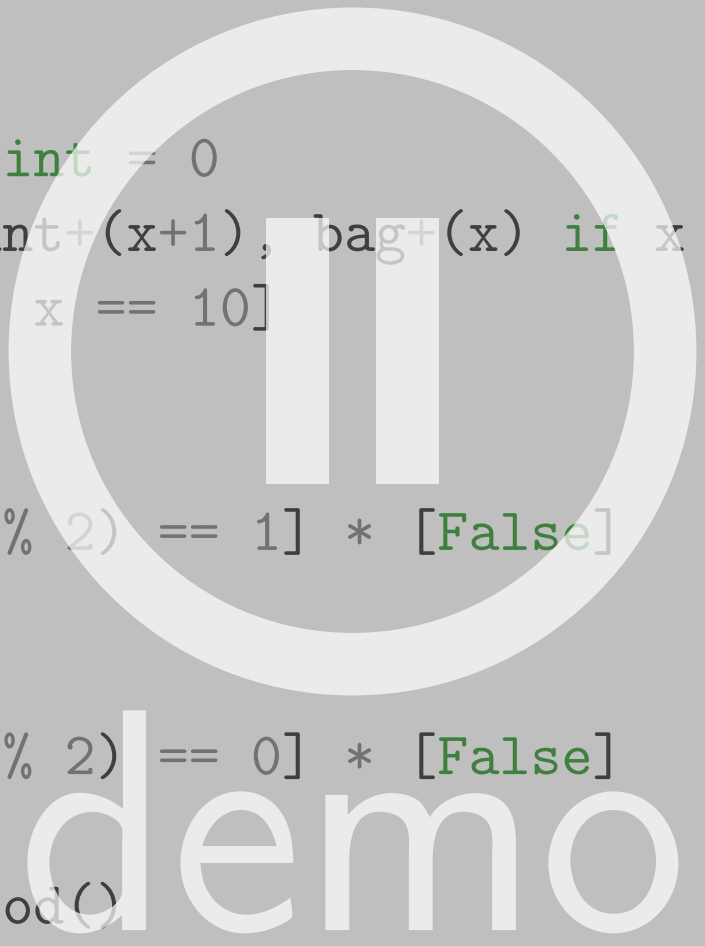
```
buffer bag : int = ()

net prod () :
    buffer count : int = 0
    [count-(x), count+(x+1), bag+(x) if x < 10]
    * [count-(x) if x == 10]

net odd () :
    [bag-(x) if (x % 2) == 1] * [False]

net even () :
    [bag-(x) if (x % 2) == 0] * [False]

odd() | even() | prod()
```



Sub-processes

```
buffer bag : int = ()

net prod () :
    buffer count : int = 0
    [count-(x), count+(x+1), bag+(x) if x < 10]
    * [count-(x) if x == 10]

net odd () :
    [bag-(x) if (x % 2) == 1] * [False]

net even () :
    [bag-(x) if (x % 2) == 0] * [False]

odd() | even() | prod()
```

Parametrised processes

```
buffer fork1 : BlackToken = dot
buffer fork2 : BlackToken = dot
buffer fork3 : BlackToken = dot
buffer fork4 : BlackToken = dot

buffer eating : int = ()

net philo (num, left: buffer, right: buffer):
    ([left-(dot), right-(dot), eating+(num)]
     ; [left+(dot), right+(dot), eating-(num)])
    * [False]

philo(1, fork1, fork2)
| philo(2, fork2, fork3)
| philo(3, fork3, fork4)
| philo(4, fork4, fork1)
```

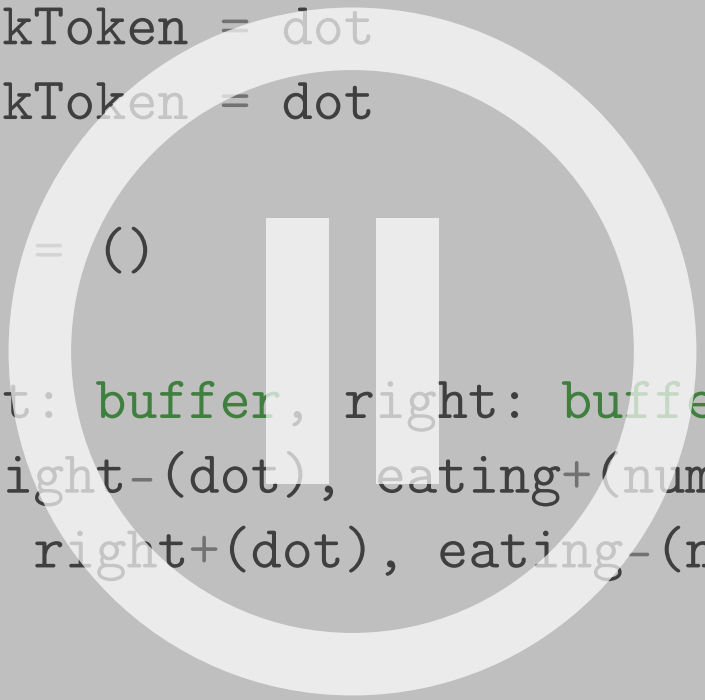
Parametrised processes

```
buffer fork1 : BlackToken = dot
buffer fork2 : BlackToken = dot
buffer fork3 : BlackToken = dot
buffer fork4 : BlackToken = dot

buffer eating : int = ()

net philo (num, left: buffer, right: buffer):
    ([left-(dot), right-(dot), eating+(num)]
     ; [left+(dot), right+(dot), eating-(num)])
    * [False]

philo(1, fork1, fork2)
| philo(2, fork2, fork3)
| philo(3, fork3, fork4)
| philo(4, fork4, fork1)
```



philo.abcd

Parametrised processes

```
buffer fork1 : BlackToken = dot
buffer fork2 : BlackToken = dot
buffer fork3 : BlackToken = dot
buffer fork4 : BlackToken = dot

buffer eating : int = ()

net philo (num, left: buffer, right: buffer):
    ([left-(dot), right-(dot), eating+(num)]
     ; [left+(dot), right+(dot), eating-(num)])
    * [False]

philo(1, fork1, fork2)
| philo(2, fork2, fork3)
| philo(3, fork3, fork4)
| philo(4, fork4, fork1)
```

Outline

A modular framework

ABCD of modelling

- Syntax and intuitive semantics
- Petri nets semantics

More than simulation

Three application domains

Verification issues

Conclusion

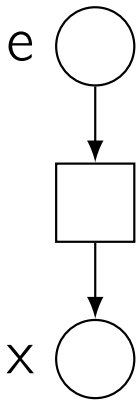
Translating ABCD into Petri nets

```
buffer glob :  int = ()
net Process (num) :
  buffer loc :  int = ()
  [glob+(42)]
  ; [glob-(n), loc+(n) if n < 7]
  ; [loc?(n), glob+(num + n)]
Process(22)
```

Translating ABCD into Petri nets

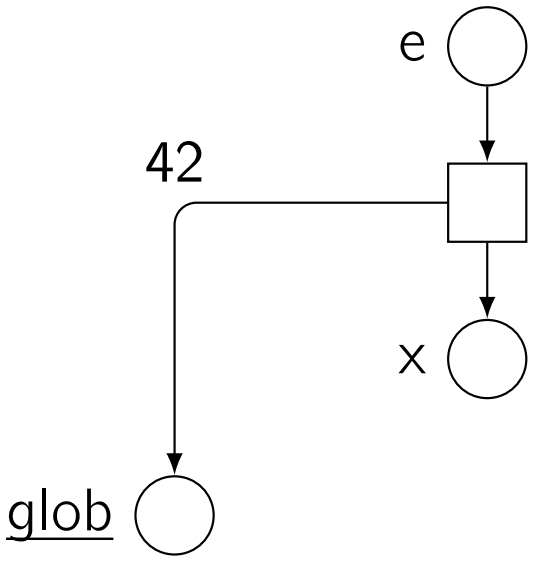
```
buffer glob :  int = ()
net Process (num=22) :
  buffer loc :  int = ()
  [glob+(42)]
  ; [glob-(n), loc+(n) if n < 7]
  ; [loc?(n), glob+(22 + n)]
Process(22)
```

Translating ABCD into Petri nets



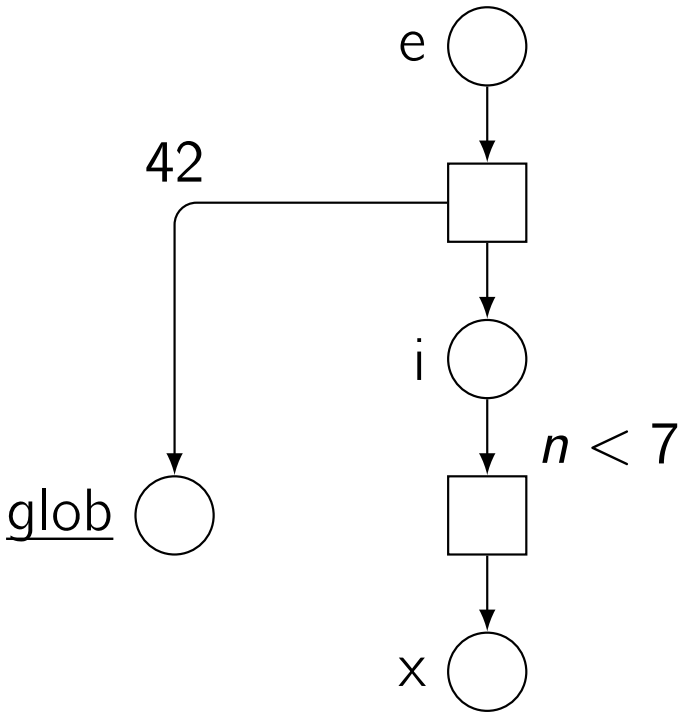

```
buffer glob :  int = ()
net Process (num=22) :
  buffer loc :  int = ()
  [glob+(42)]
  ; [glob-(n), loc+(n) if n < 7]
  ; [loc?(n), glob+(22 + n)]
Process(22)
```

Translating ABCD into Petri nets



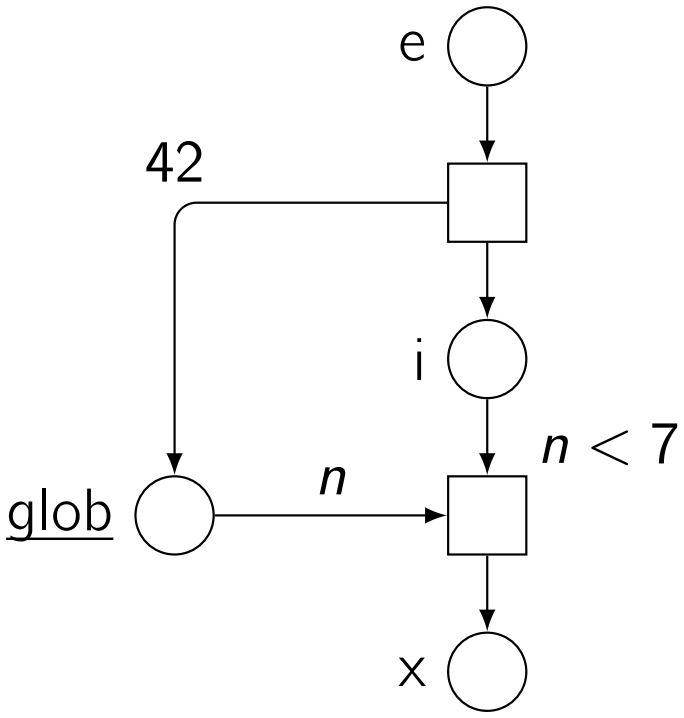

```
buffer glob :  int = ()
net Process (num=22) :
  buffer loc :  int = ()
  [glob+(42)]
  ; [glob-(n), loc+(n) if n < 7]
  ; [loc?(n), glob+(22 + n)]
Process(22)
```

Translating ABCD into Petri nets



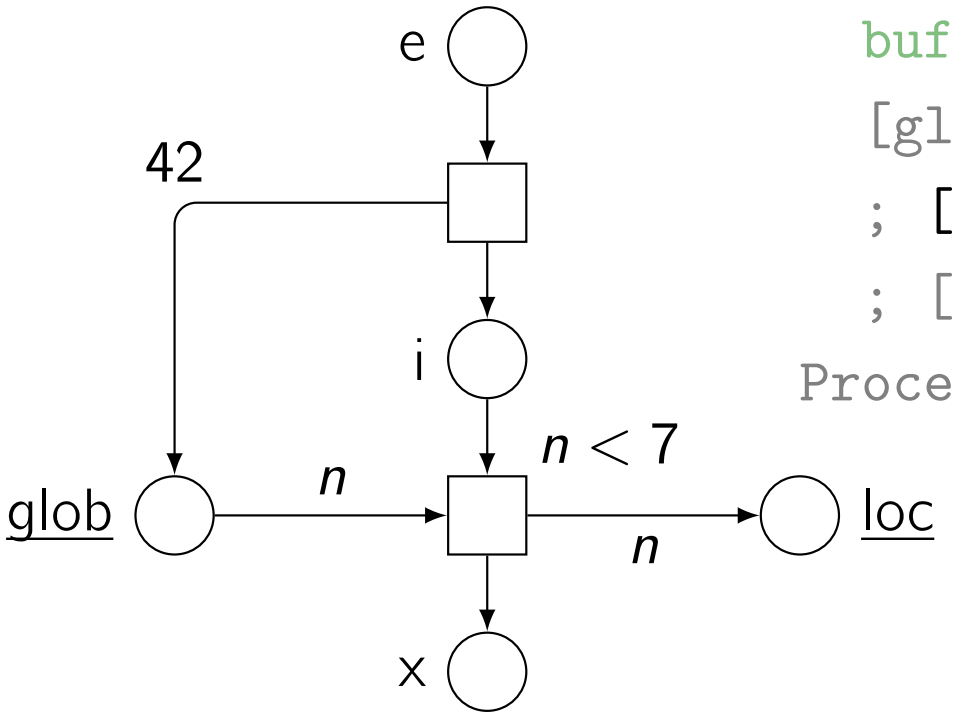

```
buffer glob :  int = ()
net Process (num=22) :
  buffer loc :  int = ()
  [glob+(42)]
  ; [glob-(n), loc+(n) if n < 7]
  ; [loc?(n), glob+(22 + n)]
Process(22)
```

Translating ABCD into Petri nets



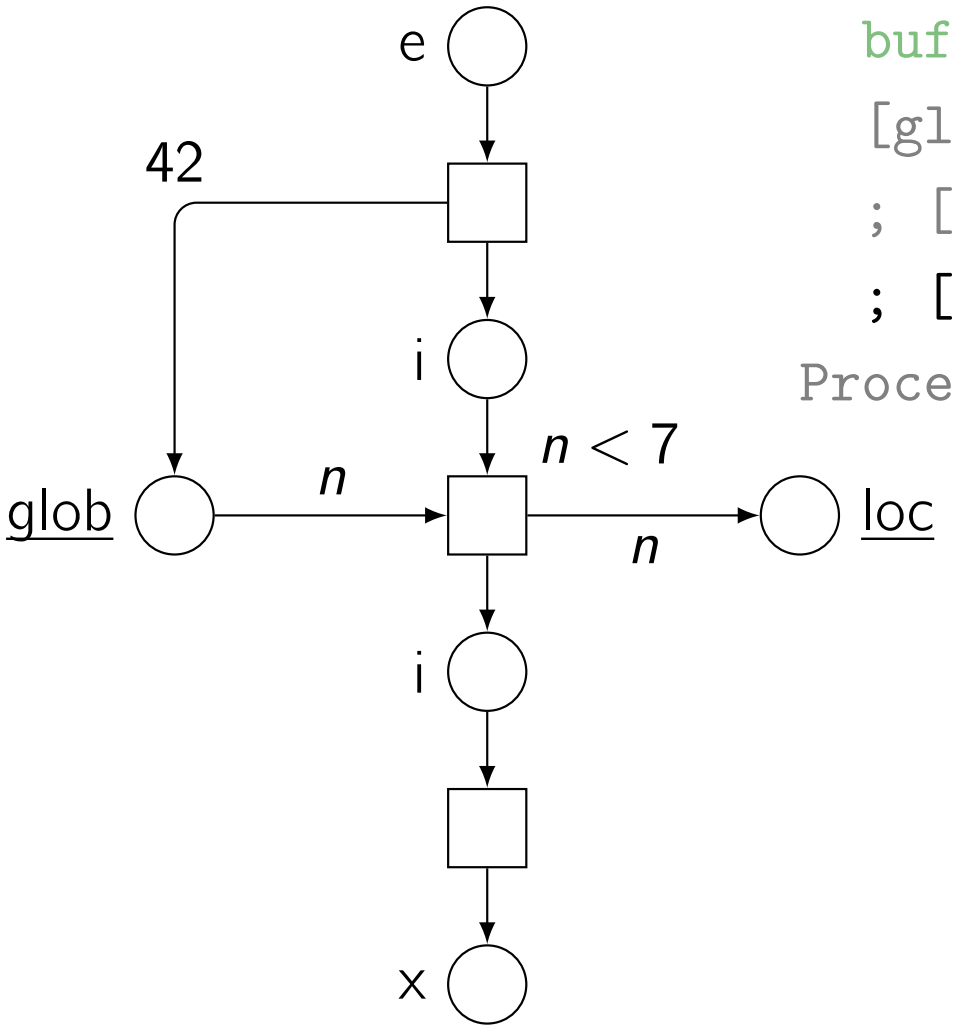

```
buffer glob :  int = ()
net Process (num=22) :
  buffer loc :  int = ()
  [glob+(42)]
  ; [glob-(n), loc+(n) if n < 7]
  ; [loc?(n), glob+(22 + n)]
Process(22)
```

Translating ABCD into Petri nets



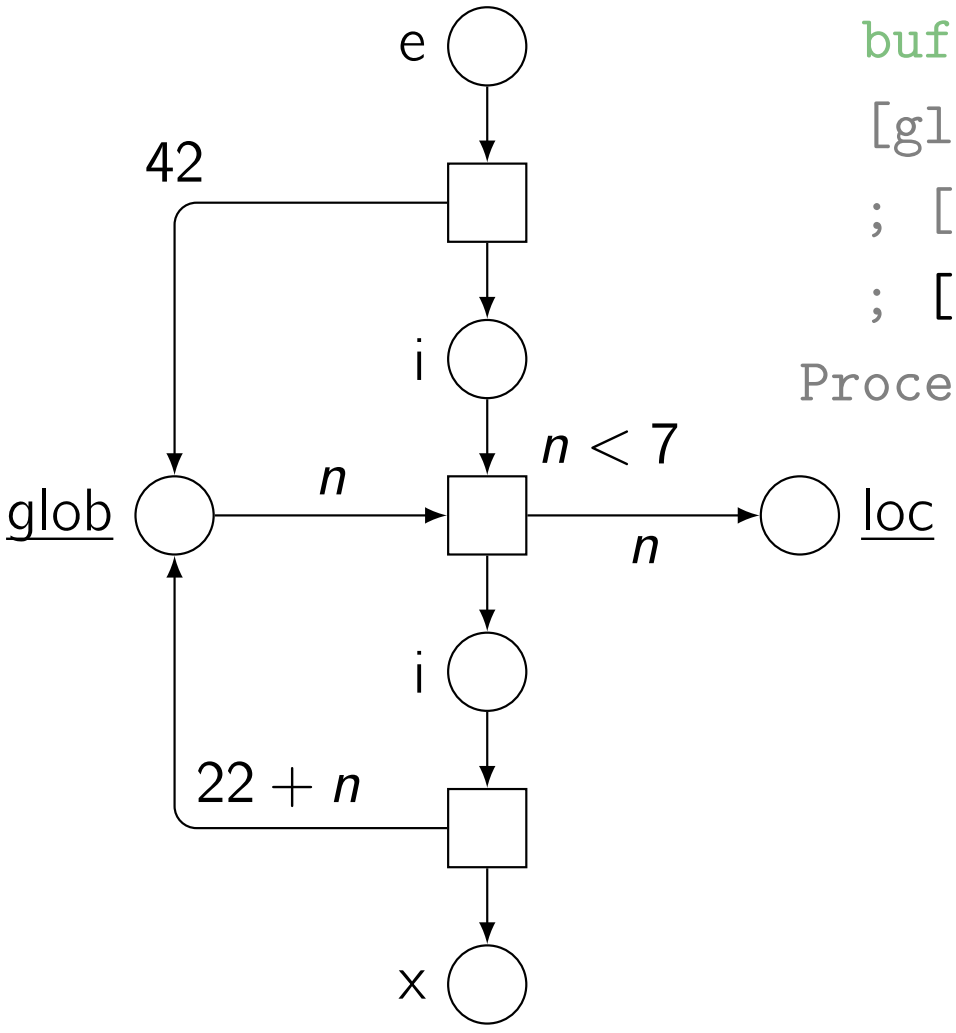

```
buffer glob :  int = ()
net Process (num=22) :
  buffer loc :  int = ()
  [glob+(42)]
  ; [glob-(n), loc+(n) if n < 7]
  ; [loc?(n), glob+(22 + n)]
Process(22)
```

Translating ABCD into Petri nets



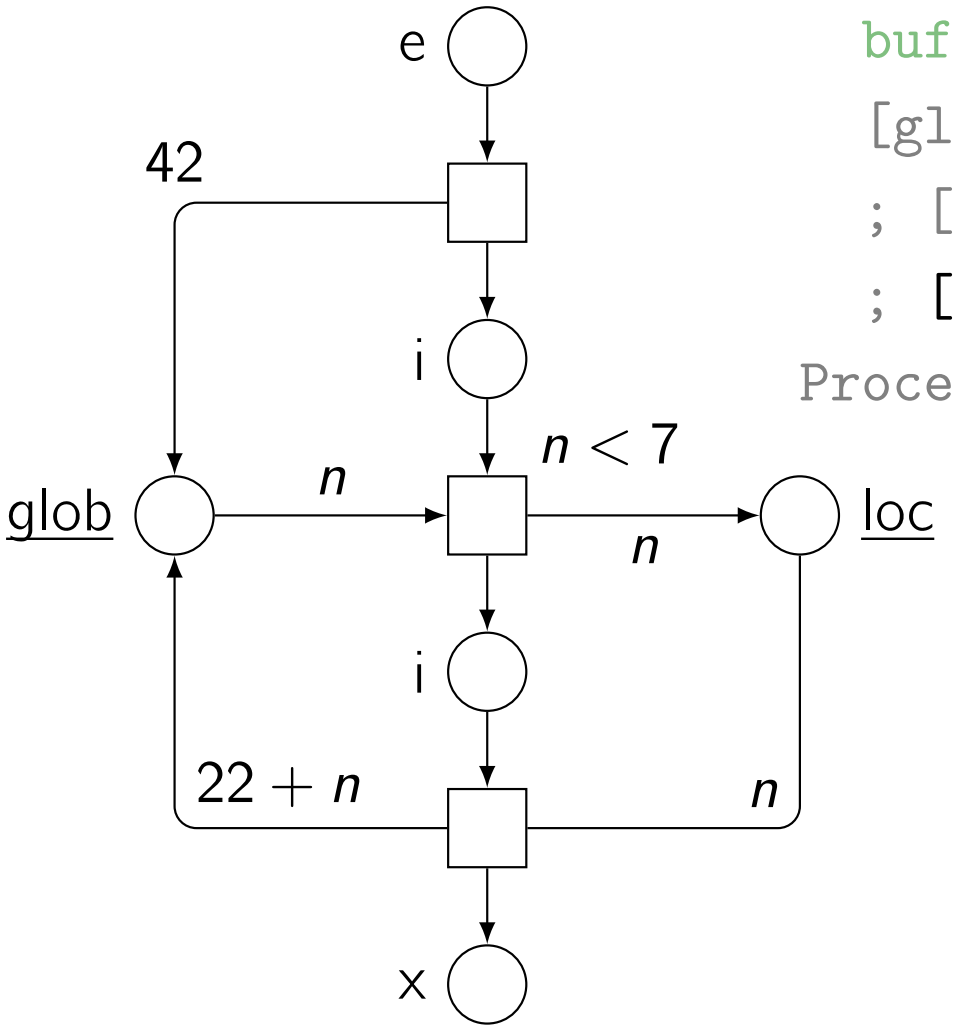

```
buffer glob :  int = ()
net Process (num=22) :
  buffer loc :  int = ()
  [glob+(42)]
  ; [glob-(n), loc+(n) if n < 7]
  ; [loc?(n), glob+(22 + n)]
Process(22)
```

Translating ABCD into Petri nets



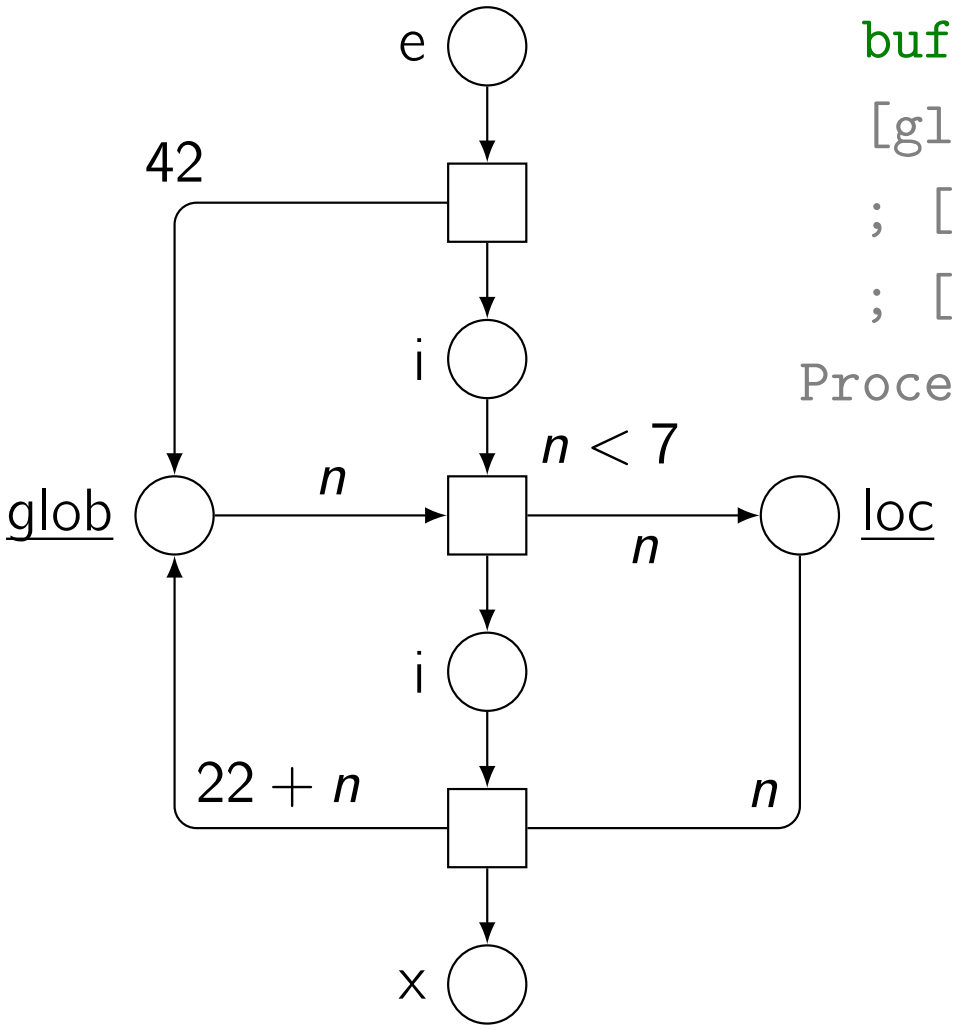

```
buffer glob :  int = ()
net Process (num=22) :
  buffer loc :  int = ()
  [glob+(42)]
  ; [glob-(n), loc+(n) if n < 7]
  ; [loc?(n), glob+(22 + n)]
Process(22)
```

Translating ABCD into Petri nets



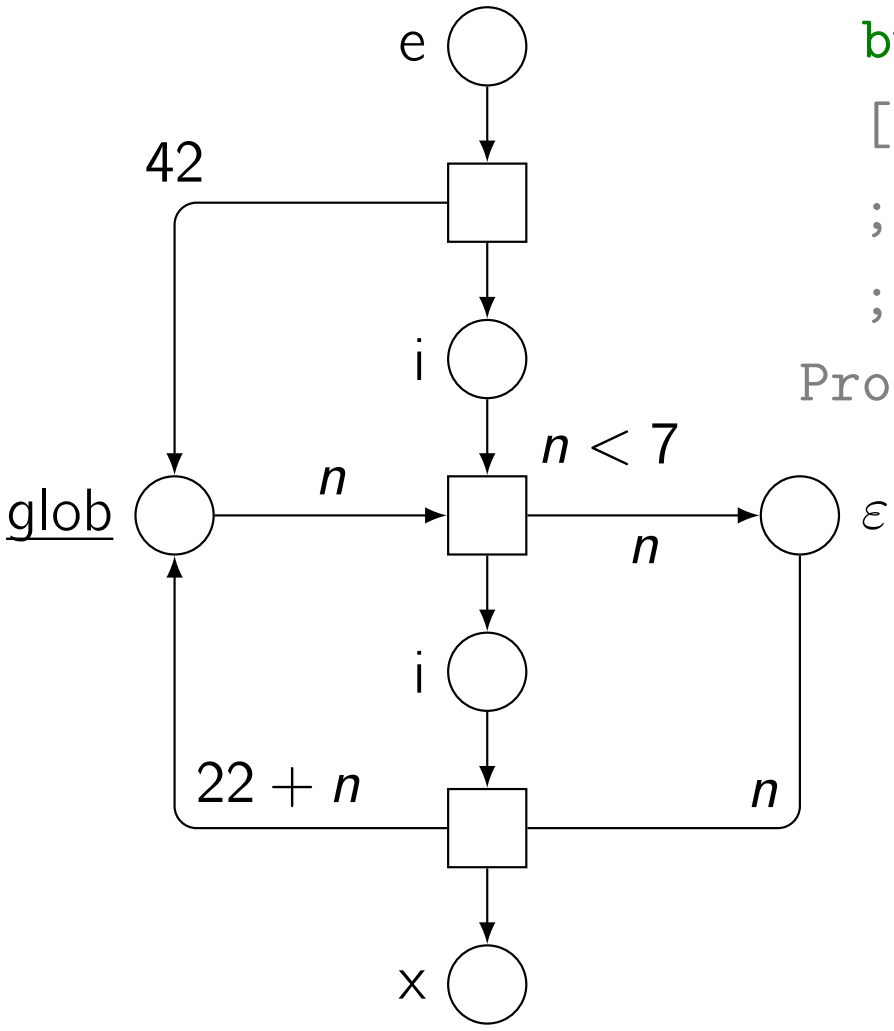

```
buffer glob :  int = ()
net Process (num=22) :
  buffer loc :  int = ()
  [glob+(42)]
  ; [glob-(n), loc+(n) if n < 7]
  ; [loc?(n), glob+(22 + n)]
Process(22)
```

Translating ABCD into Petri nets




```
buffer glob :  int = ()
net Process (num=22) :
  buffer loc :  int = ()
  [glob+(42)]
  ; [glob-(n), loc+(n) if n < 7]
  ; [loc?(n), glob+(22 + n)]
Process(22)
```

Translating ABCD into Petri nets




```
buffer glob :  int = ()
net Process (num=22) :
  buffer loc :  int = ()
  [glob+(42)]
  ; [glob-(n), loc+(n) if n < 7]
  ; [loc?(n), glob+(22 + n)]
Process(22)
```

Outline

A modular framework

ABCD of modelling

More than simulation
- Basic system
- A more sophisticated attempt

Three application domains

Verification issues

Conclusion

Railroad crossing system

- one railway track crossing a road
- a pair of gates
- trains arriving repeatedly

Goal

- specify communication between elements
- ensure safety: gates closed whenever a train is present
- ensure liveness: don't keep gates closes forever

Outline

A modular framework

ABCD of modelling

More than simulation

- Basic system
  - A more sophisticated attempt

Three application domains

Verification issues

Conclusion

Communication and gates

```
symbol UP, DOWN, OPEN, MOVING, CLOSED

buffer command : enum(UP, DOWN) = ()

net gate () :
    buffer state : enum(OPEN, MOVING, CLOSED) = OPEN
    ([command-(DOWN), state<>(OPEN=MOVING)]
     ; [state<>(MOVING=CLOSED)]
     ; [command-(UP), state<>(CLOSED=MOVING)]
     ; [state<>(MOVING=OPEN)])
    * [False]
```

Tracks and main process

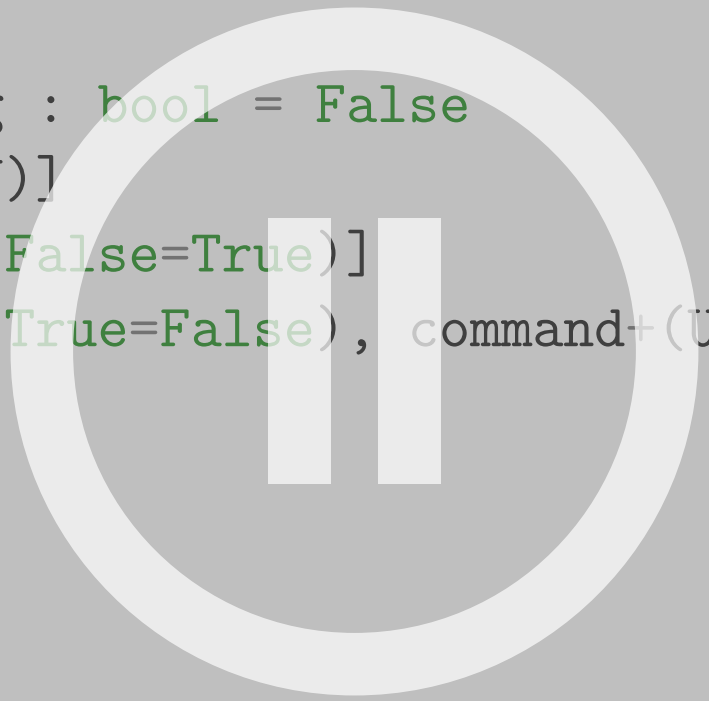
```
net track () :
    buffer crossing : bool = False
    ([command+(DOWN)]
     ; [crossing<>(False=True)]
     ; [crossing<>(True=False), command+(UP)])
    * [False]

gate() | track()
```

Tracks and main process

```
net track () :
    buffer crossing : bool = False
    ([command+(DOWN)]
     ; [crossing<>(False=True)]
     ; [crossing<>(True=False), command+(UP)])
    * [False]

gate() | track()
```



Tracks and main process

```
net track () :
    buffer crossing : bool = False
    ([command+(DOWN)]
     ; [crossing<>(False=True)]
     ; [crossing<>(True=False), command+(UP)])
    * [False]

gate() | track()
```

Tracks and main process

```
net track () :
    buffer crossing : bool = False
    ([command+(DOWN)]
     ; [crossing<>(False=True)]
     ; [crossing<>(True=False), command+(UP)])
    * [False]

gate() | track()
assert (True not in _["track().crossing"]
        or CLOSED in _["gate().state"])
```

Tracks and main process

```
net track () :
    buffer crossing : bool = False
    ([command+(DOWN)]
     ; [crossing<>(False=True)]
     ; [crossing<>(True=False), command+(UP)])
    * [False]

gate() | track()
assert (True not in _["track().crossing"]
        or CLOSED in _["gate().state"])
```

Tracks and main process

```
net track () :
    buffer crossing : bool = False
    ([command+(DOWN)]
     ; [crossing<>(False=True)]
     ; [crossing<>(True=False), command+(UP)])
    * [False]

gate() | track()
assert (True not in _["track().crossing"]
        or CLOSED in _["gate().state"])
```

Outline

A modular framework

ABCD of modelling

More than simulation

- Basic system
- A more sophisticated attempt

Three application domains

Verification issues

Conclusion

Adding a green light

```
buffer light : enum(RED, GREEN) = GREEN

net gate () :
    buffer state : enum(OPEN, MOVING, CLOSED) = OPEN
    ([command-(DOWN), state<>(OPEN=MOVING)] ;
     [state<>(MOVING=CLOSED), light<>(RED=GREEN)] ;
     [command-(UP), state<>(CLOSED=MOVING)] ;
     [state<>(MOVING=OPEN)])
    * [False]

net track () :
    buffer crossing : bool = False
    ([command+(DOWN), light<>(GREEN=RED)] ;
     [crossing<>(False=True), light?(GREEN)] ;
     [crossing<>(True=False), command+(UP)])
    * [False]
```

Adding a green light

```
buffer light : enum(RED, GREEN) = GREEN

net gate () :
    buffer state : enum(OPEN, MOVING, CLOSED) = OPEN
    ([command-(DOWN), state<>(OPEN=MOVING)] ;
     [state<>(MOVING=CLOSED), light<>(RED=GREEN)] ;
     [command-(UP), state<>(CLOSED=MOVING)] ;
     [state<>(MOVING=OPEN)])
    * [False]

net track () :
    buffer crossing : bool = False
    ([command+(DOWN), light<>(GREEN=RED)] ;
     [crossing<>(False=True), light?(GREEN)] ;
     [crossing<>(True=False), command+(UP)])
    * [False]
```

Adding a green light

```
buffer light : enum(RED, GREEN) = GREEN

net gate () :
    buffer state : enum(OPEN, MOVING, CLOSED) = OPEN
    ([command-(DOWN), state<>(OPEN=MOVING)] ;
     [state<>(MOVING=CLOSED), light<>(RED=GREEN)] ;
     [command-(UP), state<>(CLOSED=MOVING)] ;
     [state<>(MOVING=OPEN)])
    * [False]

net track () :
    buffer crossing : bool = False
    ([command+(DOWN), light<>(GREEN=RED)] ;
     [crossing<>(False=True), light?(GREEN)] ;
     [crossing<>(True=False), command+(UP)])
    * [False]
```

Outline

A modular framework

ABCD of modelling

More than simulation

Three application domains
- Timed systems
- Security protocols
- Biological regulatory networks

Verification issues

Conclusion

Outline

A modular framework

ABCD of modelling

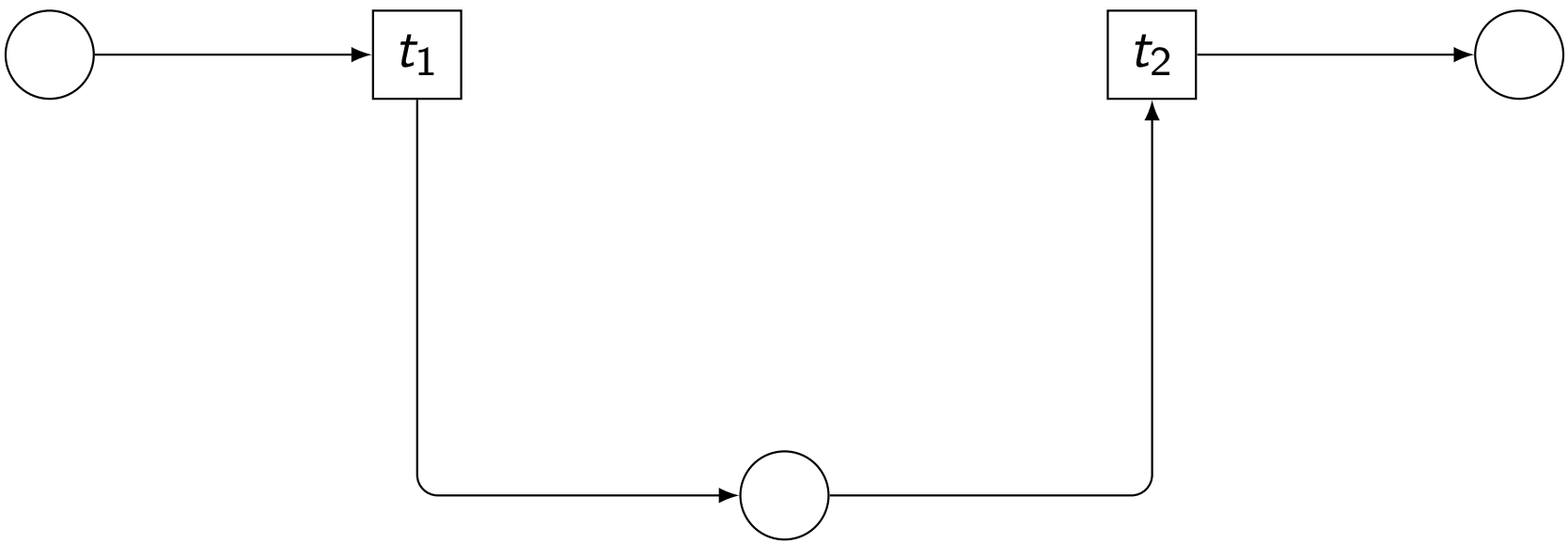
More than simulation

Three application domains
- Timed systems
- Security protocols
- Biological regulatory networks

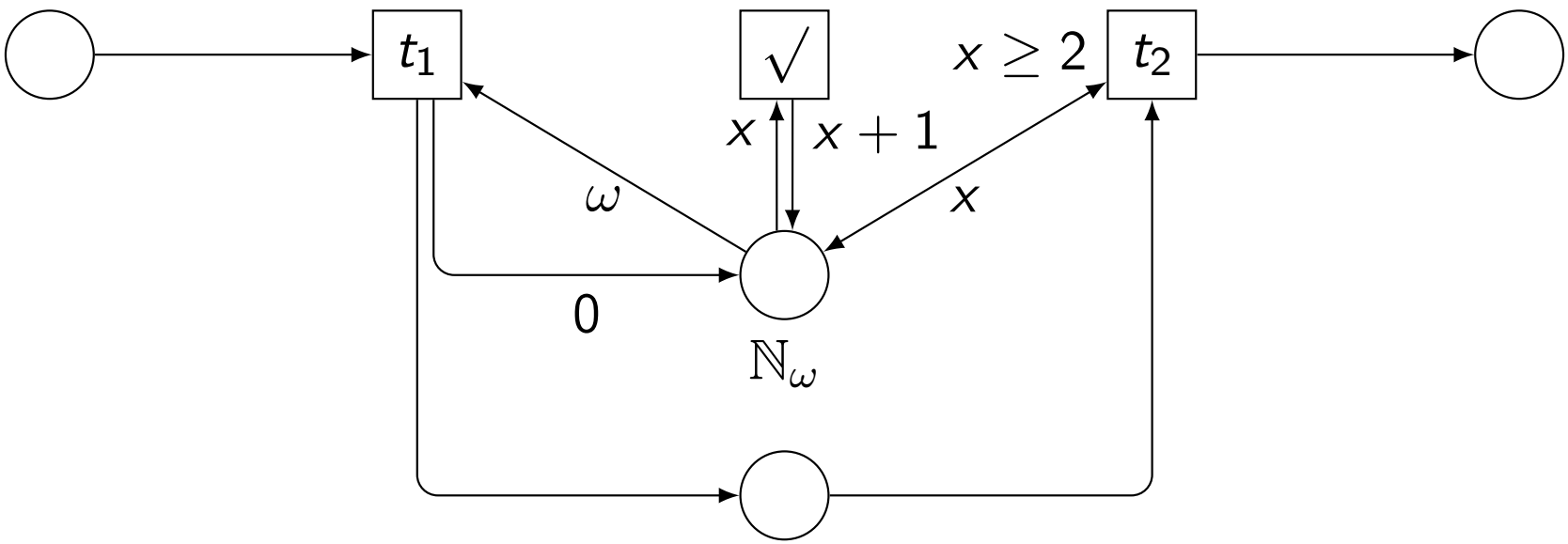
Verification issues

Conclusion

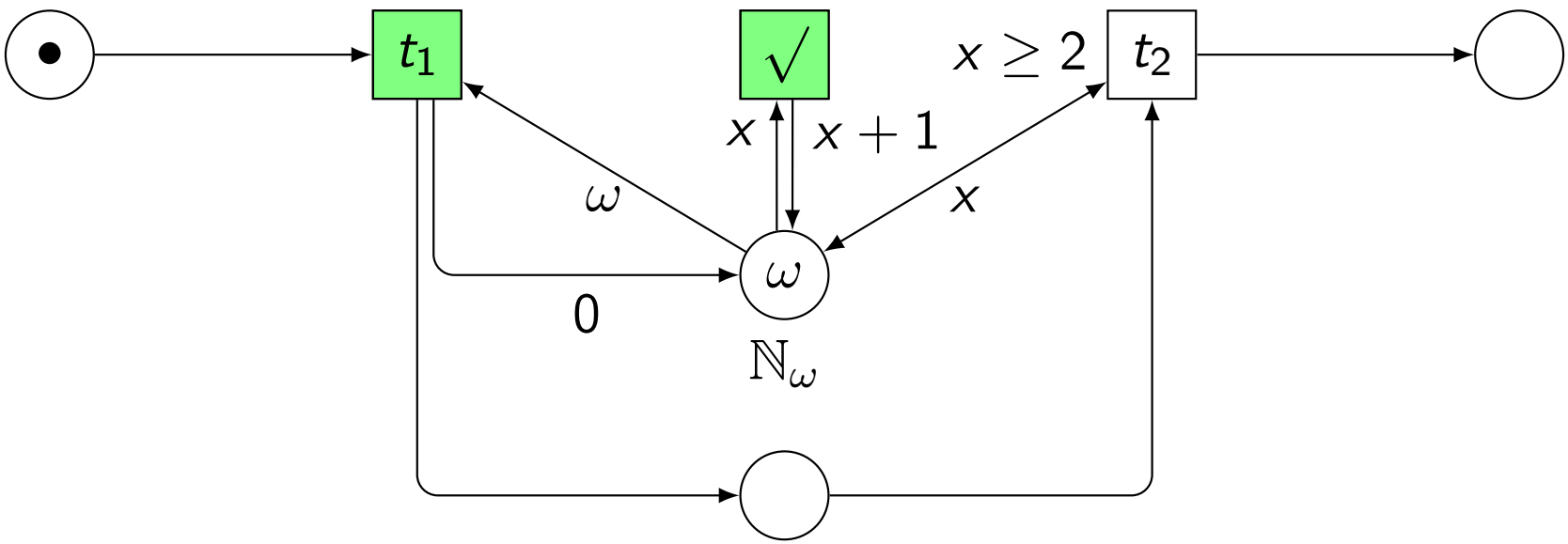
Measuring time and enforcing deadlines



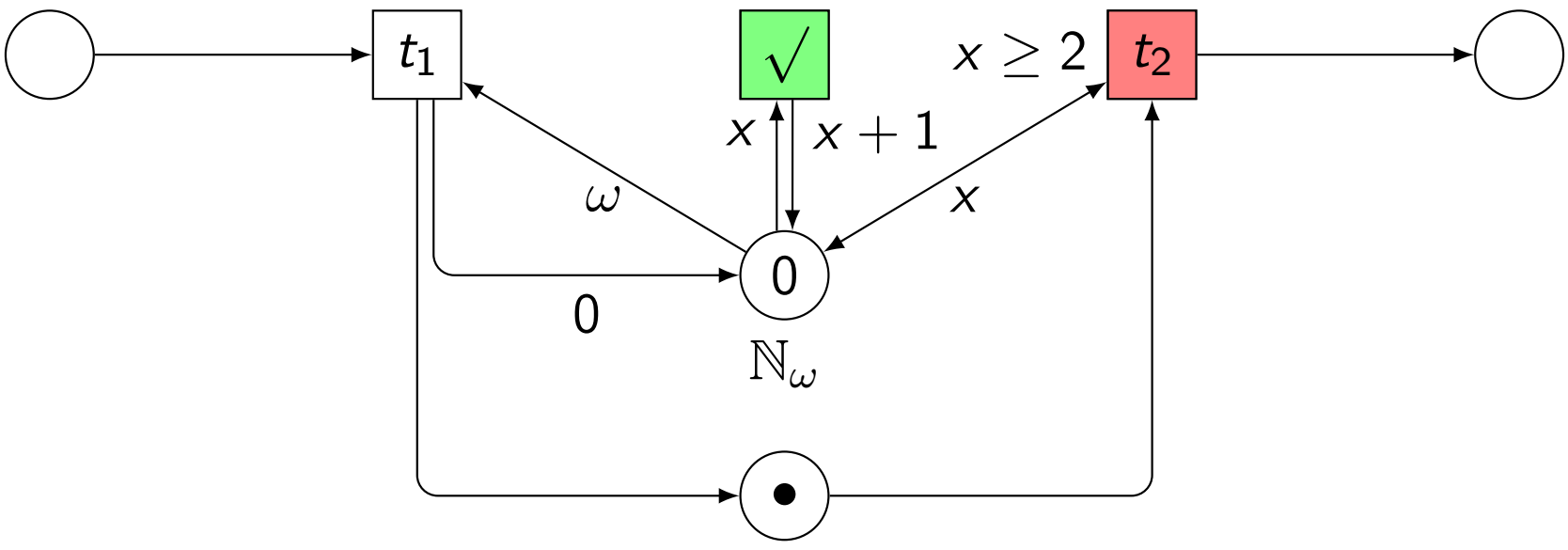
Measuring time and enforcing deadlines



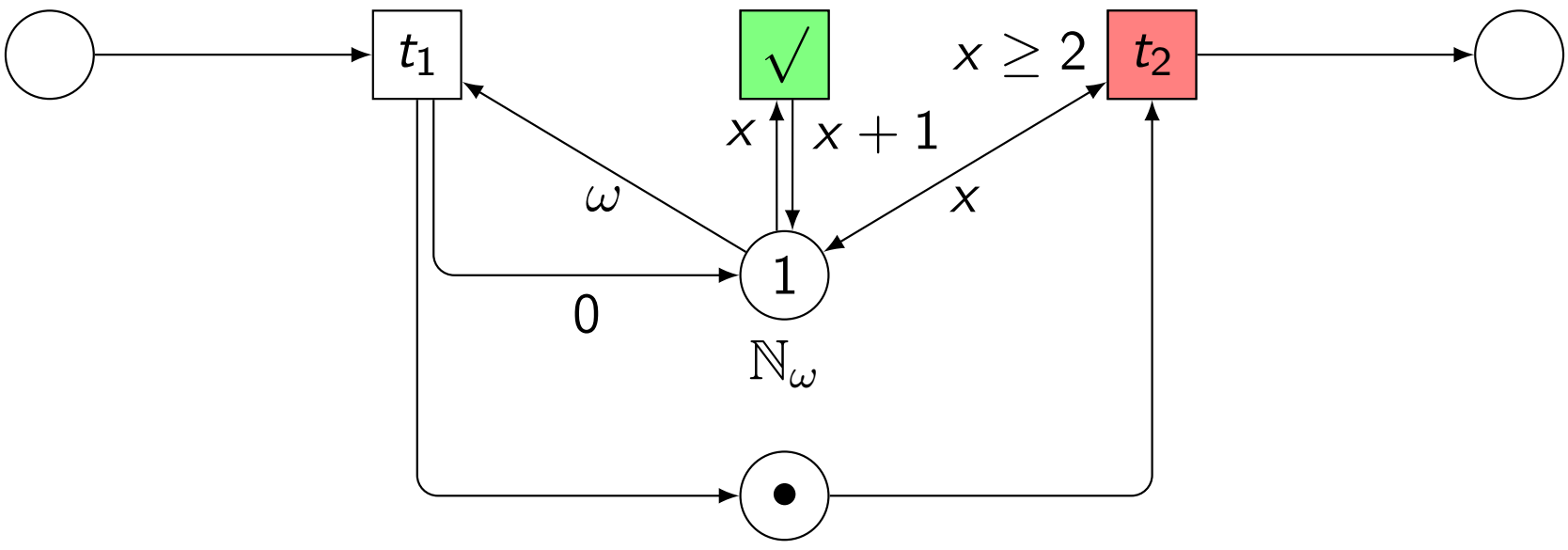
Measuring time and enforcing deadlines



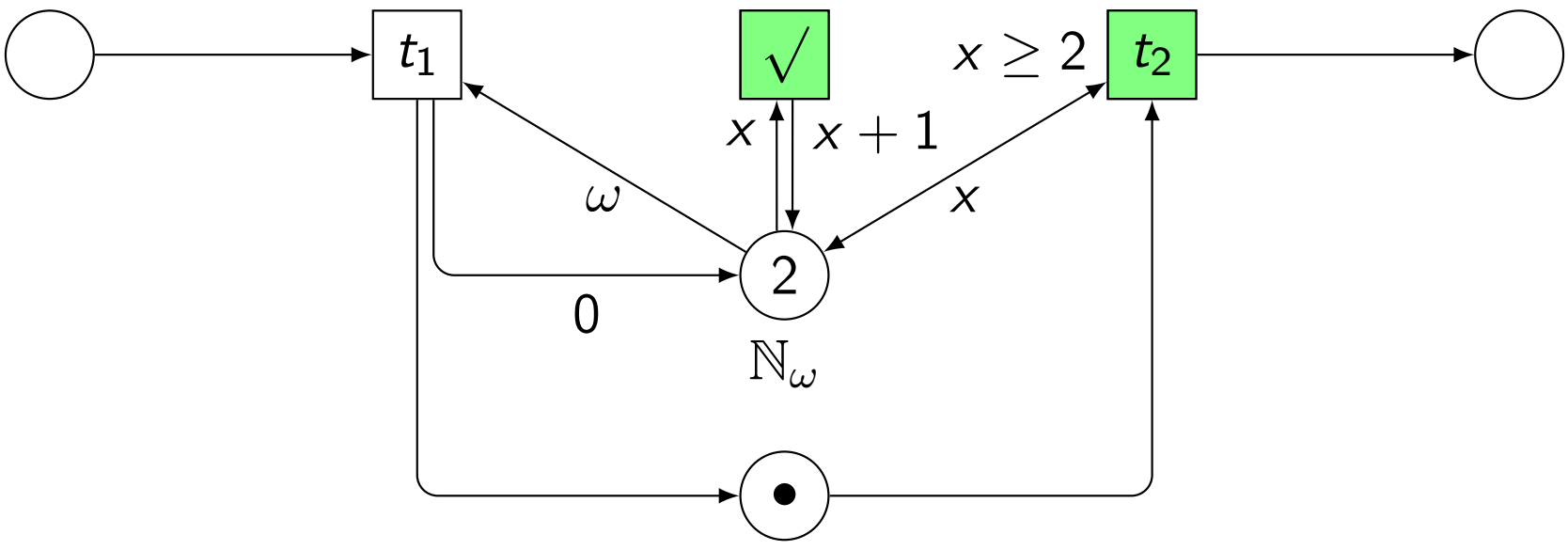
Measuring time and enforcing deadlines



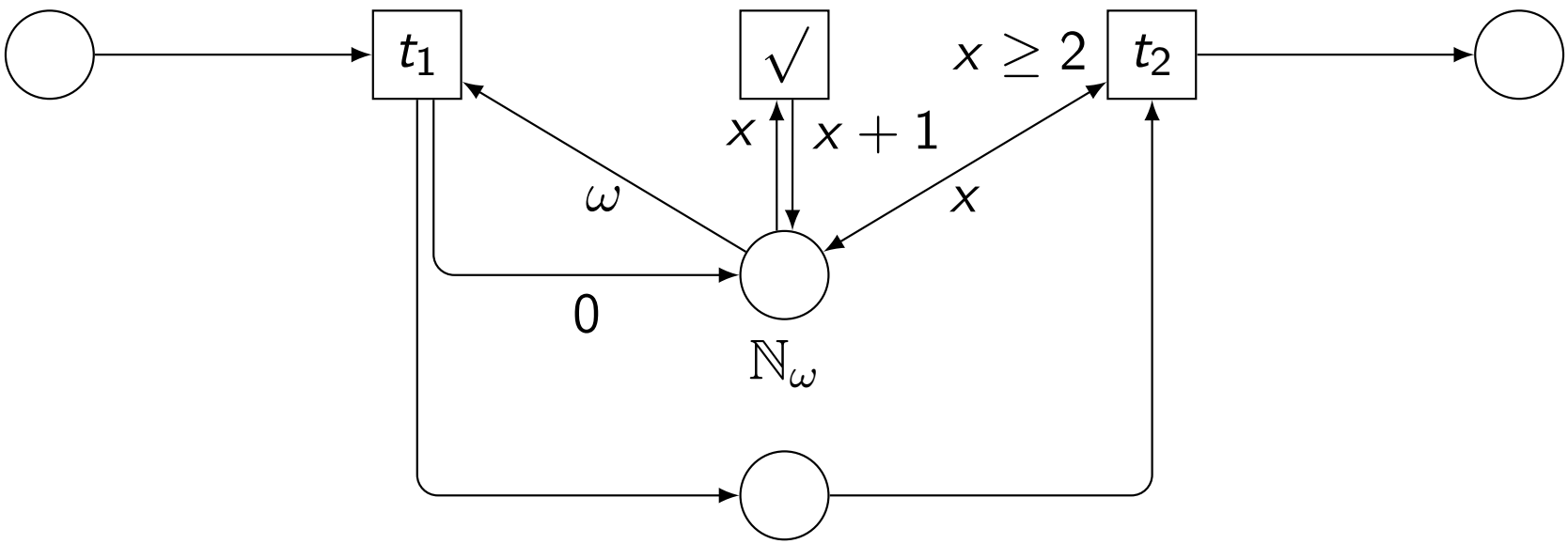
Measuring time and enforcing deadlines



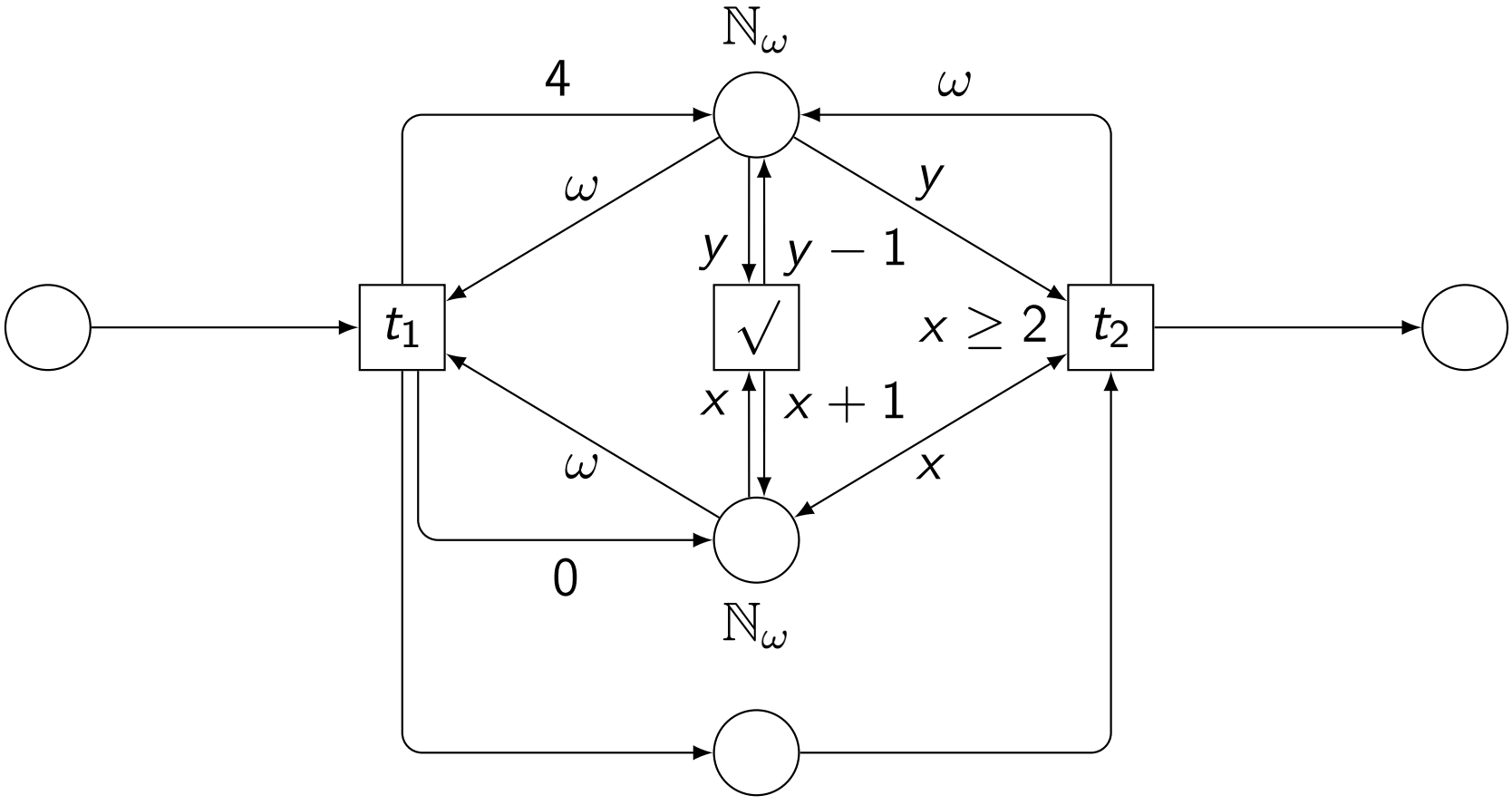
Measuring time and enforcing deadlines



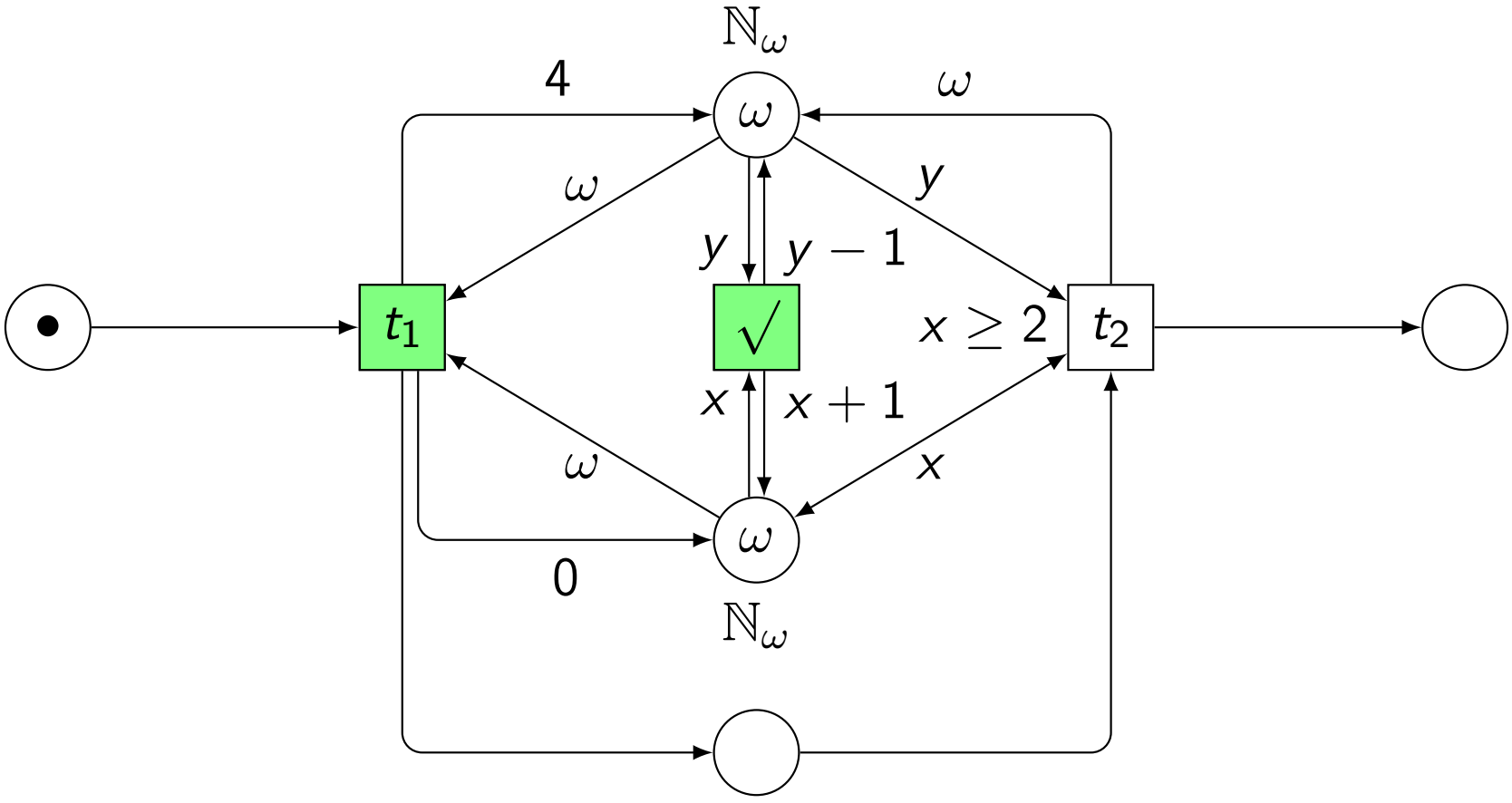
Measuring time and enforcing deadlines



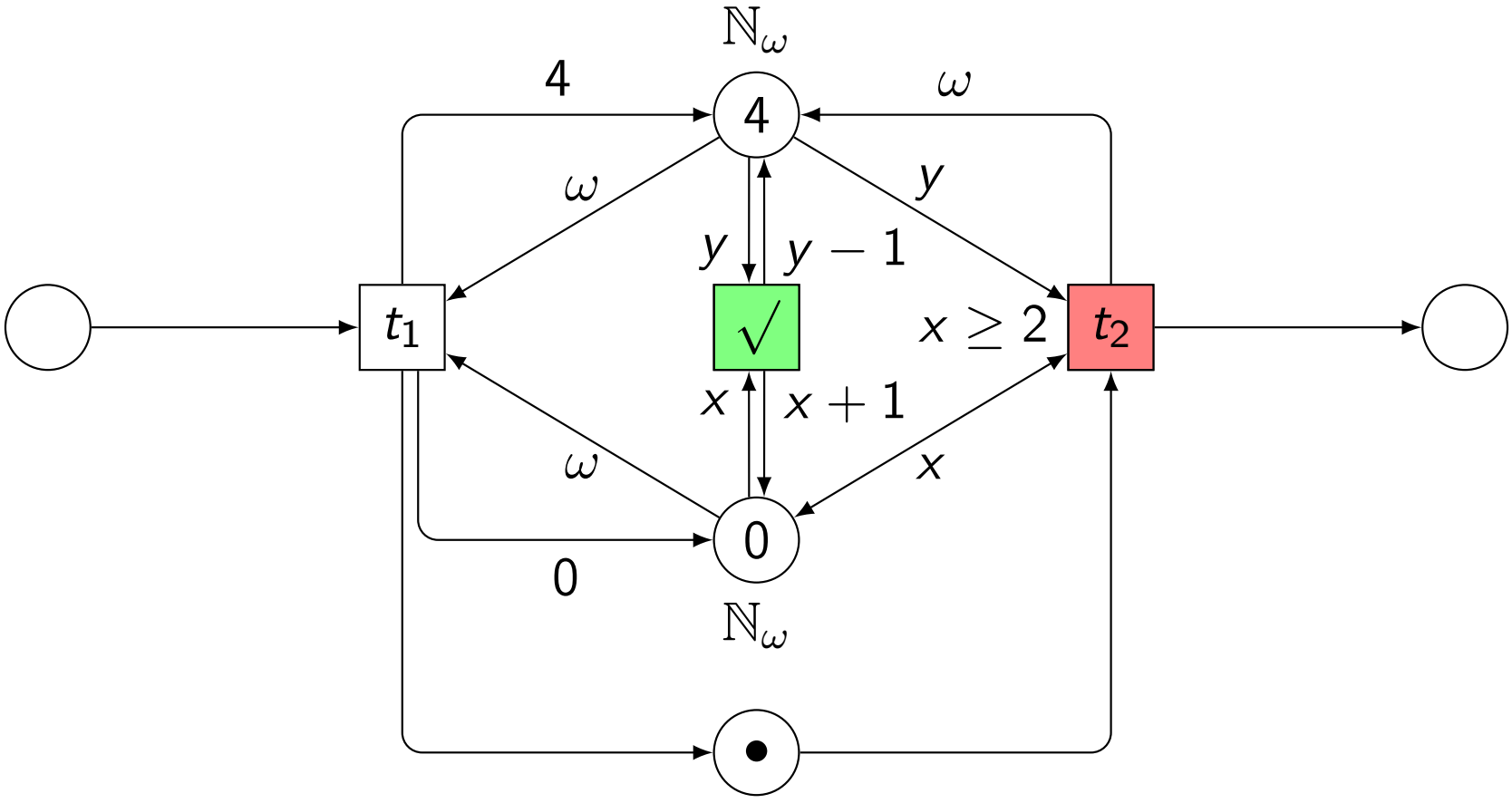
Measuring time and enforcing deadlines



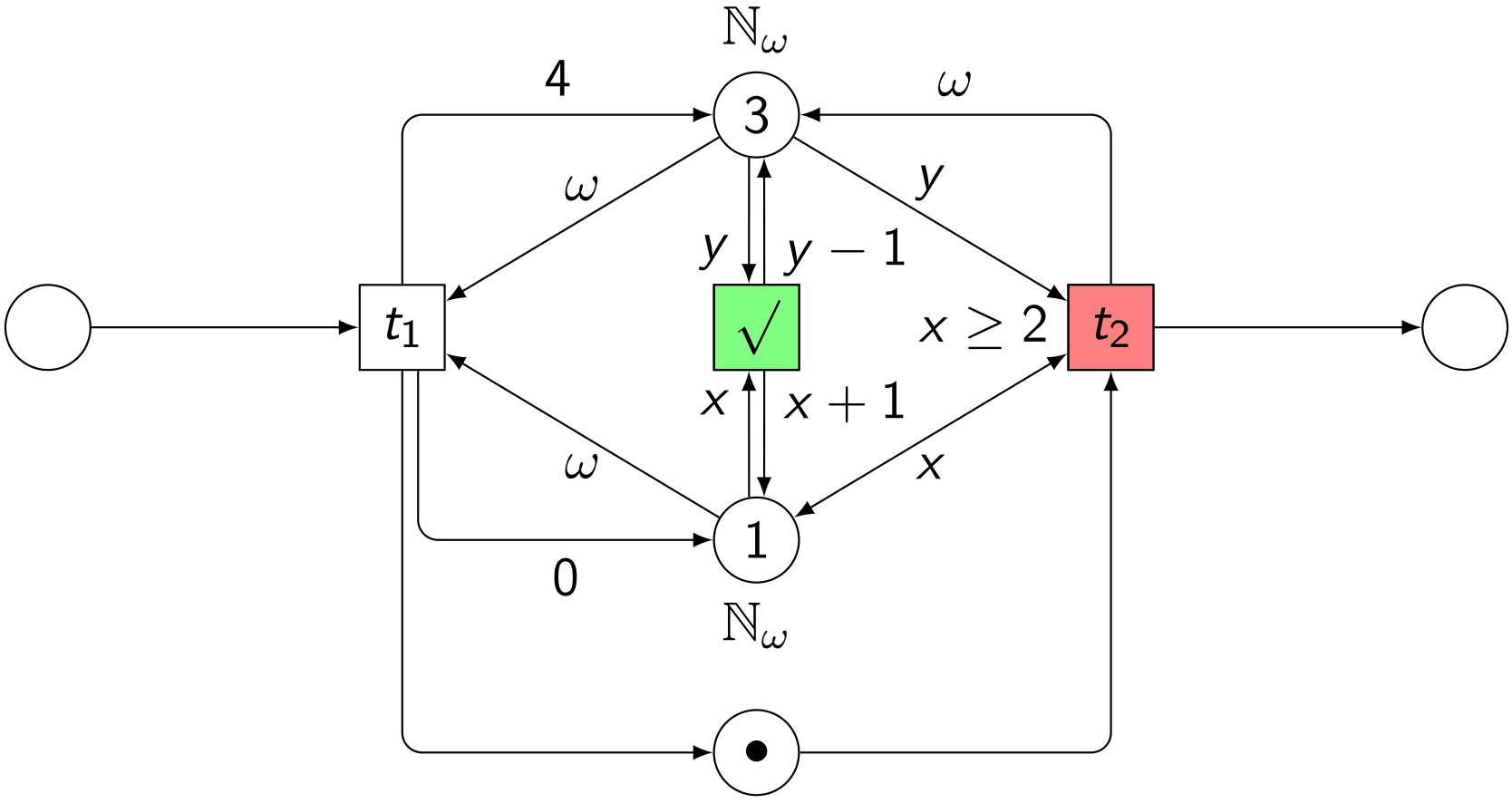
Measuring time and enforcing deadlines



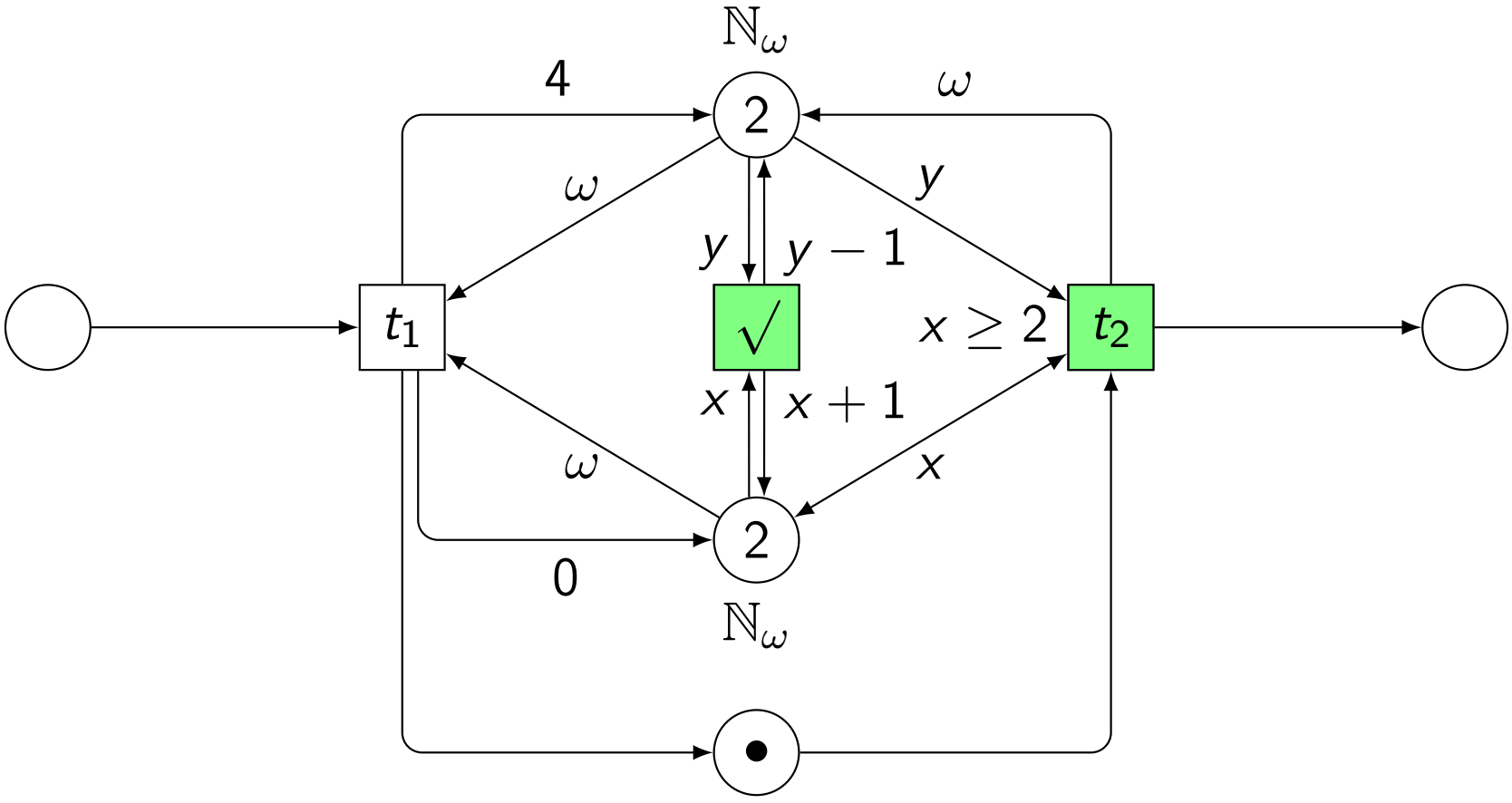
Measuring time and enforcing deadlines



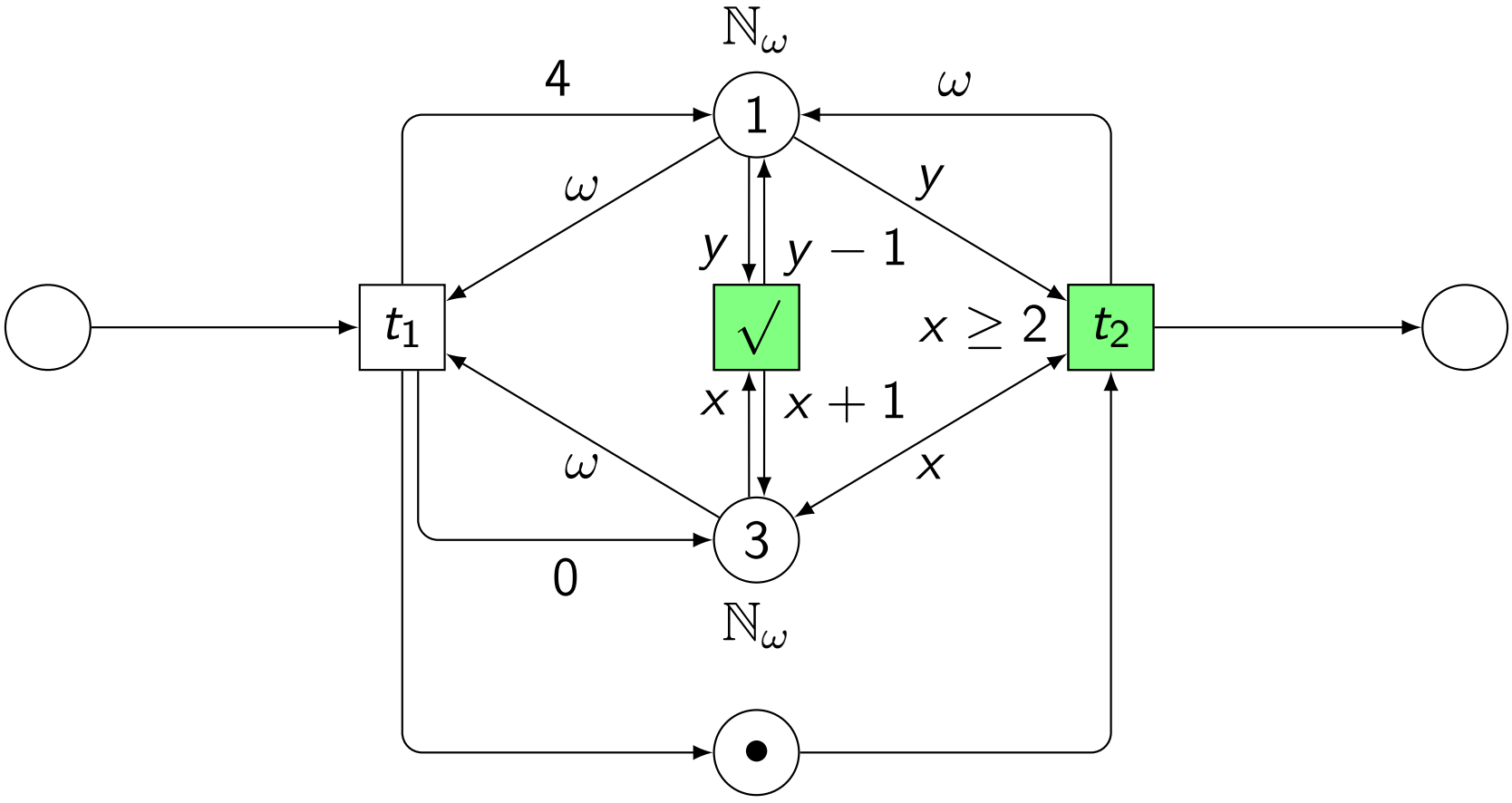
Measuring time and enforcing deadlines



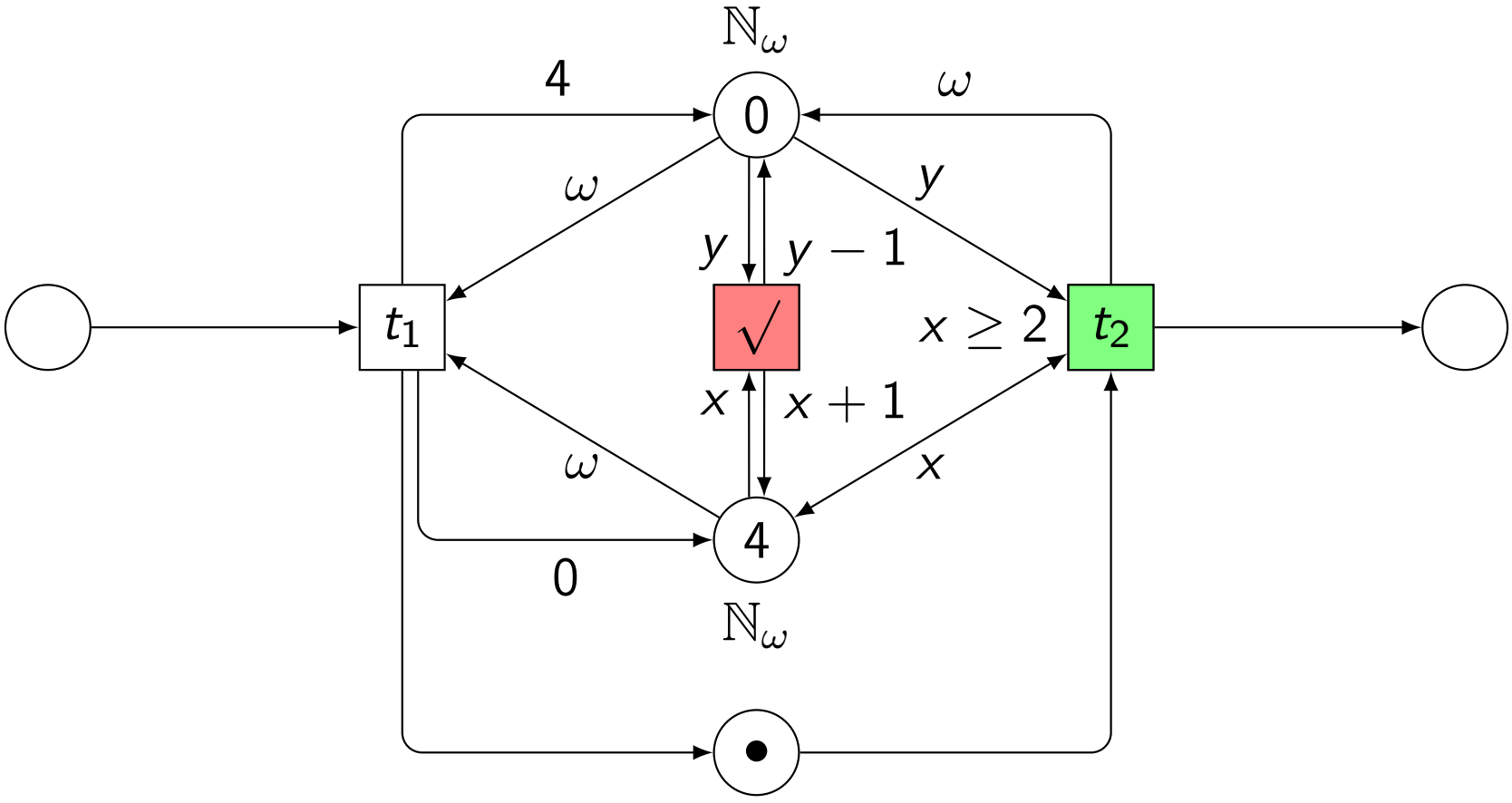
Measuring time and enforcing deadlines



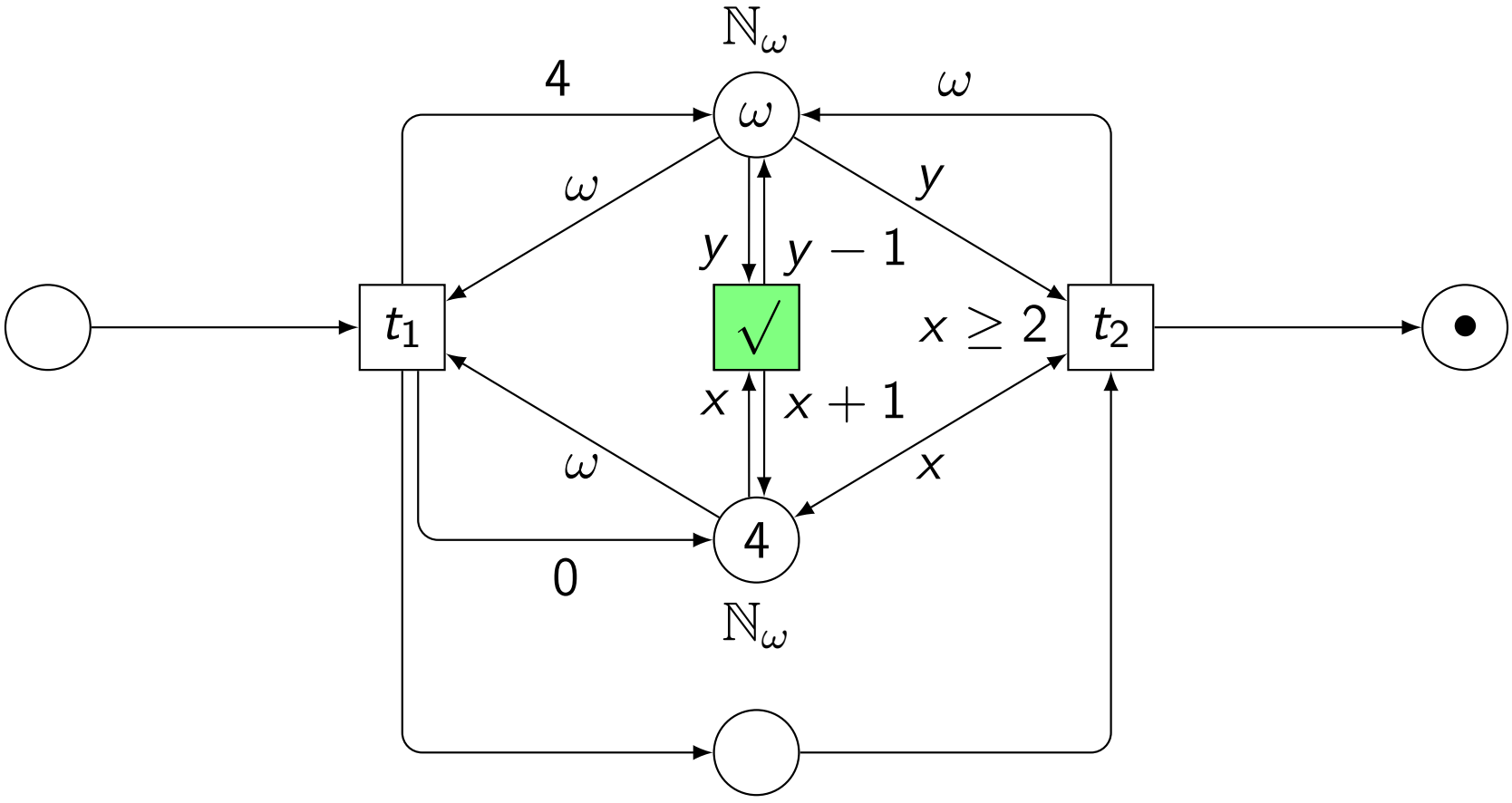
Measuring time and enforcing deadlines



Measuring time and enforcing deadlines



Measuring time and enforcing deadlines



Causal time properties

- usable in practical modelling
- may be more efficient than timed-automata
- several ticks are possible (more or less synchronised)
- can be used through a syntactical layer
- executable in real-time (under reasonable assumptions)
- introduces extra combinatorial explosion (see below)

Outline

A modular framework

ABCD of modelling

More than simulation

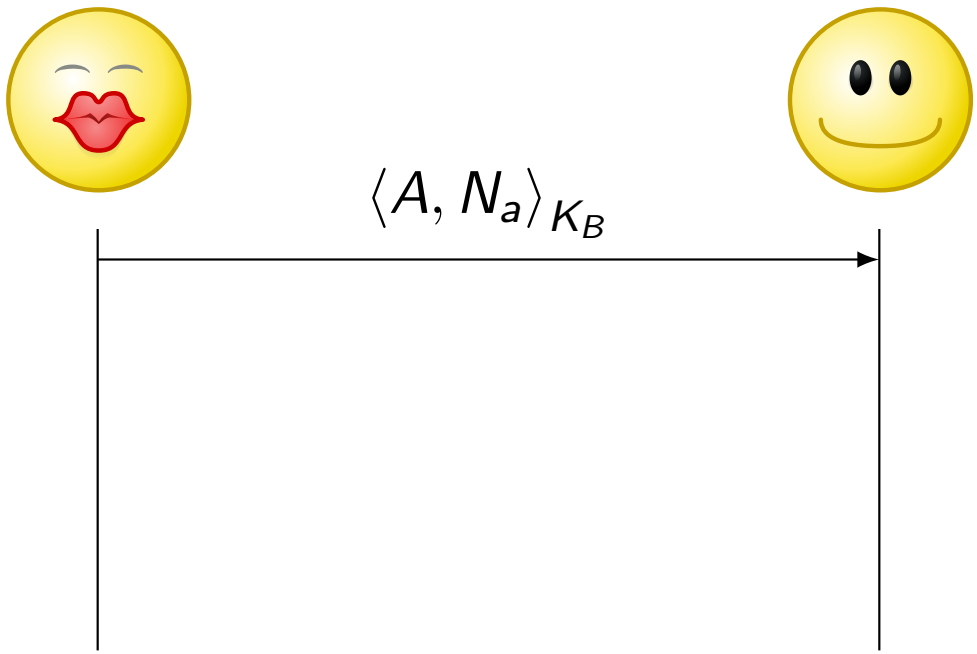
Three application domains
- Timed systems
- Security protocols
- Biological regulatory networks

Verification issues

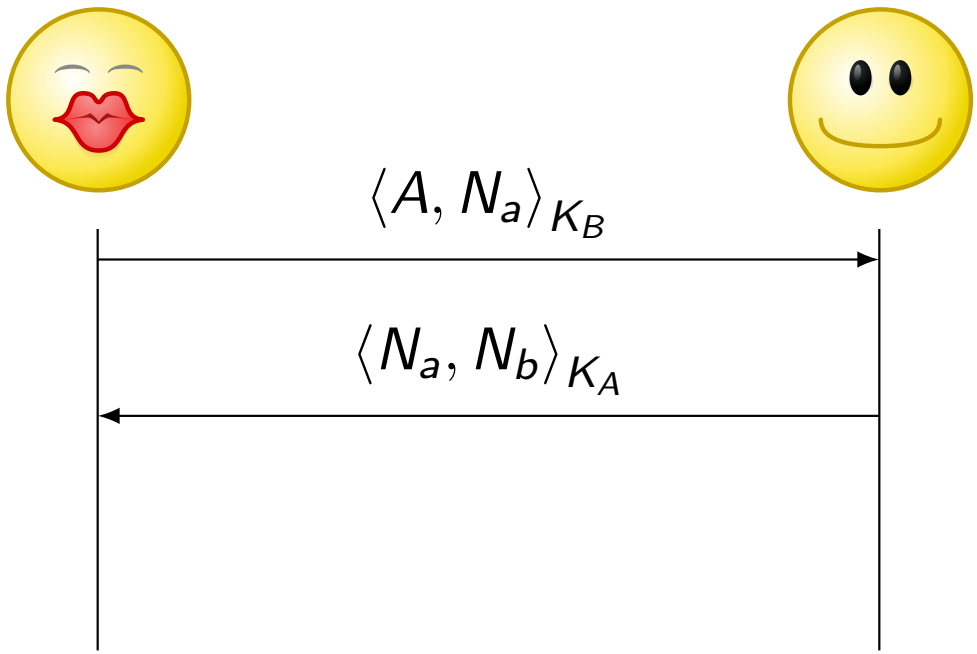
Conclusion

Needham-Schroeder public key protocol (1978)

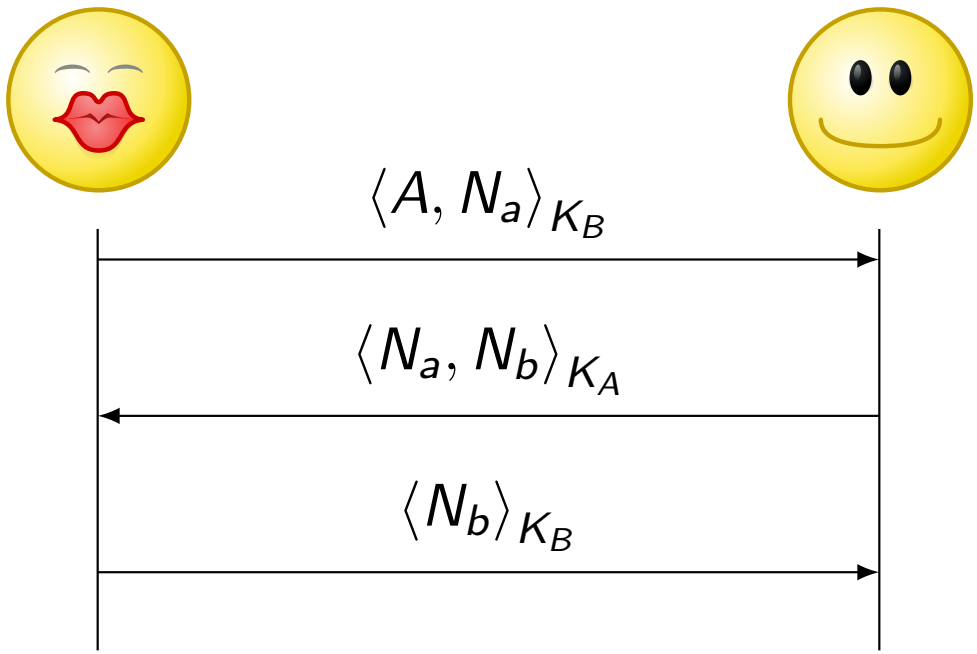
Needham-Schroeder public key protocol (1978)



Needham-Schroeder public key protocol (1978)

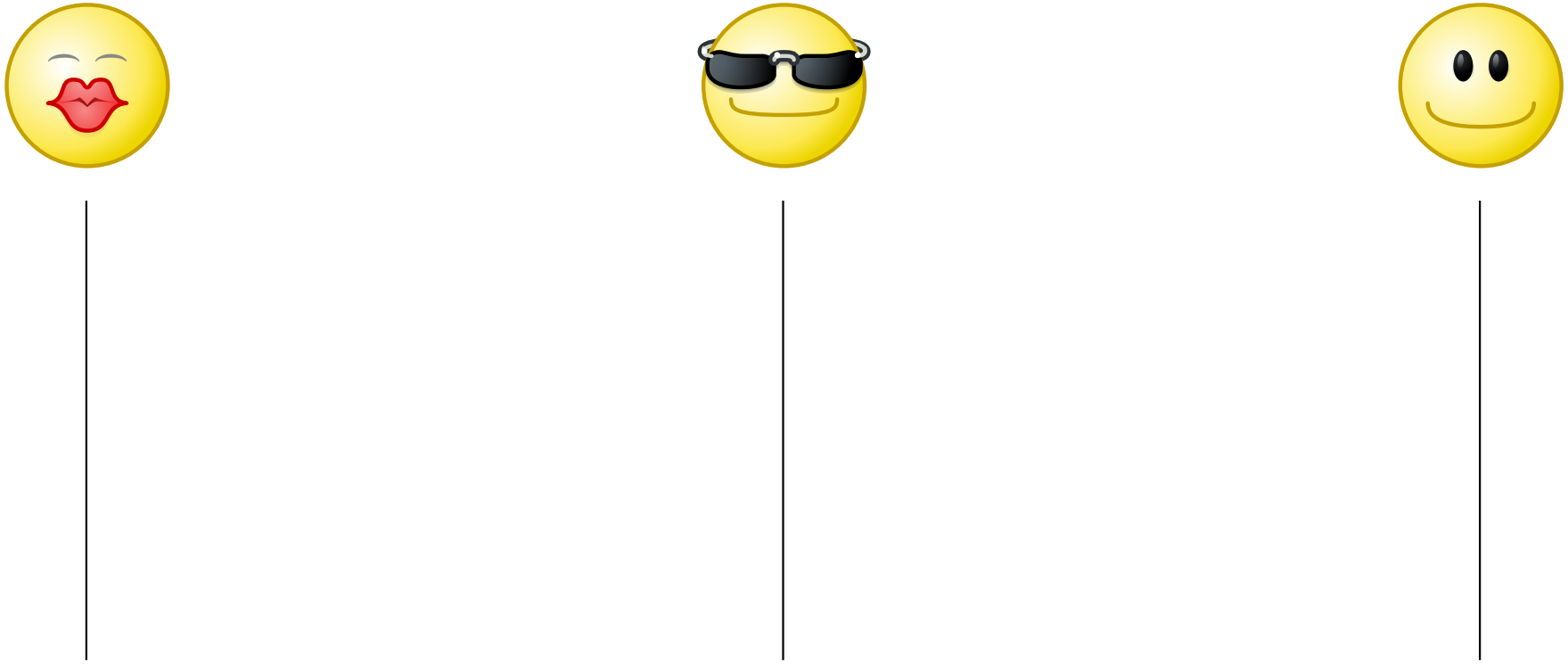


Needham-Schroeder public key protocol (1978)



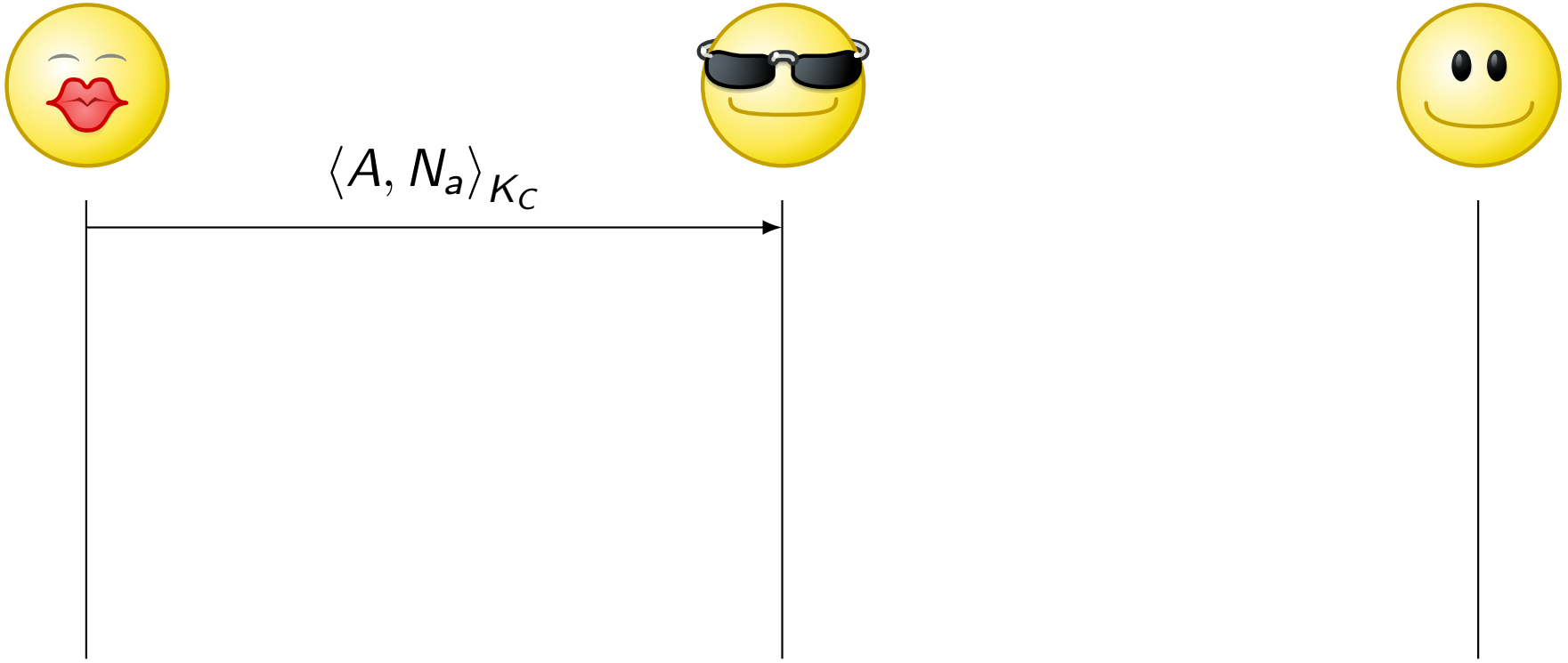
Attack discovered 17 years later!

Lowe (1995), using a dedicated process algebra



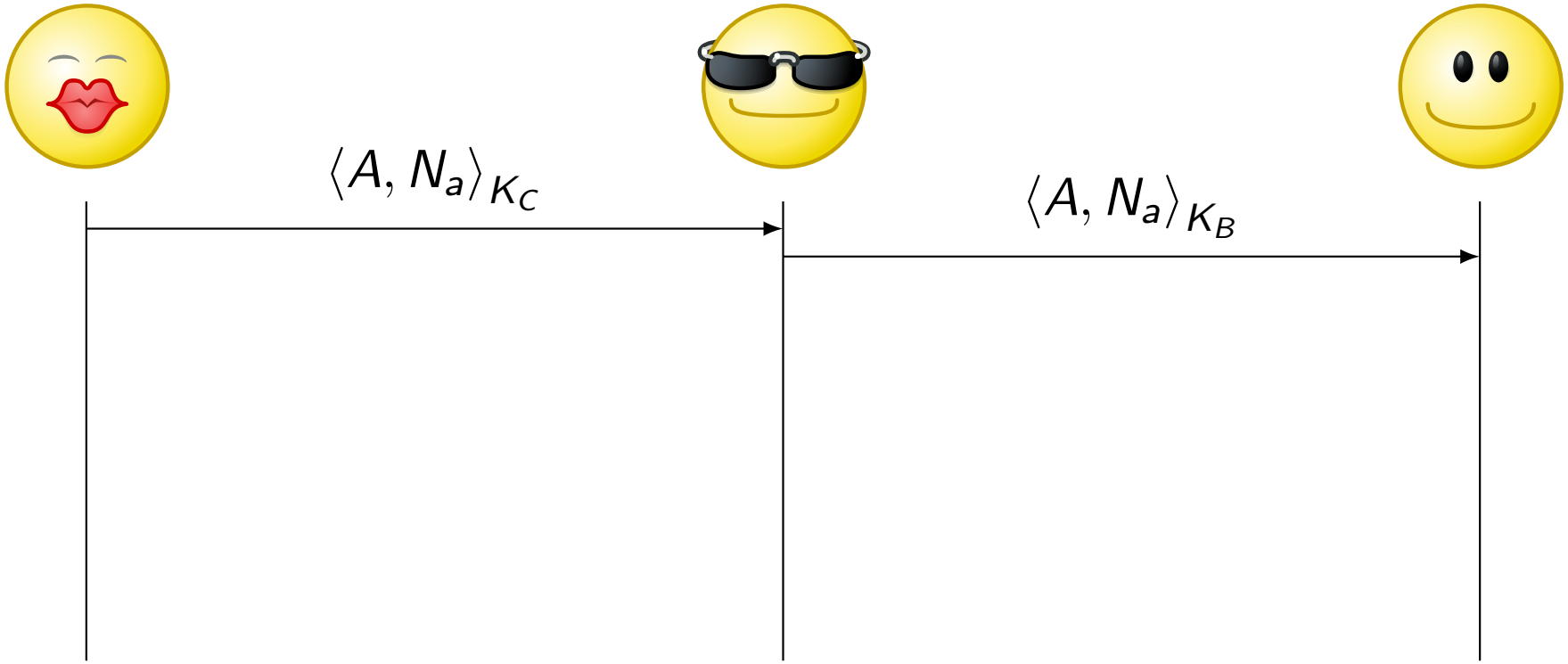
Attack discovered 17 years later!

Lowe (1995), using a dedicated process algebra



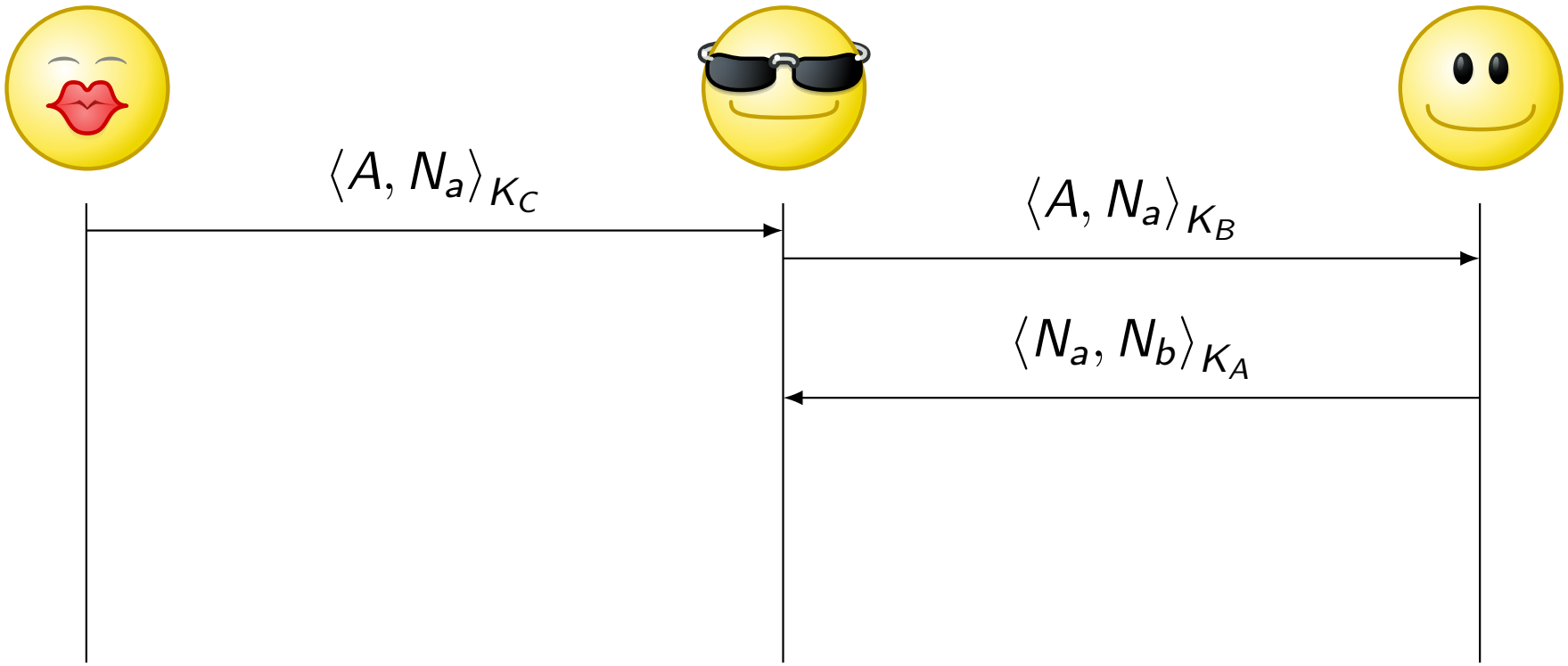
Attack discovered 17 years later!

Lowe (1995), using a dedicated process algebra



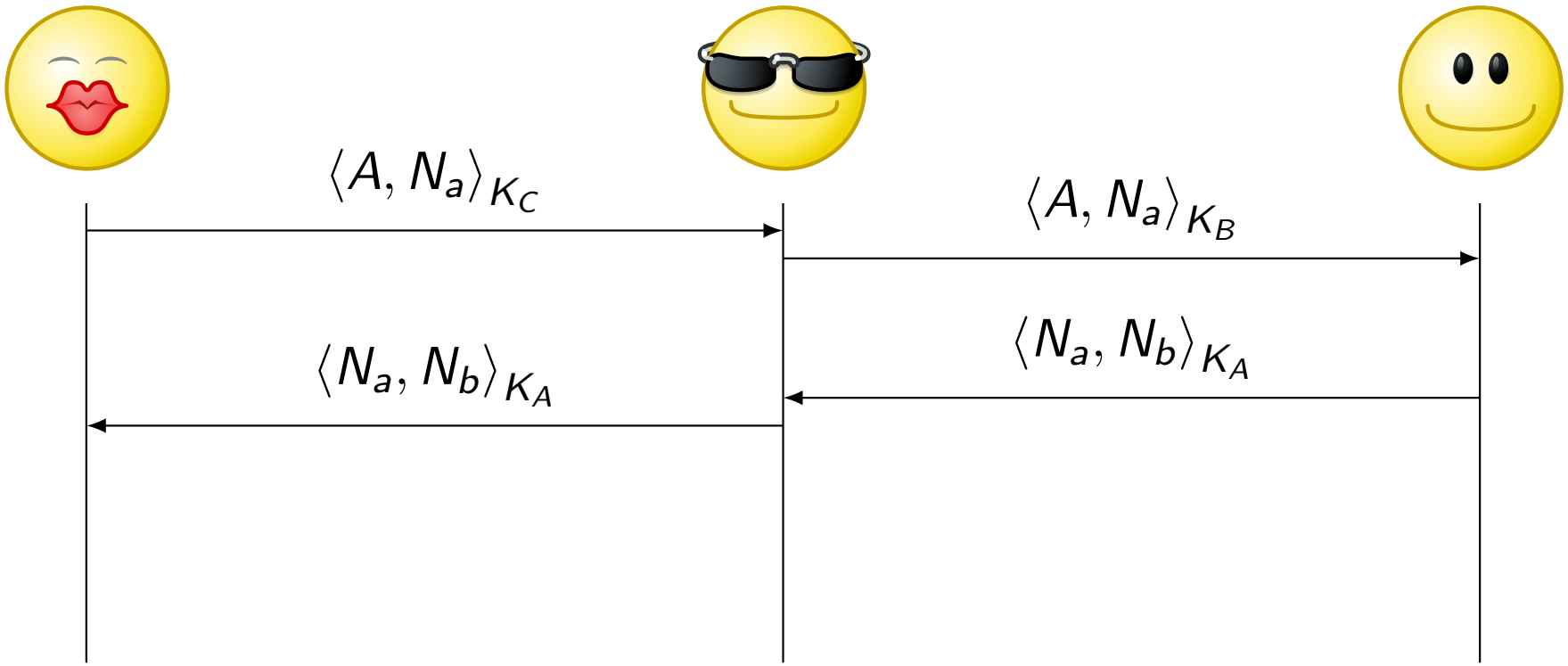
Attack discovered 17 years later!

Lowe (1995), using a dedicated process algebra



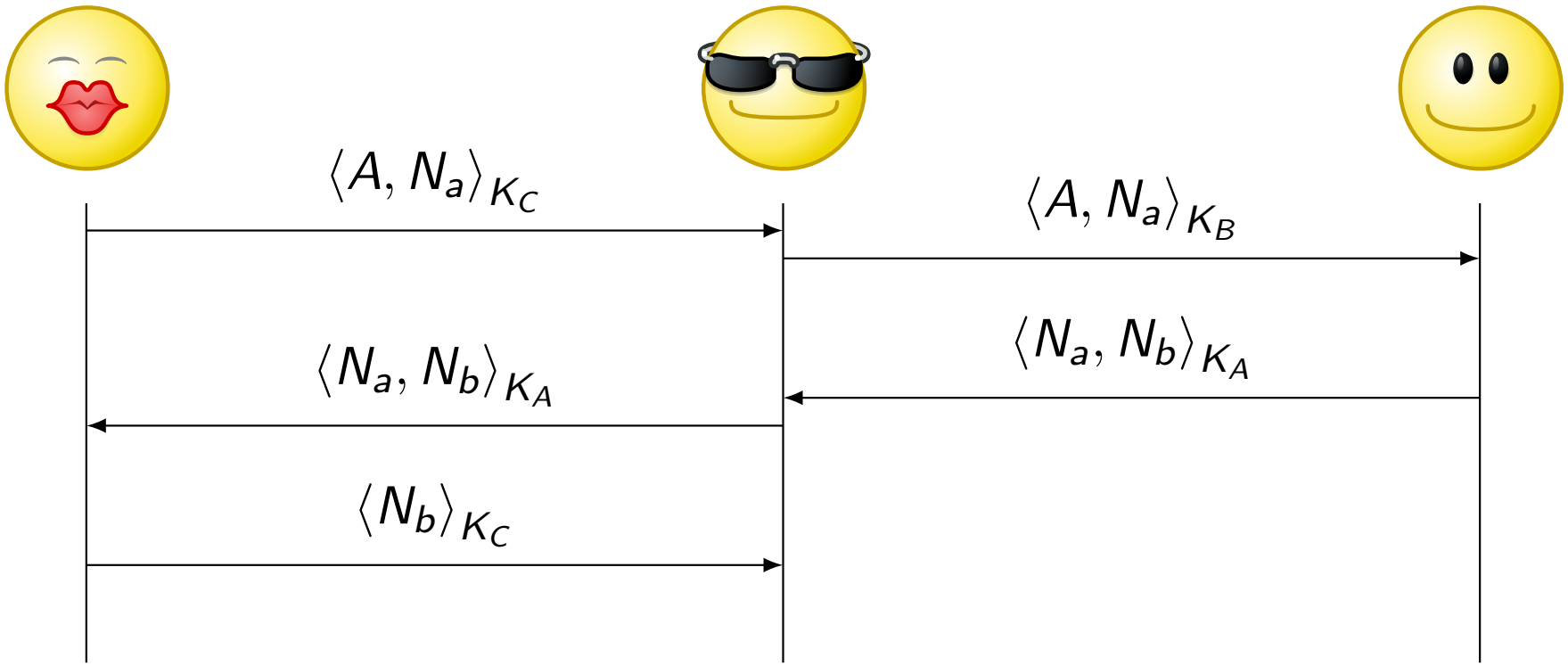
Attack discovered 17 years later!

Lowe (1995), using a dedicated process algebra



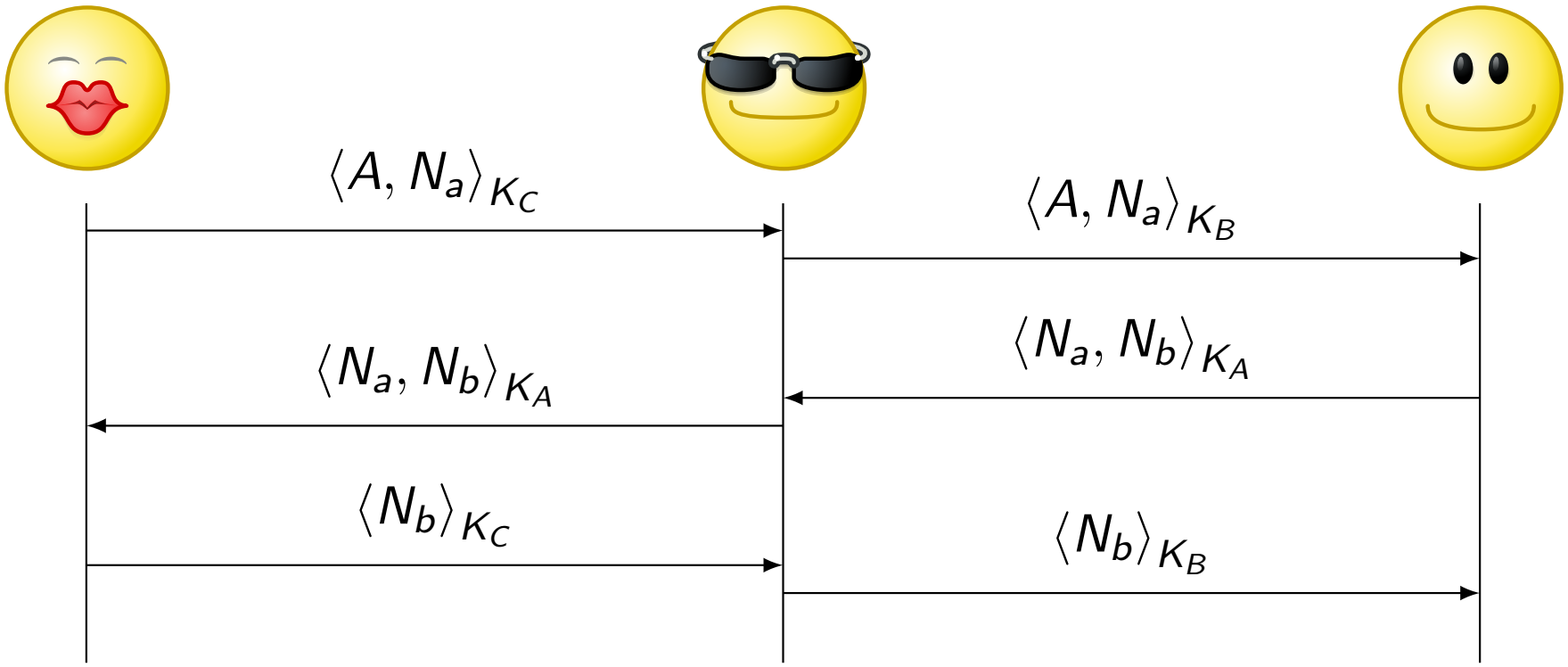
Attack discovered 17 years later!

Lowe (1995), using a dedicated process algebra



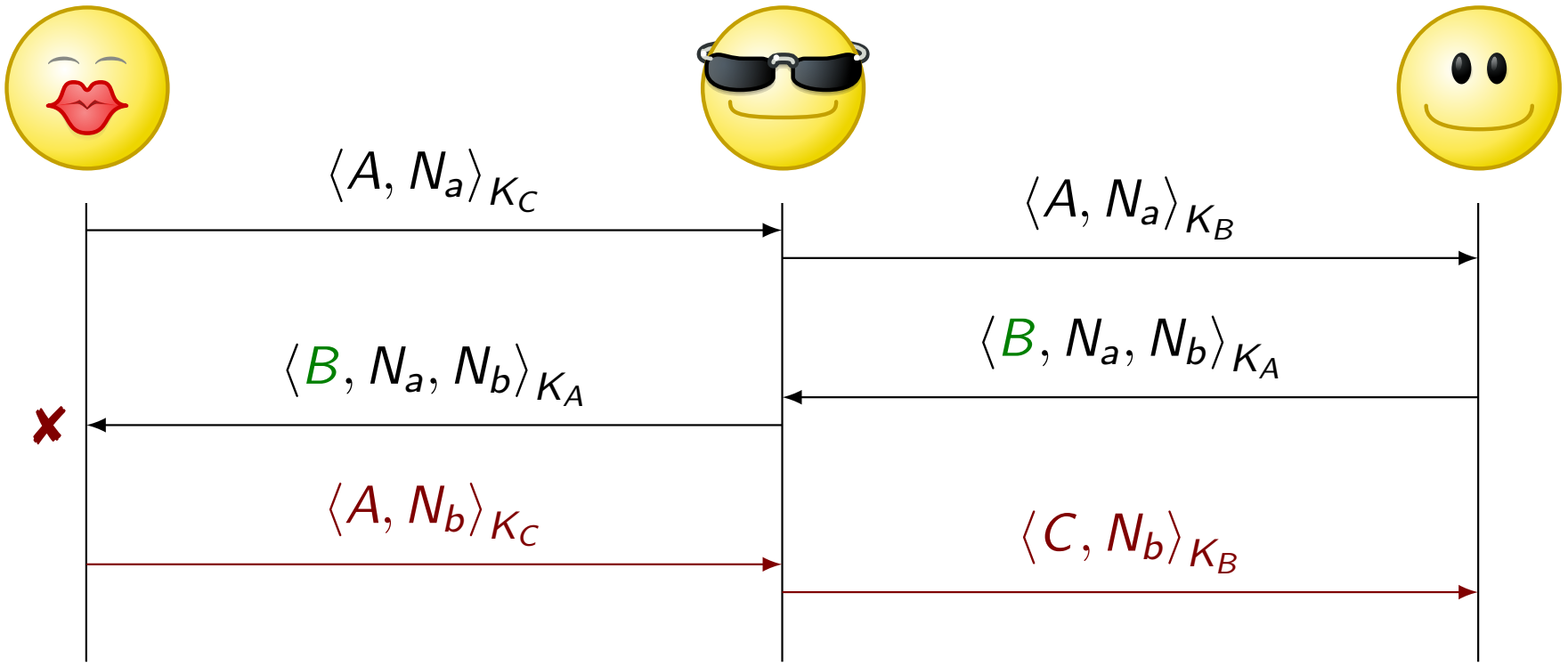
Attack discovered 17 years later!

Lowe (1995), using a dedicated process algebra



Attack discovered 17 years later!

Lowe (1995), using a dedicated process algebra



Modelling NS protocol with ABCD

Alice and global declarations

$$A \rightarrow B : \langle A, N_a \rangle_B$$
$$B \rightarrow A : \langle N_a, N_b \rangle_A$$
$$A \rightarrow B : \langle N_b \rangle_B$$

```
from dolev_yao import *

buffer nw : object = ()
```

Modelling NS protocol with ABCD

Alice and global declarations

$$A \rightarrow B : \langle A, N_a \rangle_B$$
$$B \rightarrow A : \langle N_a, N_b \rangle_A$$
$$A \rightarrow B : \langle N_b \rangle_B$$

```
from dolev_yao import *

buffer nw : object = ()

net Alice (this, peer) :
```

Modelling NS protocol with ABCD

Alice and global declarations

$$A \rightarrow B : \langle A, N_a \rangle_B$$
$$B \rightarrow A : \langle N_a, N_b \rangle_A$$
$$A \rightarrow B : \langle N_b \rangle_B$$

```
from dolev_yao import *

buffer nw : object = ()

net Alice (this, peer) :
    buffer nonce : Nonce = ()
```

Modelling NS protocol with ABCD

Alice and global declarations

$$A \rightarrow B : \langle A, N_a \rangle_B$$
$$B \rightarrow A : \langle N_a, N_b \rangle_A$$
$$A \rightarrow B : \langle N_b \rangle_B$$

```
from dolev_yao import *

buffer nw : object = ()

net Alice (this, peer) :
    buffer nonce : Nonce = ()
    [nw+(CRYPT, (PUB, peer), this, Nonce(this))]
```

Modelling NS protocol with ABCD

Alice and global declarations

$$A \rightarrow B : \langle A, N_a \rangle_B$$
$$B \rightarrow A : \langle N_a, N_b \rangle_A$$
$$A \rightarrow B : \langle N_b \rangle_B$$

```
from dolev_yao import *

buffer nw : object = ()

net Alice (this, peer) :
    buffer nonce : Nonce = ()
    [nw+(CRYPT, (PUB, peer), this, Nonce(this))]
    ; [nw-(CRYPT, (PUB, this), Na, Nb), nonce+(Nb) if Na == Nonce(this)]
```

Modelling NS protocol with ABCD

Alice and global declarations

$$A \rightarrow B : \langle A, N_a \rangle_B$$
$$B \rightarrow A : \langle N_a, N_b \rangle_A$$
$$A \rightarrow B : \langle N_b \rangle_B$$

```
from dolev_yao import *

buffer nw : object = ()

net Alice (this, peer) :
    buffer nonce : Nonce = ()
    [nw+(CRYPT, (PUB, peer), this, Nonce(this))]
    ; [nw-(CRYPT, (PUB, this), Na, Nb), nonce+(Nb) if Na == Nonce(this)]
    ; [nonce?(Nb), nw+(CRYPT, (PUB, peer), Nb)]
```

Modelling Bob

$$A \rightarrow B \ : \ \langle A, N_a \rangle_B$$
$$B \rightarrow A \ : \ \langle N_a, N_b \rangle_A$$
$$A \rightarrow B \ : \ \langle N_b \rangle_B$$

```
net Bob (this) :
  buffer peer : int = ()
  buffer nonce : Nonce = ()
  [nw-(CRYPT, (PUB, this), A, Na), peer+(A), nonce+(Na)]
  ; [peer?(A), nonce?(Na), nw+(CRYPT, (PUB, A), Na, Nonce(this))]
  ; [nw-(CRYPT, (PUB, this), Nb) if Nb == Nonce(this)]
```

Modelling Mallory and a scenario

$$A \rightarrow B \ : \ \langle A, N_a \rangle_B$$
$$B \rightarrow A \ : \ \langle N_a, N_b \rangle_A$$
$$A \rightarrow B \ : \ \langle N_b \rangle_B$$

```
net Mallory (this, init) :
  buffer knowledge : object = (this, Nonce(this), (PRIV, this)) + init
```

Modelling Mallory and a scenario

$$A \rightarrow B \ : \ \langle A, N_a \rangle_B$$
$$B \rightarrow A \ : \ \langle N_a, N_b \rangle_A$$
$$A \rightarrow B \ : \ \langle N_b \rangle_B$$

```
net Mallory (this, init) :
  buffer knowledge : object = (this, Nonce(this), (PRIV, this)) + init
  buffer spy : object = Spy((CRYPT, (PUB, int), int, Nonce),
                            (CRYPT, (PUB, int), Nonce, Nonce),
                            (CRYPT, (PUB, int), Nonce))
```

Modelling Mallory and a scenario

$$A \rightarrow B \ : \ \langle A, N_a \rangle_B$$
$$B \rightarrow A \ : \ \langle N_a, N_b \rangle_A$$
$$A \rightarrow B \ : \ \langle N_b \rangle_B$$

```
net Mallory (this, init) :
  buffer knowledge : object = (this, Nonce(this), (PRIV, this)) + init
  buffer spy : object = Spy((CRYPT, (PUB, int), int, Nonce),
                            (CRYPT, (PUB, int), Nonce, Nonce),
                            (CRYPT, (PUB, int), Nonce))
  ([spy?(s), nw-(m), knowledge>>(k), knowledge<<(s.learn(m, k))]
```

Modelling Mallory and a scenario

$$A \rightarrow B \ : \ \langle A, N_a \rangle_B$$
$$B \rightarrow A \ : \ \langle N_a, N_b \rangle_A$$
$$A \rightarrow B \ : \ \langle N_b \rangle_B$$

```
net Mallory (this, init) :
  buffer knowledge : object = (this, Nonce(this), (PRIV, this)) + init
  buffer spy : object = Spy((CRYPT, (PUB, int), int, Nonce),
                            (CRYPT, (PUB, int), Nonce, Nonce),
                            (CRYPT, (PUB, int), Nonce))
  ([spy?(s), nw-(m), knowledge>>(k), knowledge<<(s.learn(m, k))]
   ; ([True] + [spy?(s), knowledge?(x), nw+(x) if s.message(x)]))
```

Modelling Mallory and a scenario

$$A \rightarrow B \ : \ \langle A, N_a \rangle_B$$
$$B \rightarrow A \ : \ \langle N_a, N_b \rangle_A$$
$$A \rightarrow B \ : \ \langle N_b \rangle_B$$

```
net Mallory (this, init) :
  buffer knowledge : object = (this, Nonce(this), (PRIV, this)) + init
  buffer spy : object = Spy((CRYPT, (PUB, int), int, Nonce),
                            (CRYPT, (PUB, int), Nonce, Nonce),
                            (CRYPT, (PUB, int), Nonce))
  ([spy?(s), nw-(m), knowledge>>(k), knowledge<<(s.learn(m, k))]
   ; ([True] + [spy?(s), knowledge?(x), nw+(x) if s.message(x)]))
  * [False]
```

Modelling Mallory and a scenario

$$A \rightarrow B \ : \ \langle A, N_a \rangle_B$$
$$B \rightarrow A \ : \ \langle N_a, N_b \rangle_A$$
$$A \rightarrow B \ : \ \langle N_b \rangle_B$$

```
net Mallory (this, init) :
  buffer knowledge : object = (this, Nonce(this), (PRIV, this)) + init
  buffer spy : object = Spy((CRYPT, (PUB, int), int, Nonce),
                            (CRYPT, (PUB, int), Nonce, Nonce),
                            (CRYPT, (PUB, int), Nonce))
  ([spy?(s), nw-(m), knowledge>>(k), knowledge<<(s.learn(m, k))]
   ; ([True] + [spy?(s), knowledge?(x), nw+(x) if s.message(x)]))
  * [False]

Alice(1, 3) | Bob(2) | Mallory(3, (1, (PUB, 1), 2, (PUB, 2)))
```

Verification

- search states such that
  - Bob is in an exit marking
  - Bob's buffers peer and nonce are not consistently marked
- reachable markings computed with SNAKES
- 1810 markings, 2 for the known attack

Verification

- search states such that
  - Bob is in an exit marking
  - Bob's buffers peer and nonce are not consistently marked
- reachable markings computed with SNAKES
- 1810 markings, 2 for the known attack
- comparison with a Helena implementation
  - comparable execution times (compilation + verification)
  - much faster and easier modelling

Outline

A modular framework

ABCD of modelling

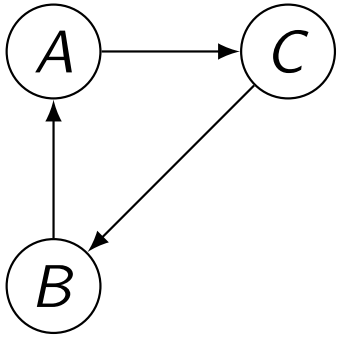
More than simulation

Three application domains
- Timed systems
- Security protocols
- Biological regulatory networks

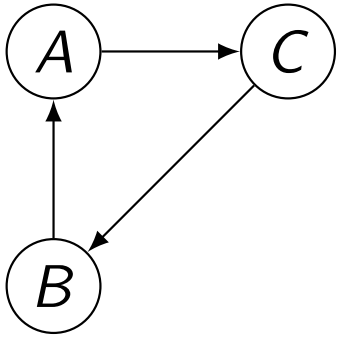
Verification issues

Conclusion

A modular logical formalism



A modular logical formalism

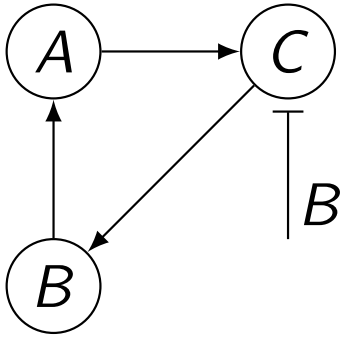


$$K_A(x_B) \stackrel{\text{df}}{=} 1 \text{ if } (x_B = 1) \text{ else } 0$$

$$K_B(x_C) \stackrel{\text{df}}{=} 1 \text{ if } (x_C = 1) \text{ else } 0$$

$$K_C(x_A) \stackrel{\text{df}}{=} 1 \text{ if } (x_A = 1) \text{ else } 0$$

A modular logical formalism

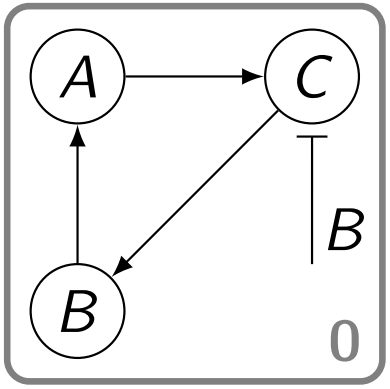


$$K_A(x_B) \stackrel{\text{df}}{=} 1 \text{ if } (x_B = 1) \text{ else } 0$$

$$K_B(x_C) \stackrel{\text{df}}{=} 1 \text{ if } (x_C = 1) \text{ else } 0$$

$$K_C(x_A, \overline{x_B}) \stackrel{\text{df}}{=} 1 \text{ if } (x_A = 1 \wedge \overline{x_B} = 0) \text{ else } 0$$

A modular logical formalism

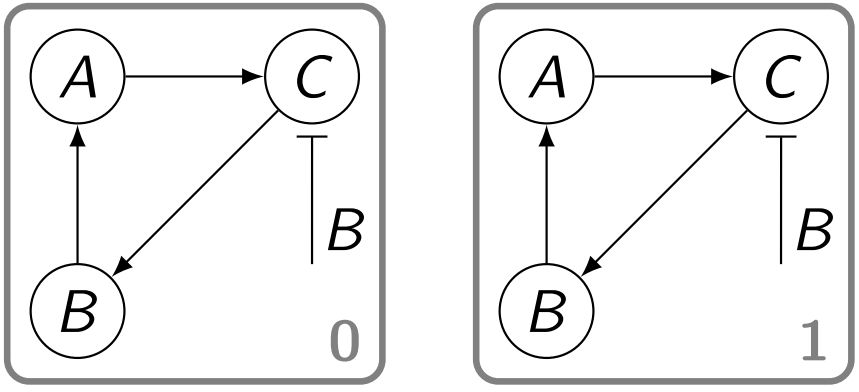


$$K_A(x_B) \stackrel{\text{df}}{=} 1 \text{ if } (x_B = 1) \text{ else } 0$$

$$K_B(x_C) \stackrel{\text{df}}{=} 1 \text{ if } (x_C = 1) \text{ else } 0$$

$$K_C(x_A, \overline{x_B}) \stackrel{\text{df}}{=} 1 \text{ if } (x_A = 1 \wedge \overline{x_B} = 0) \text{ else } 0$$

A modular logical formalism

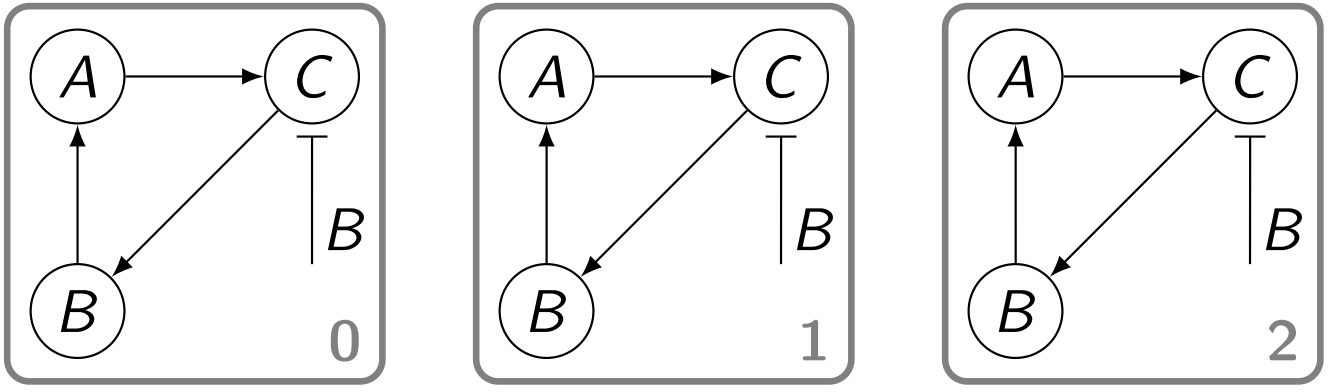


$$K_A(x_B) \stackrel{\text{df}}{=} 1 \text{ if } (x_B = 1) \text{ else } 0$$

$$K_B(x_C) \stackrel{\text{df}}{=} 1 \text{ if } (x_C = 1) \text{ else } 0$$

$$K_C(x_A, \overline{x_B}) \stackrel{\text{df}}{=} 1 \text{ if } (x_A = 1 \wedge \overline{x_B} = 0) \text{ else } 0$$

A modular logical formalism

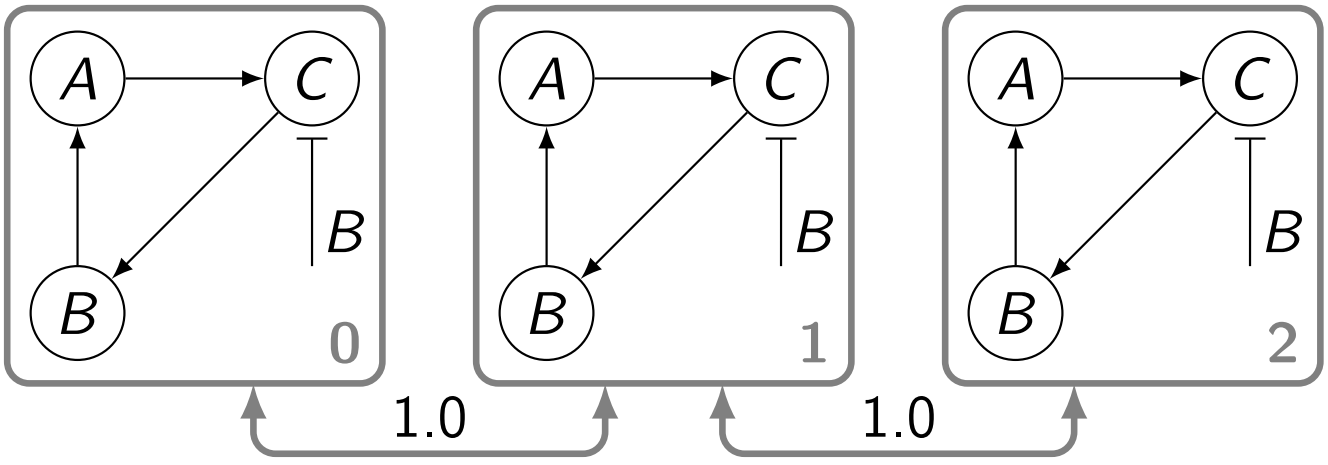


$$K_A(x_B) \stackrel{\text{df}}{=} 1 \text{ if } (x_B = 1) \text{ else } 0$$

$$K_B(x_C) \stackrel{\text{df}}{=} 1 \text{ if } (x_C = 1) \text{ else } 0$$

$$K_C(x_A, \overline{x_B}) \stackrel{\text{df}}{=} 1 \text{ if } (x_A = 1 \wedge \overline{x_B} = 0) \text{ else } 0$$

A modular logical formalism

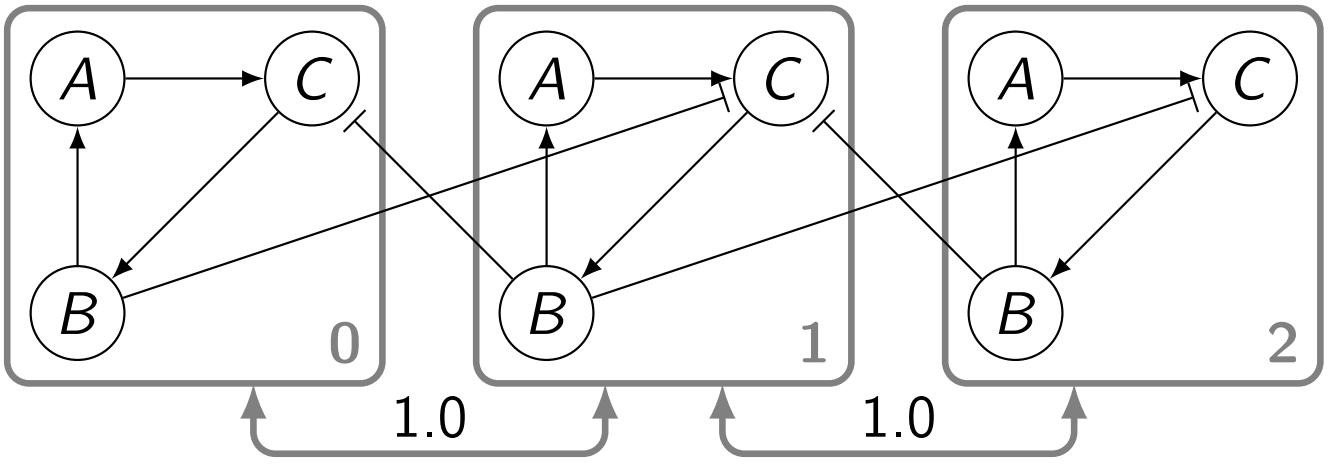


$K_A(x_B) \stackrel{\text{df}}{=} 1$ if $(x_B = 1)$ else 0

$K_B(x_C) \stackrel{\text{df}}{=} 1$ if $(x_C = 1)$ else 0

$K_C(x_A, \overline{x_B}) \stackrel{\text{df}}{=} 1$ if $(x_A = 1 \wedge \overline{x_B} = 0)$ else 0

A modular logical formalism

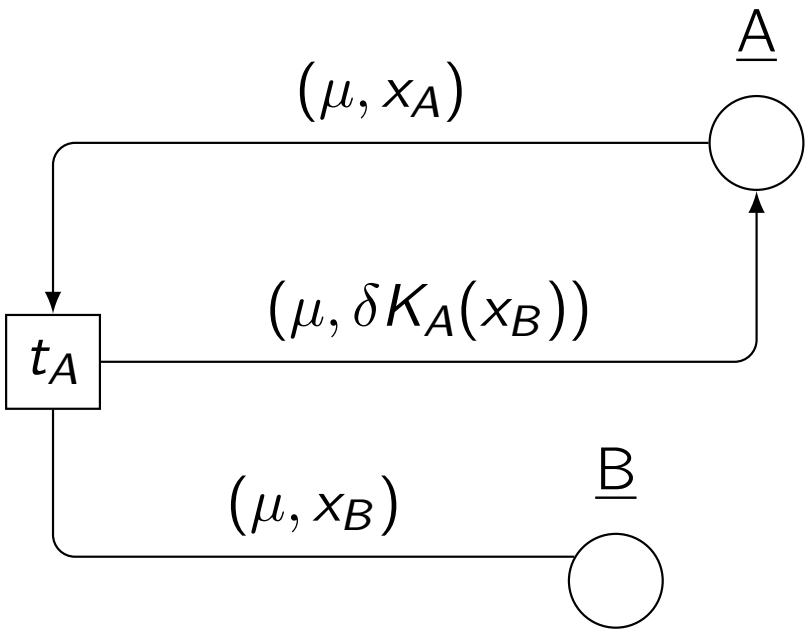


$K_A(x_B) \stackrel{\text{df}}{=} 1 \text{ if } (x_B = 1) \text{ else } 0$

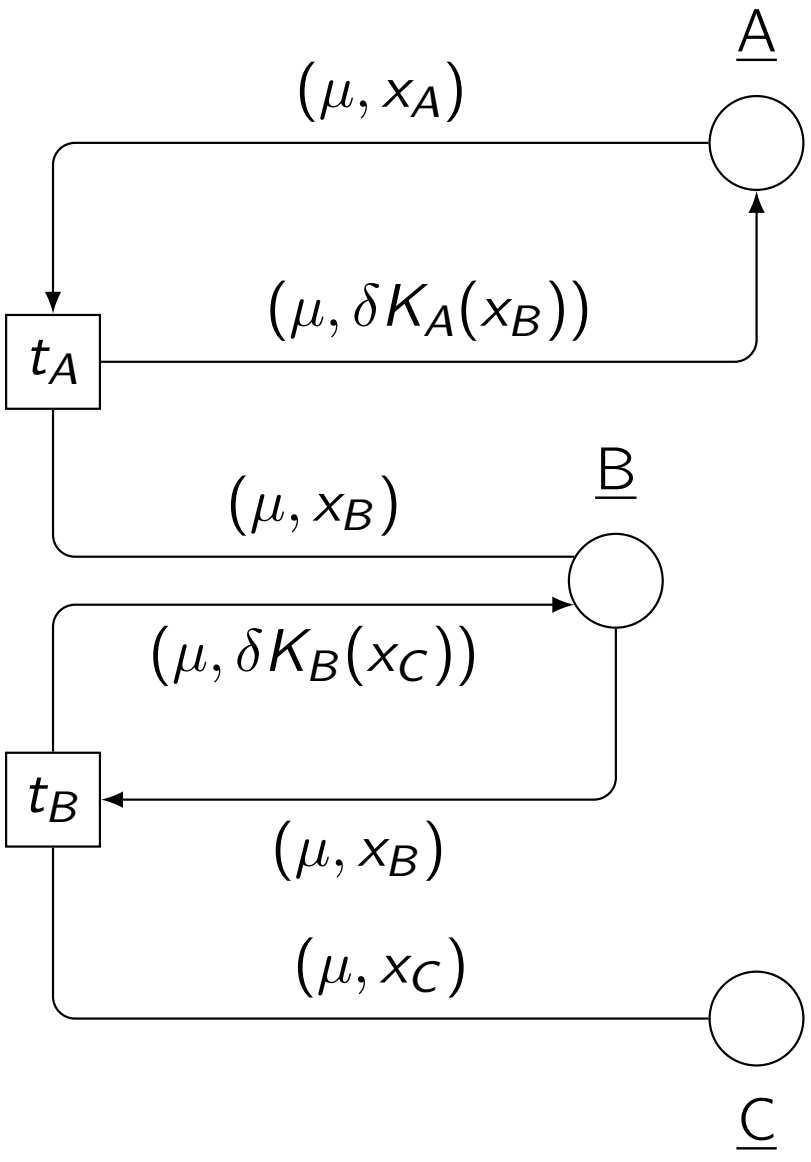
$K_B(x_C) \stackrel{\text{df}}{=} 1 \text{ if } (x_C = 1) \text{ else } 0$

$K_C(x_A, \overline{x_B}) \stackrel{\text{df}}{=} 1 \text{ if } (x_A = 1 \wedge \overline{x_B} = 0) \text{ else } 0$

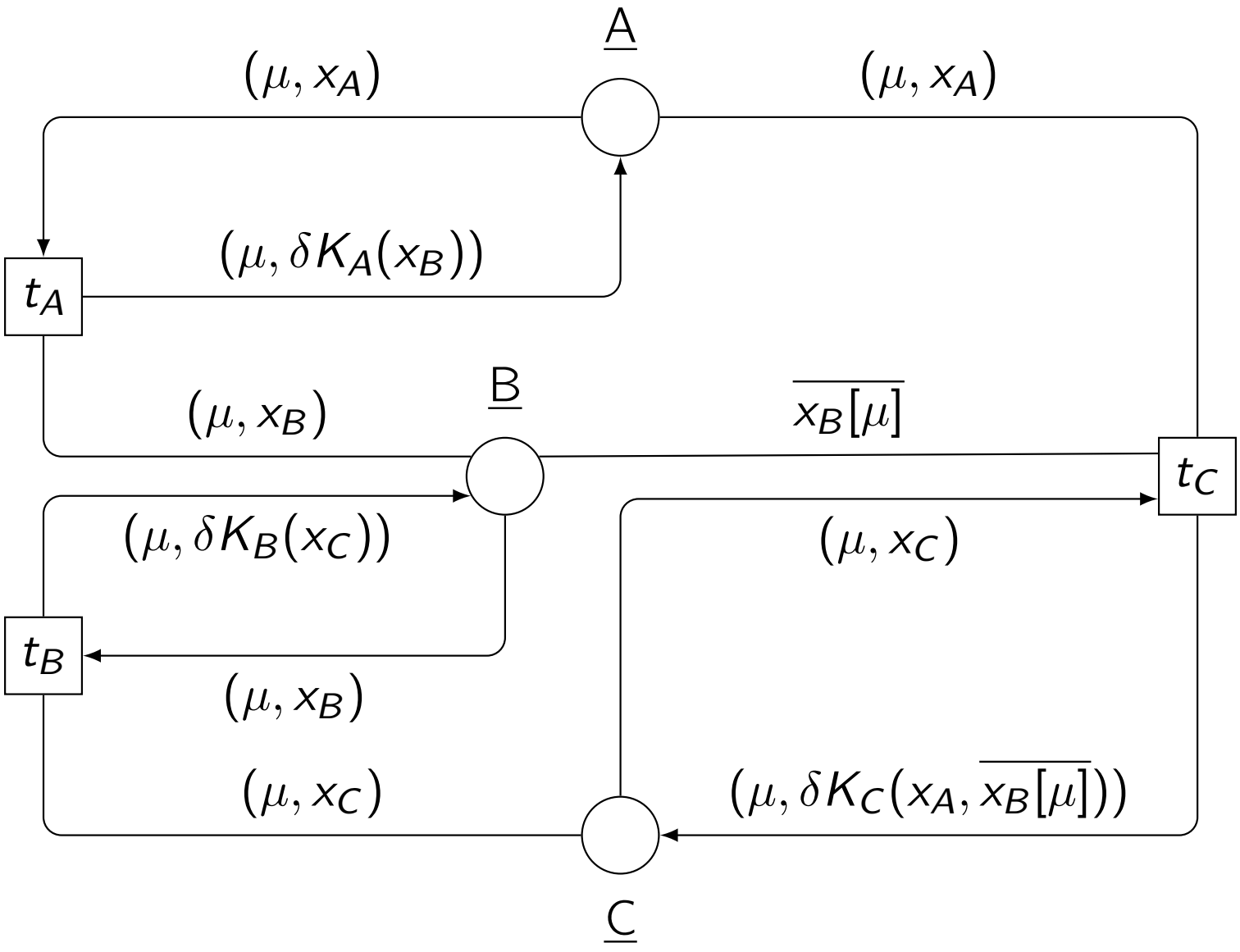
Petri net semantics



Petri net semantics



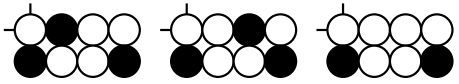
Petri net semantics



Analysis of patterning mechanisms

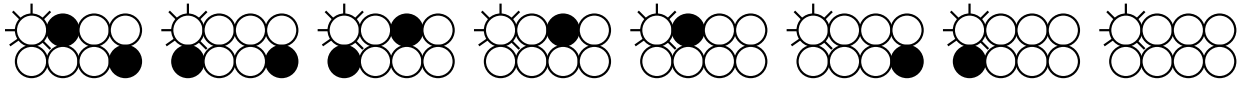
4-connected

64 reachable states (3 stable)



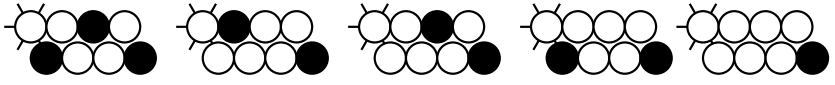
8-connected

4096 reachable states (8 stable)



6-connected

512 reachable states (5 stable)



Outline

A modular framework

ABCD of modelling

More than simulation

Three application domains

Verification issues
- Optimised compilation
- Parallel computation
- Process identifiers
- Clock counters

Conclusion

Outline

A modular framework

ABCD of modelling

More than simulation

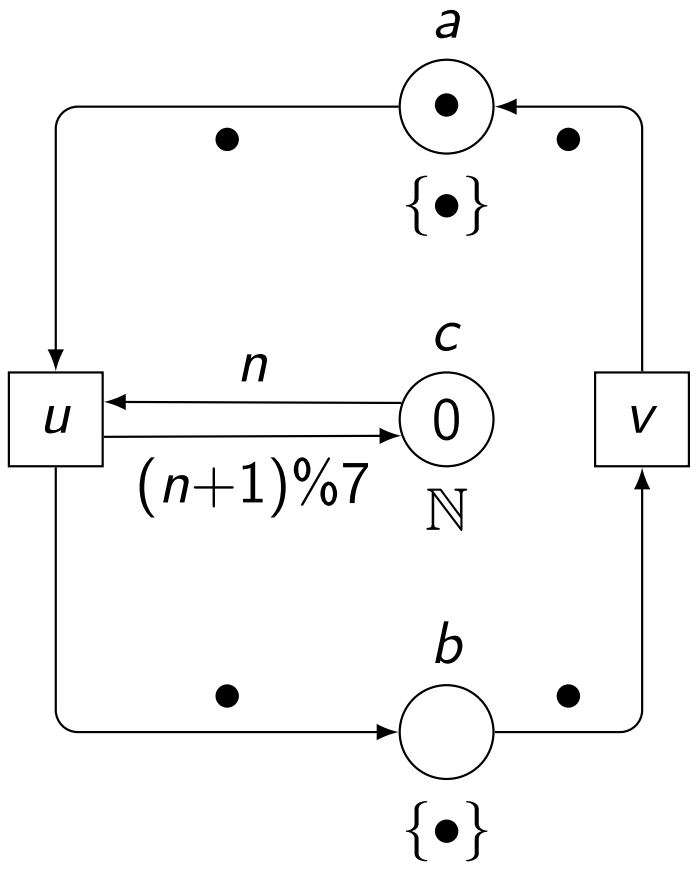
Three application domains

Verification issues

- Optimised compilation
- Parallel computation
- Process identifiers
- Clock counters

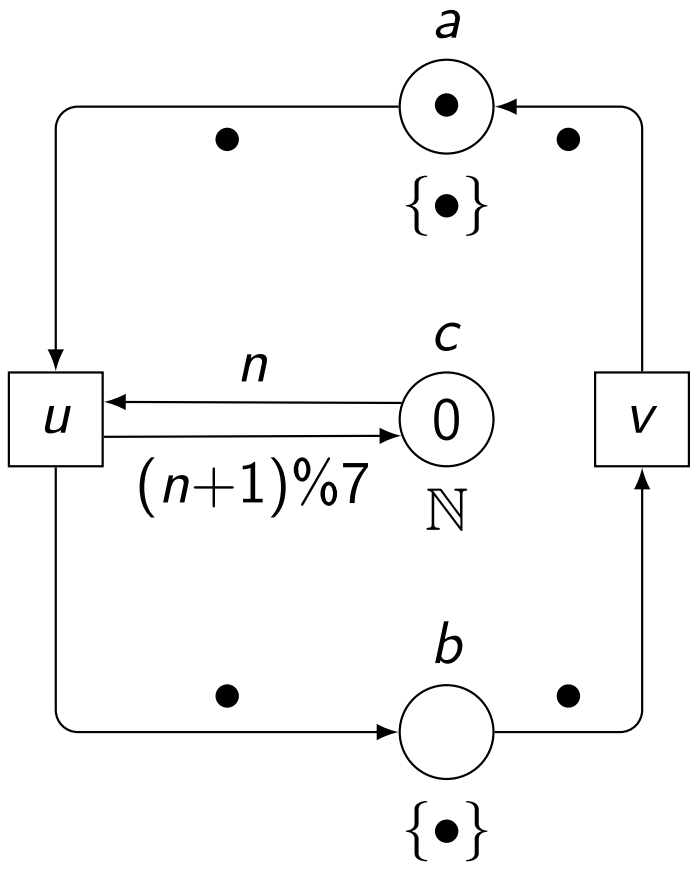
Conclusion

Compiling Petri nets with optimisations



Compiling Petri nets with optimisations

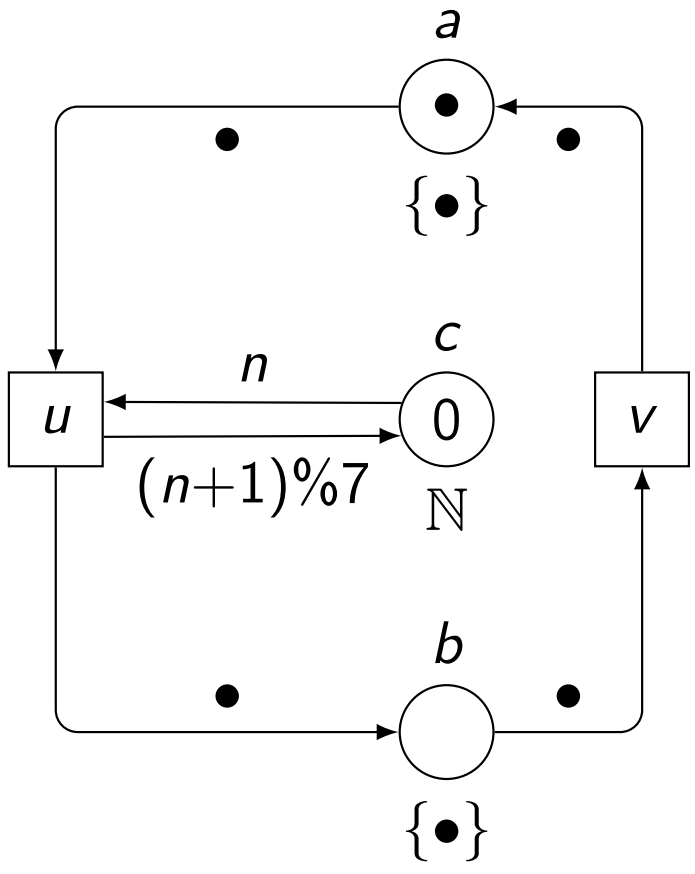
def initial_marking () :
  return $(a \mapsto \{\bullet\}, b \mapsto \{\}, c \mapsto \{0\})$



Compiling Petri nets with optimisations

```
def initial_marking () :
    return  (a ↦ {•}, b ↦ {}, c ↦ {0})

def modes_u (m) :
    result ← {}
    for x in m(a) if x = • :
        for y in m(c) :
            β ← (n ↦ y)
            if β((n + 1)%7) ∈ ℕ :
                result ← result ∪ {β}
    return result
```

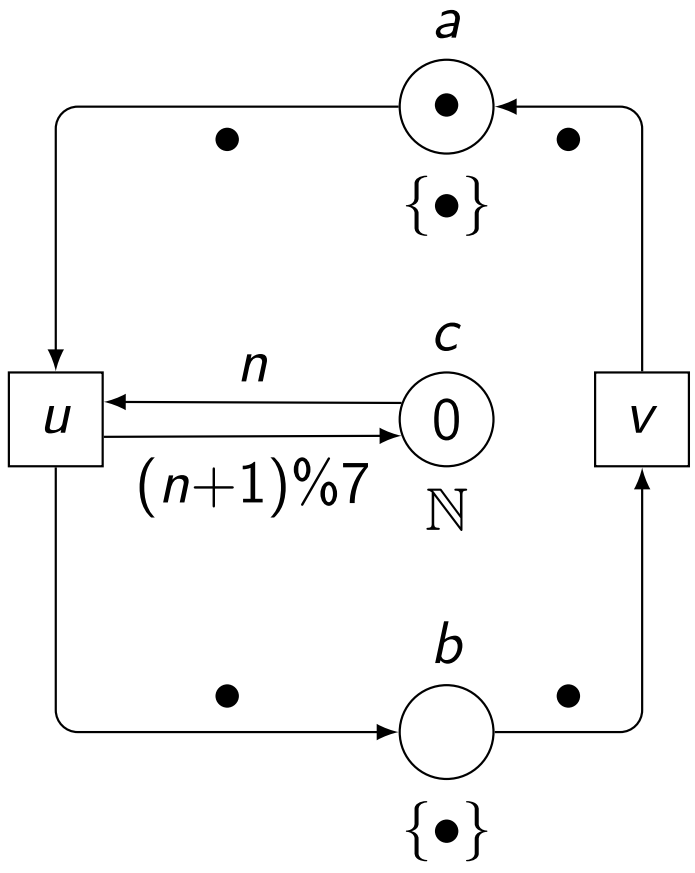


Compiling Petri nets with optimisations

```
def initial_marking () :
   return  (a ↦ {•}, b ↦ {}, c ↦ {0})

def modes_u (m) :
   result ← {}
   for x in m(a) if x = • :
      for y in m(c) :
         β ← (n ↦ y)
         if β((n + 1)%7) ∈ ℕ :
            result ← result ∪ {β}
   return result

def fire_u (m, β) :
   return (a ↦ m(a) − {•},
           b ↦ m(b) + {•},
           c ↦ m(c) − {β(n)} + {β((n + 1)%7)})
```

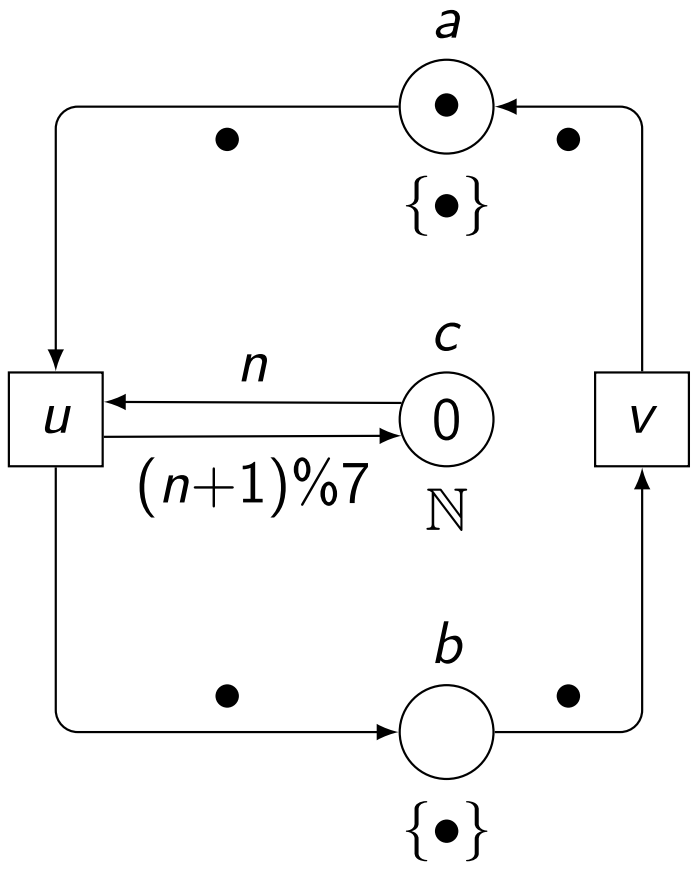


Compiling Petri nets with optimisations

```
def initial_marking () :
    return  (a ↦ {•}, b ↦ {}, c ↦ {0})

def modes_u (m) :
    result ← {}
    for x in m(a) if x = • :
        for y in m(c) :
            β ← (n ↦ y)
            if β((n + 1)%7) ∈ ℕ :
                result ← result ∪ {β}
    return result

def fire_u (m, β) :
    return (a ↦ m(a) − {•},
            b ↦ m(b) + {•},
            c ↦ m(c) − {β(n)} + {β((n + 1)%7)})
```



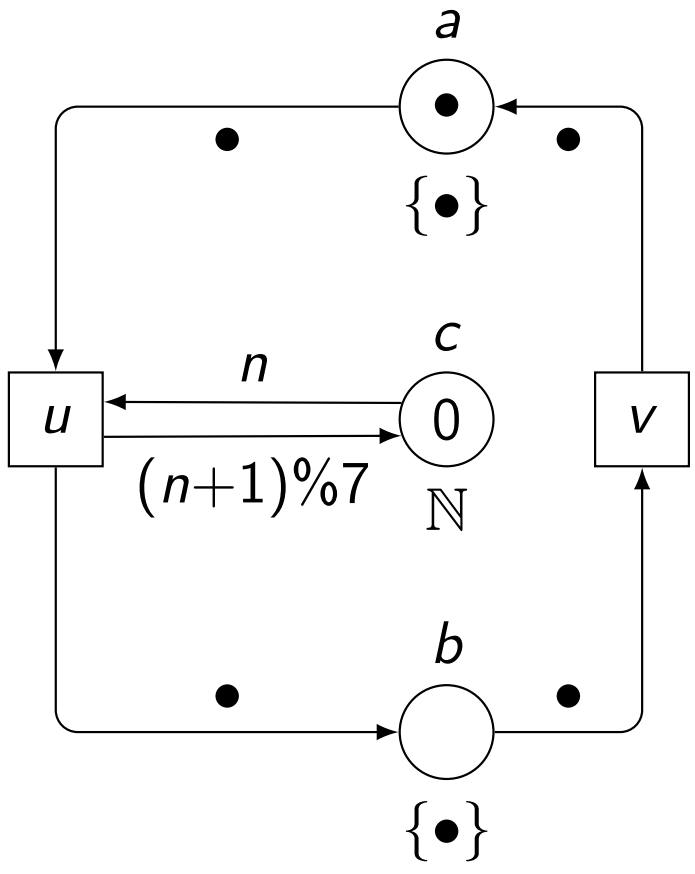
$\ell(a) = \ell(b) = \{\bullet\}$

Compiling Petri nets with optimisations

```
def initial_marking () :
    return  (a ↦ 1, b ↦ 0, c ↦ {0})

def modes_u (m) :
    result ← {}
    for x in m(a) if x = • :
        for y in m(c) :
            β ← (n ↦ y)
            if β((n + 1)%7) ∈ ℕ :
                result ← result ∪ {β}
    return result

def fire_u (m, β) :
    return (a ↦ m(a) − {•},
            b ↦ m(b) + {•},
            c ↦ m(c) − {β(n)} + {β((n + 1)%7)})
```



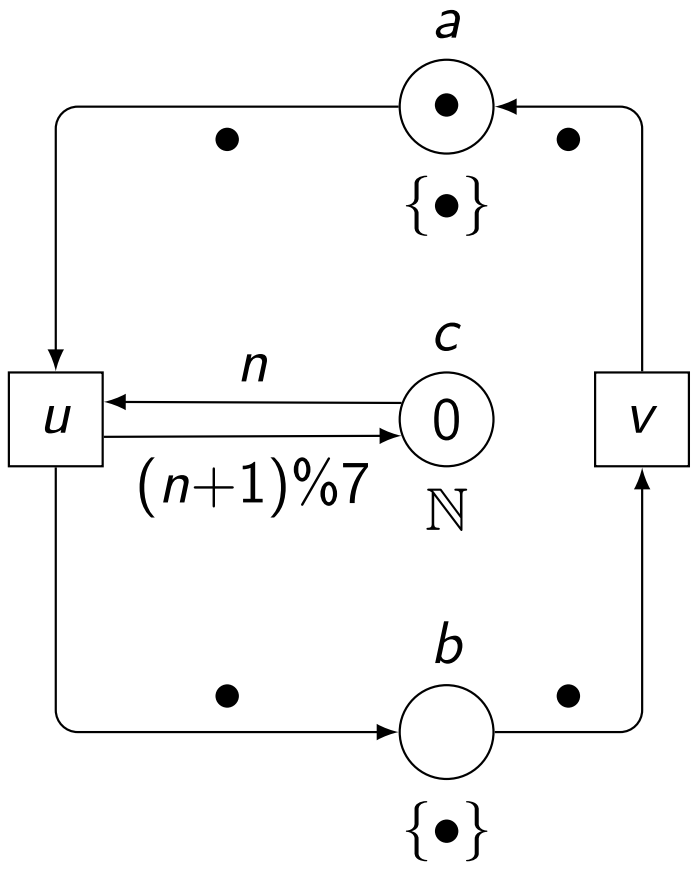
$\ell(a) = \ell(b) = \{\bullet\}$

Compiling Petri nets with optimisations

```
def initial_marking () :
    return  (a ↦ 1, b ↦ 0, c ↦ {0})

def modes_u (m) :
    result ← {}
    if m(a) ≥ 1 :
        for y in m(c) :
            β ← (n ↦ y)
            if β((n + 1)%7) ∈ ℕ :
                result ← result ∪ {β}
    return result

def fire_u (m, β) :
    return (a ↦ m(a) − {•},
            b ↦ m(b) + {•},
            c ↦ m(c) − {β(n)} + {β((n + 1)%7)})
```



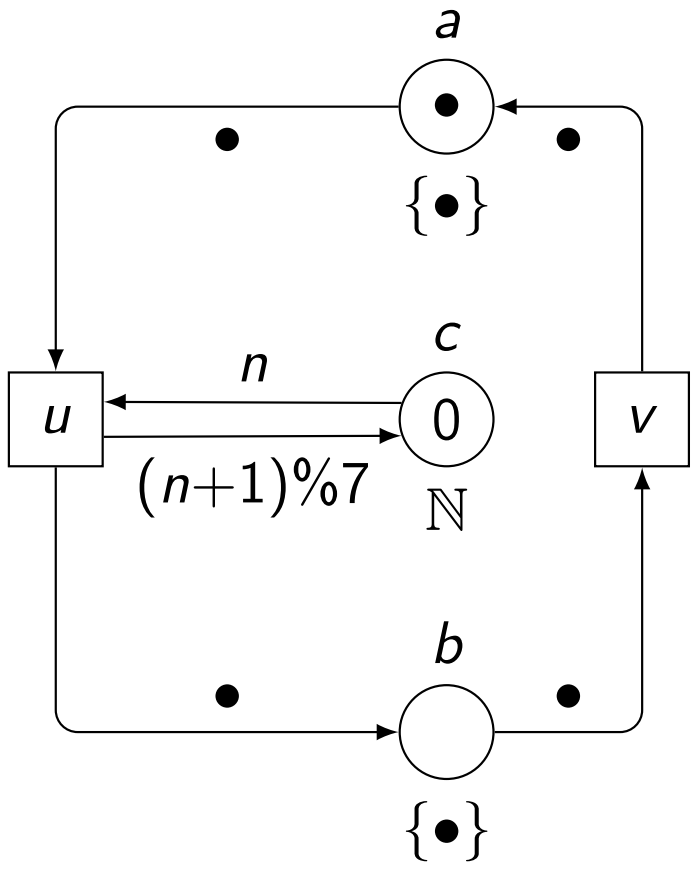
$\ell(a) = \ell(b) = \{\bullet\}$

Compiling Petri nets with optimisations

```
def initial_marking () :
    return  (a ↦ 1, b ↦ 0, c ↦ {0})

def modes_u (m) :
    result ← {}
    if m(a) ≥ 1 :
        for y in m(c) :
            β ← (n ↦ y)
            if β((n + 1)%7) ∈ ℕ :
                result ← result ∪ {β}
    return result

def fire_u (m, β) :
    return (a ↦ m(a) − 1,
            b ↦ m(b) + 1,
            c ↦ m(c) − {β(n)} + {β((n + 1)%7)})
```



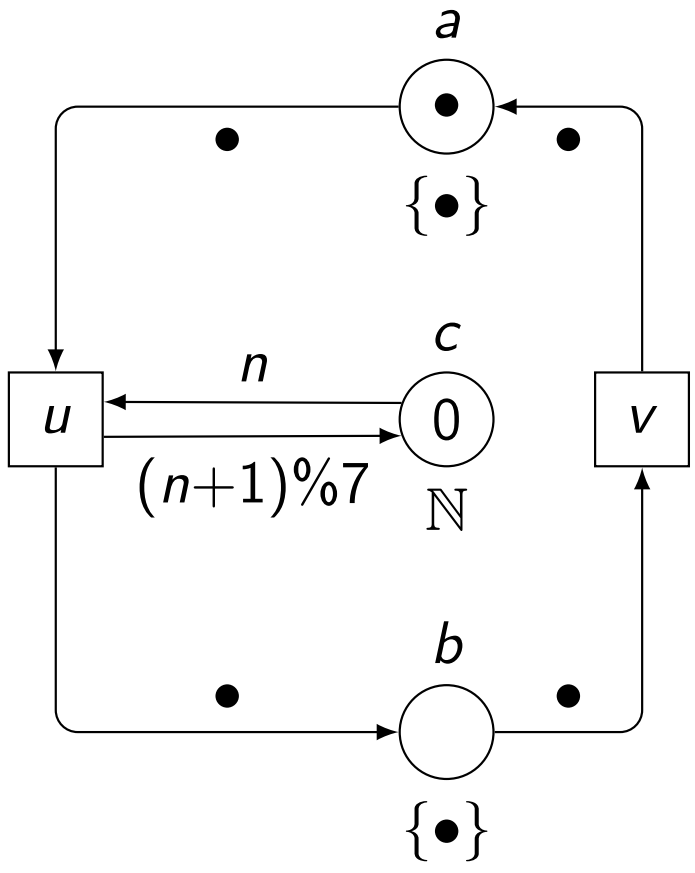
$\ell(a) = \ell(b) = \{\bullet\}$

Compiling Petri nets with optimisations

```
def initial_marking () :
    return  (a ↦ 1, b ↦ 0, c ↦ {0})

def modes_u (m) :
    result ← {}
    if m(a) ≥ 1 :
        for y in m(c) :
            β ← (n ↦ y)
            if β((n + 1)%7) ∈ ℕ :
                result ← result ∪ {β}
    return result

def fire_u (m, β) :
    return (a ↦ m(a) − 1,
            b ↦ m(b) + 1,
            c ↦ m(c) − {β(n)} + {β((n + 1)%7)})
```



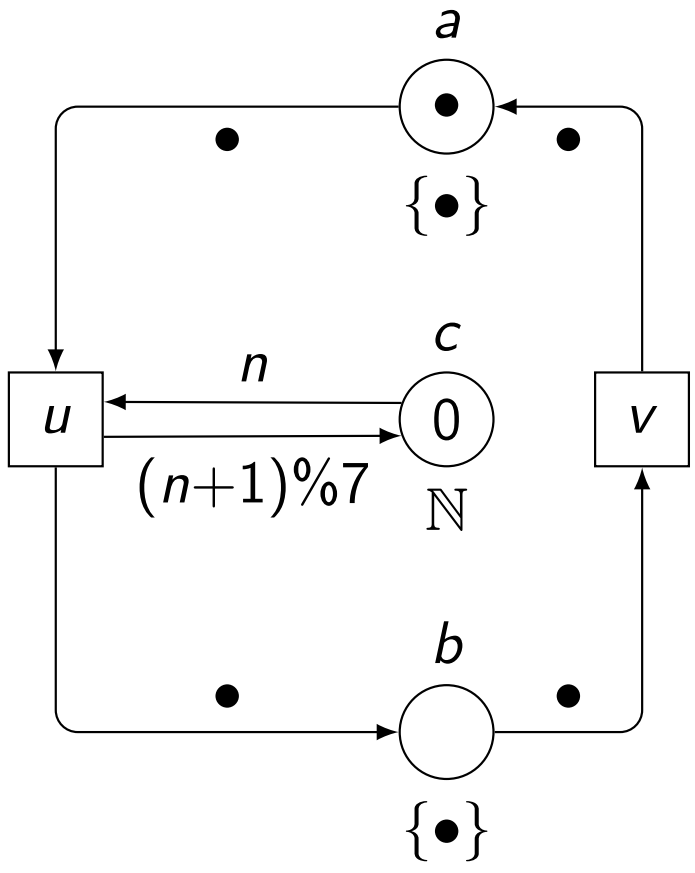
a, b and c are 1-safe

Compiling Petri nets with optimisations

```
def initial_marking () :
    return  (a ↦ True, b ↦ False, c ↦ 0)

def modes_u (m) :
    result ← {}
    if m(a) ≥ 1 :
        for y in m(c) :
            β ← (n ↦ y)
            if β((n + 1)%7) ∈ ℕ :
                result ← result ∪ {β}
    return result

def fire_u (m, β) :
    return (a ↦ m(a) − 1,
            b ↦ m(b) + 1,
            c ↦ m(c) − {β(n)} + {β((n + 1)%7)})
```



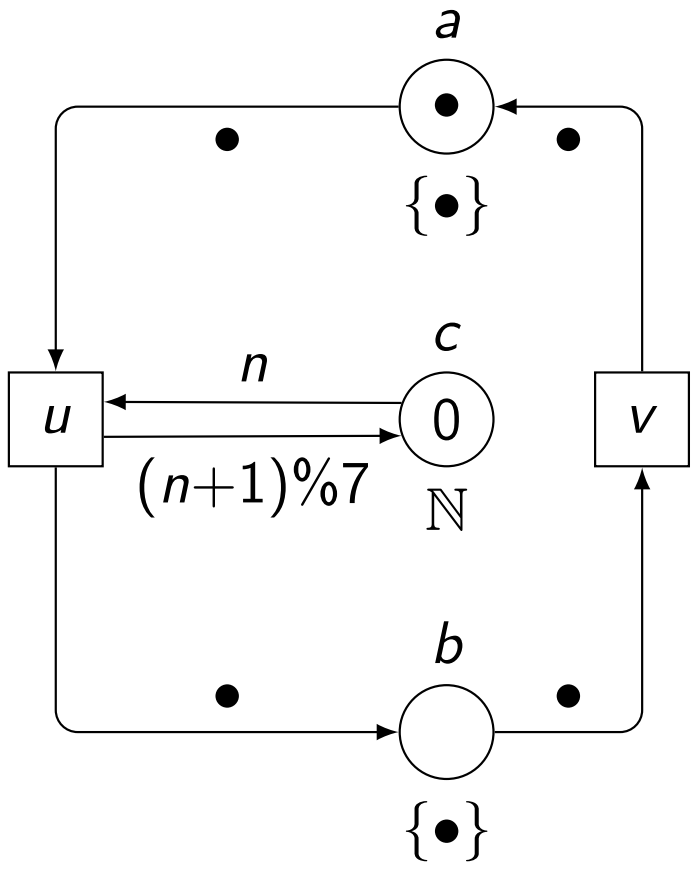
a, b and c are 1-safe

Compiling Petri nets with optimisations

```
def initial_marking () :
    return  (a ↦ True, b ↦ False, c ↦ 0)

def modes_u (m) :
    result ← {}
    if m(a) :
        for y in m(c) :
            β ← (n ↦ y)
            if β((n + 1)%7) ∈ ℕ :
                result ← result ∪ {β}
    return result

def fire_u (m, β) :
    return (a ↦ m(a) − 1,
            b ↦ m(b) + 1,
            c ↦ m(c) − {β(n)} + {β((n + 1)%7)})
```



a, b and c are 1-safe

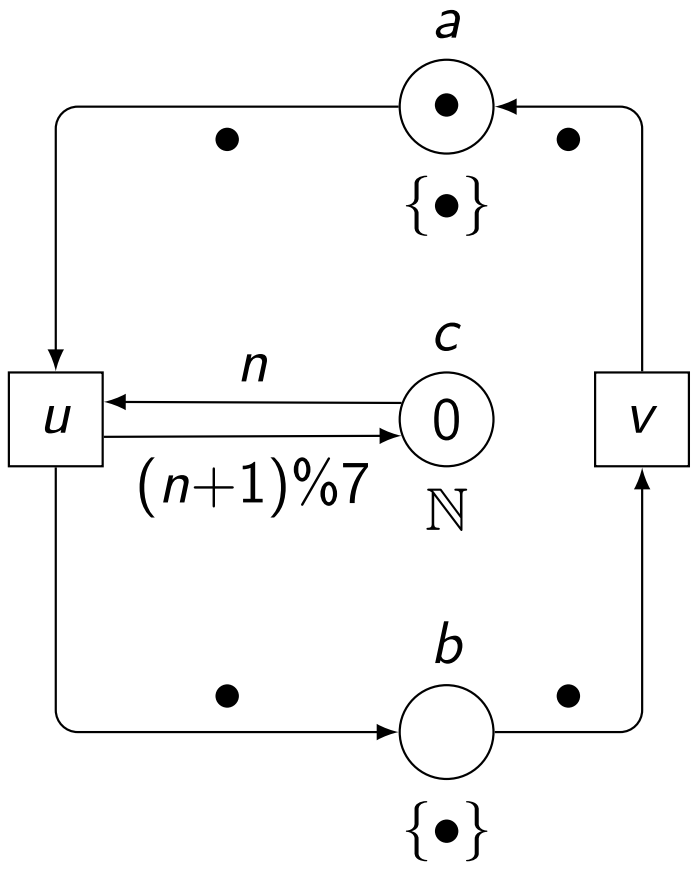
Compiling Petri nets with optimisations

def initial_marking () :
  return $(a \mapsto \text{True}, b \mapsto \text{False}, c \mapsto 0)$

def modes_u (m) :
  result $\leftarrow \{\}$
  if m(a) :

    $\beta \leftarrow (n \mapsto \text{m}(c))$
    if $\beta((n+1)\%7) \in \mathbb{N}$:
      result $\leftarrow$ result $\cup \{\beta\}$
  return result

def fire_u (m, β) :
  return $(a \mapsto \text{m}(a) - 1,$
    $b \mapsto \text{m}(b) + 1,$
    $c \mapsto \text{m}(c) - \{\beta(n)\} + \{\beta((n+1)\%7)\})$



a, b and c are 1-safe

Compiling Petri nets with optimisations

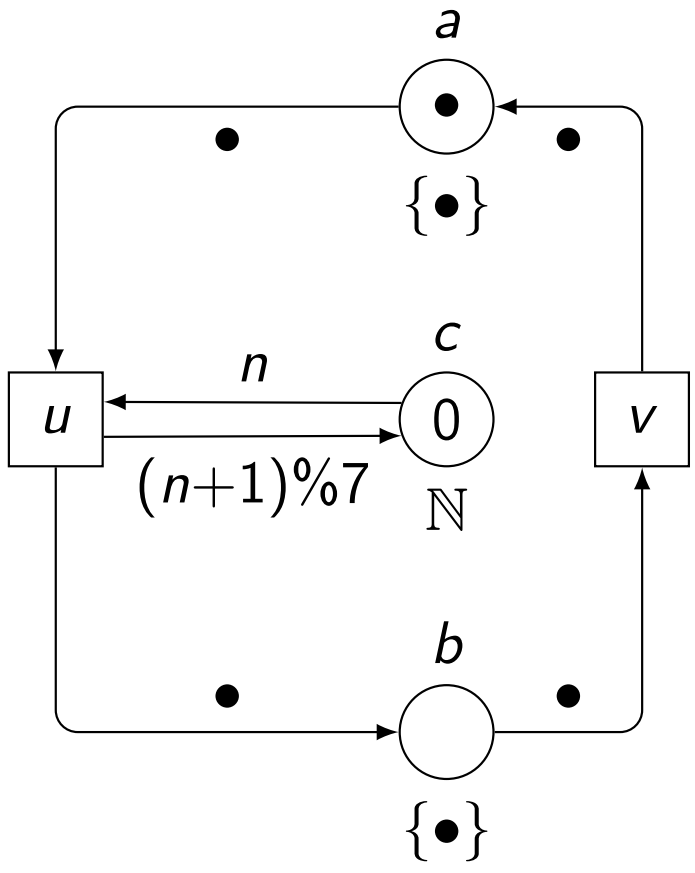
def initial_marking () :
  return $(a \mapsto \text{True}, b \mapsto \text{False}, c \mapsto 0)$

def modes_u (m) :

  if m(a) :

    $\beta \leftarrow (n \mapsto \text{m}(c))$
    if $\beta((n+1)\%7) \in \mathbb{N}$:
      return $\{\beta\}$
  return $\{\}$

def fire_u (m, β) :
  return $(a \mapsto \text{m}(a) - 1,$
    $b \mapsto \text{m}(b) + 1,$
    $c \mapsto \text{m}(c) - \{\beta(n)\} + \{\beta((n+1)\%7)\})$



a, b and c are 1-safe

Compiling Petri nets with optimisations

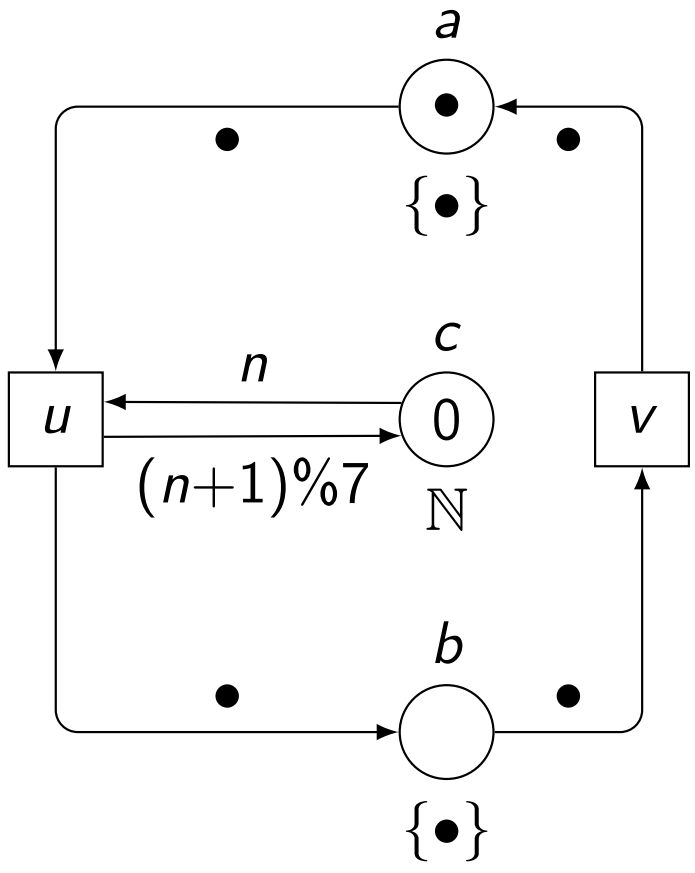
```
def initial_marking () :
   return  (a ↦ True, b ↦ False, c ↦ 0)

def modes_u (m) :

   if m(a) :

      β ← (n ↦ m(c))
      if β((n + 1)%7) ∈ ℕ :
         return {β}
   return {}

def fire_u (m, β) :
   return (a ↦ False,
           b ↦ True,
           c ↦ β((n + 1)%7))
```



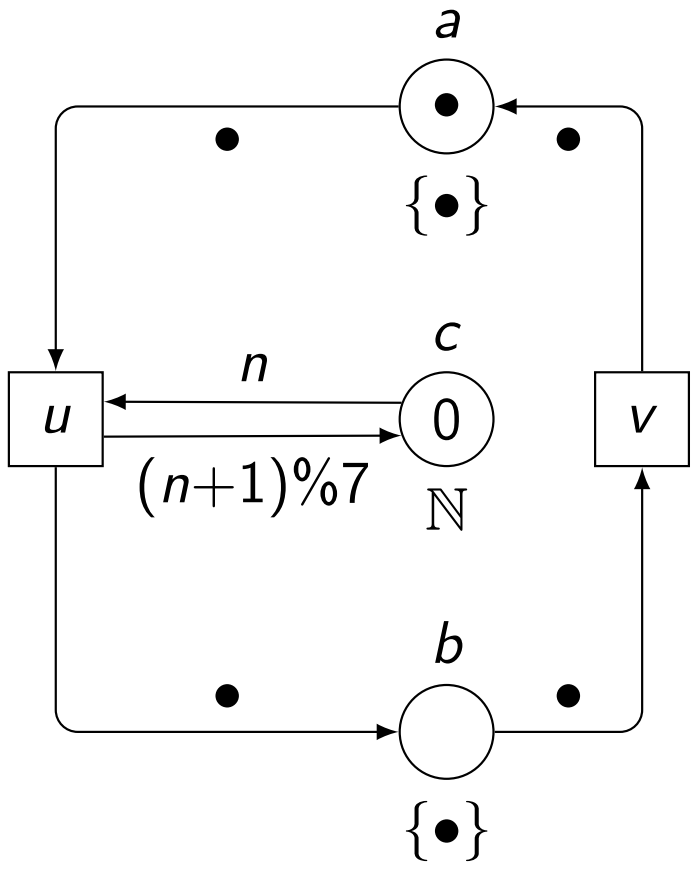
a, b and c are 1-safe

Further optimisations

- pass bindings as parameters
- maintain sets of possible firings
  - use place invariants
  - use token flows
- exploit how the Petri net is constructed
  - deduce invariants
  - deduce places boundedness

Further optimisations

- pass bindings as parameters
- maintain sets of possible firings
  - use place invariants
  - use token flows
- exploit how the Petri net is constructed
  - deduce invariants
  - deduce places boundedness



Further optimisations

- pass bindings as parameters
- maintain sets of possible firings
  - use place invariants
  - use token flows
- exploit how the Petri net is constructed
  - deduce invariants
  - deduce places boundedness

Further optimisations

- pass bindings as parameters
- maintain sets of possible firings
  - use place invariants
  - use token flows
- exploit how the Petri net is constructed
  - deduce invariants
  - deduce places boundedness

Experiments

- speedups from 6.5 to 800
- better speedups on "less coloured" Petri nets
- work in progress

Outline

A modular framework

ABCD of modelling

More than simulation

Three application domains

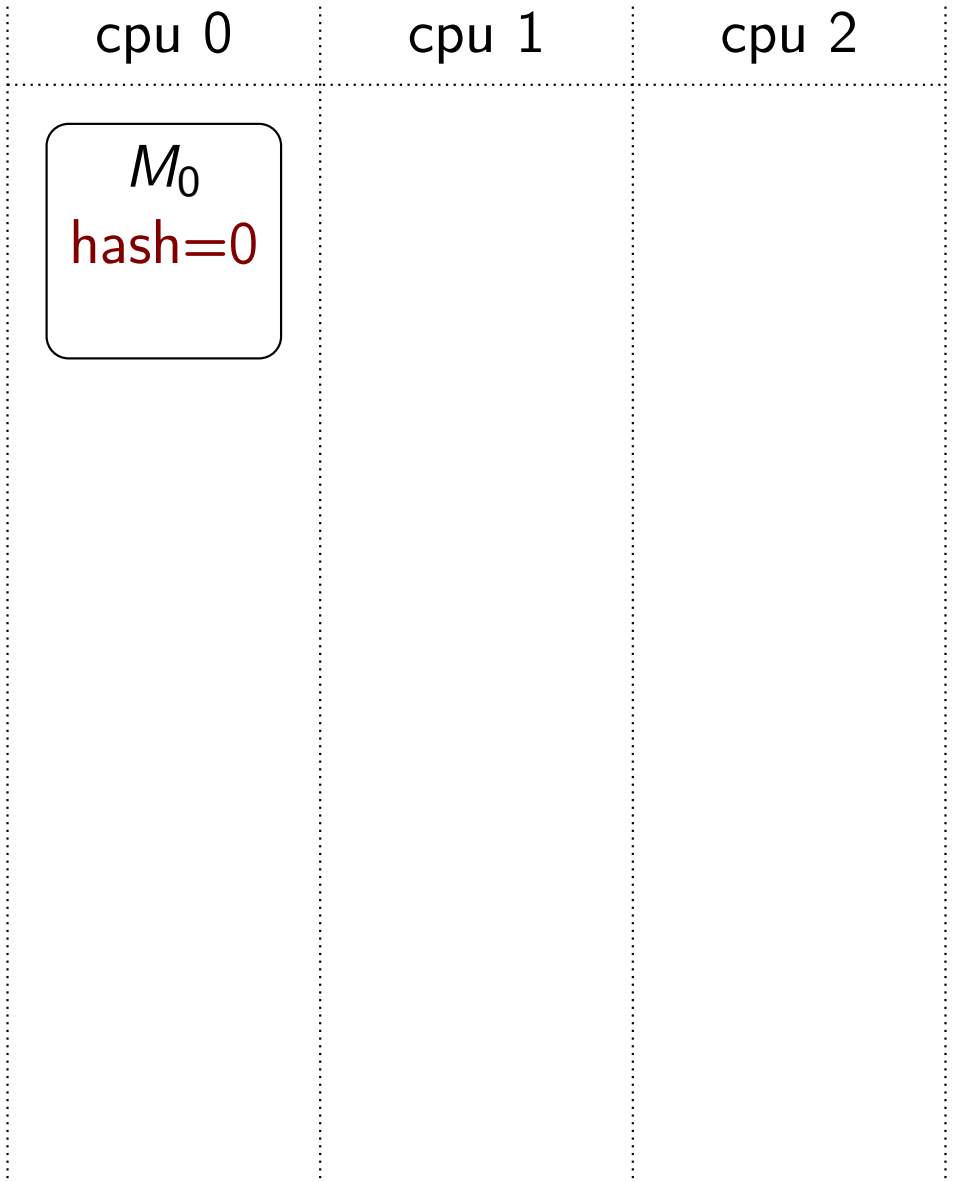
Verification issues
- Optimised compilation
- Parallel computation
- Process identifiers
- Clock counters

Conclusion

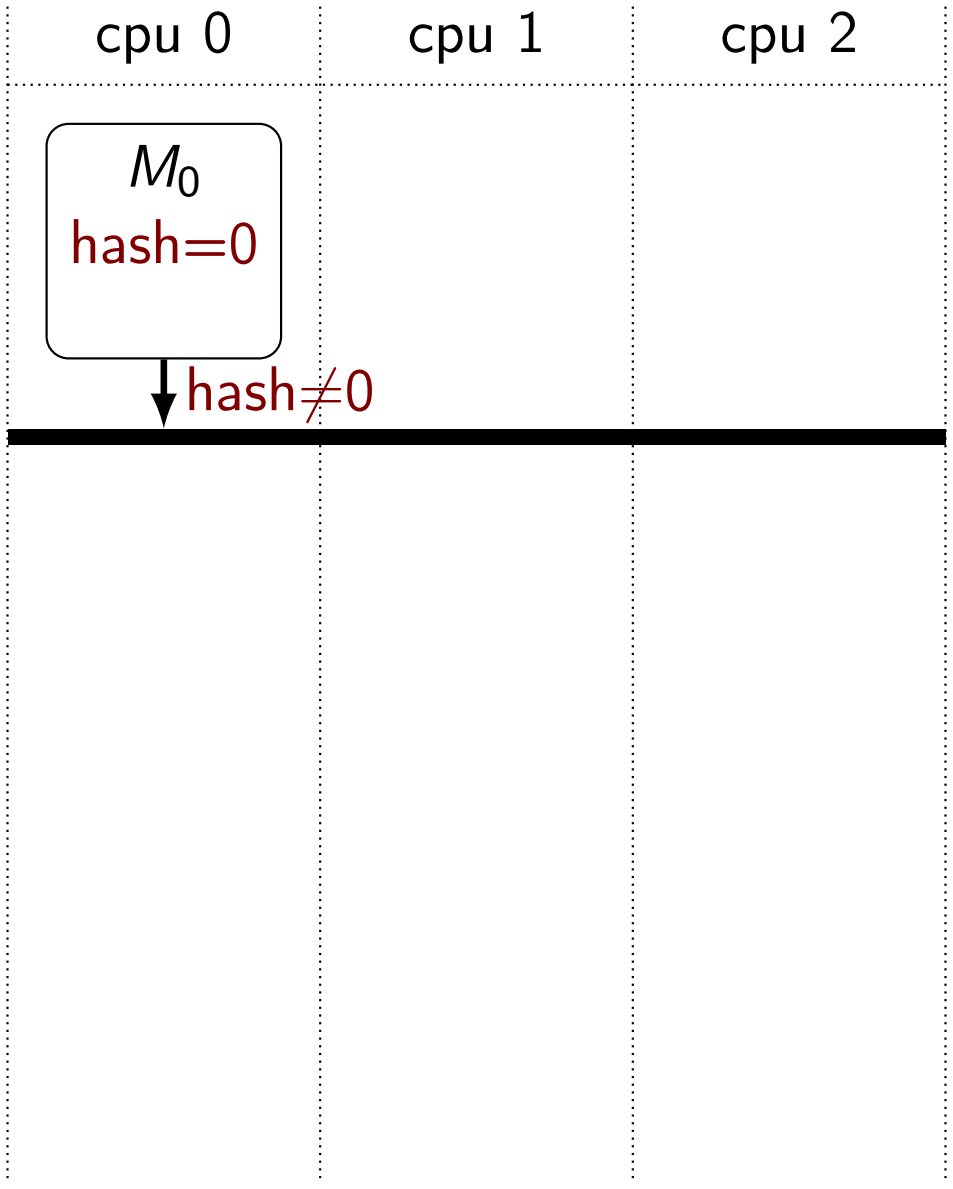
Computing the reachable markings in parallel

cpu 0	cpu 1	cpu 2
M_0		

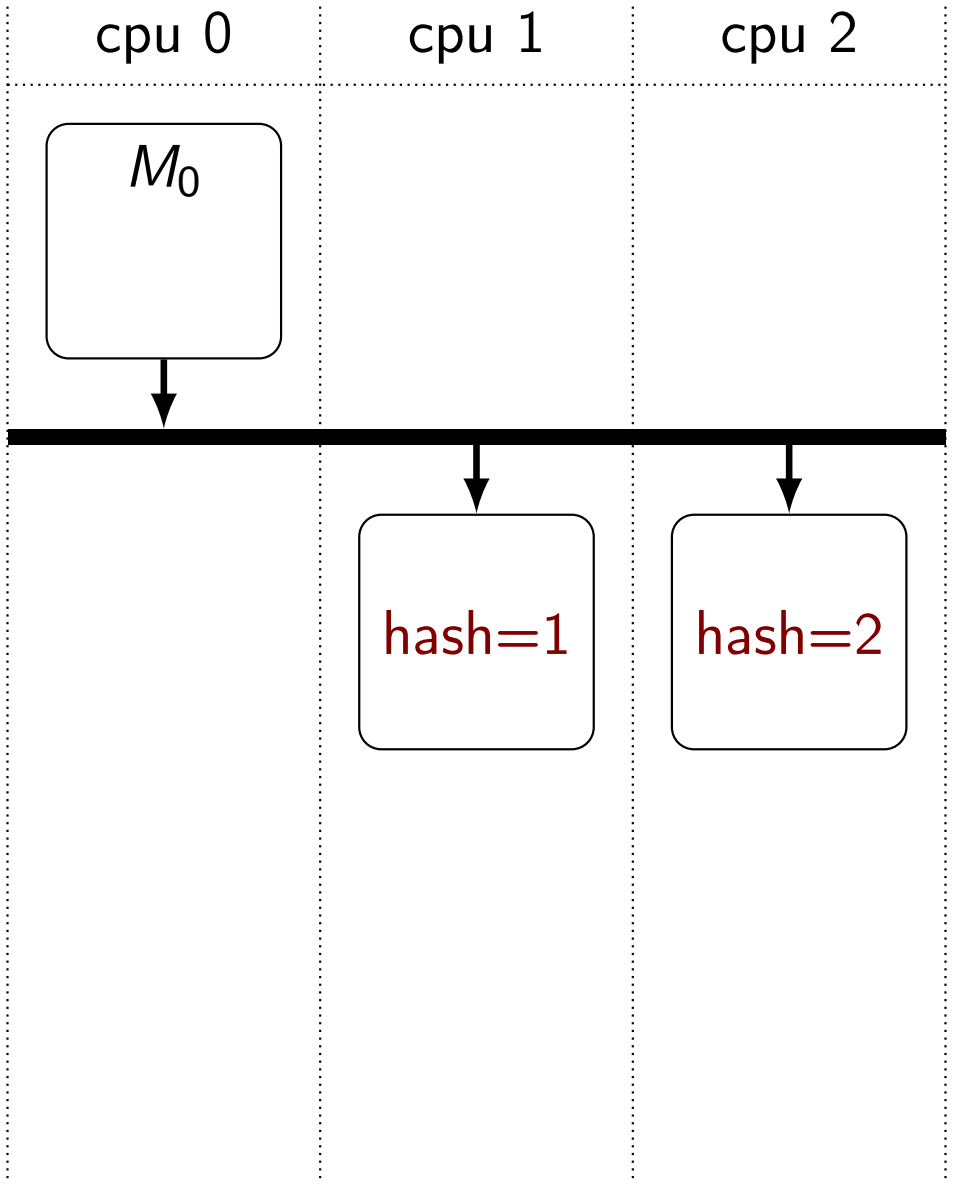
Computing the reachable markings in parallel



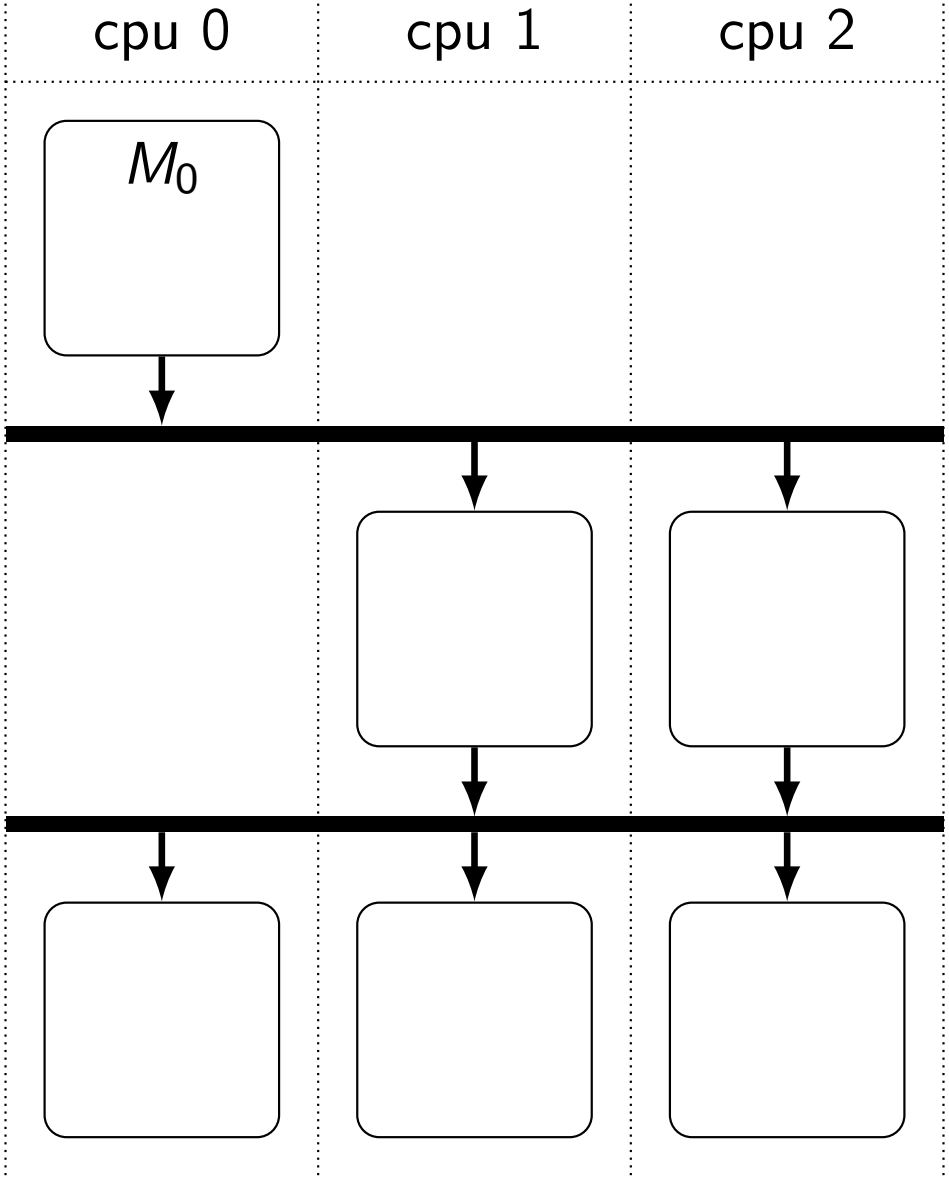
Computing the reachable markings in parallel



Computing the reachable markings in parallel



Computing the reachable markings in parallel

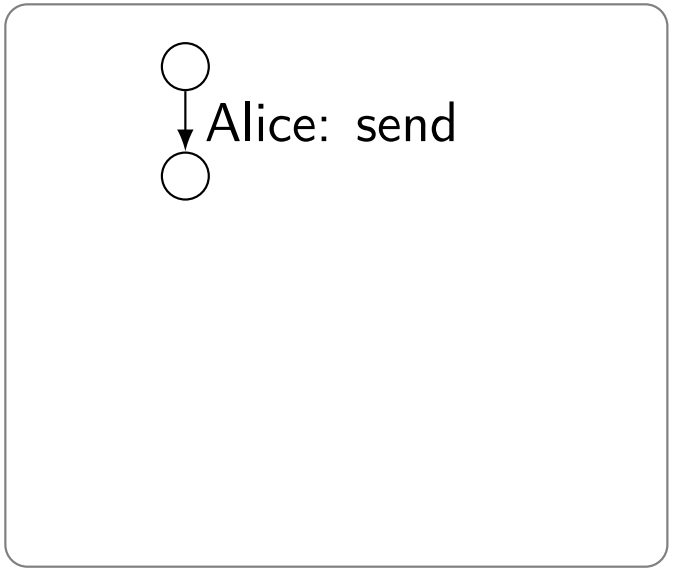


Improving the computation

- merge the two first steps
  - initially put M_0 at each cpu
- increase classes sizes
  - hash on a subset of places
- distribute enough work
  - chosen places must have enough different markings
- consider models structure: security protocols
  - hash after receive events
  - hash on accumulated received values
  - drop known states at each step

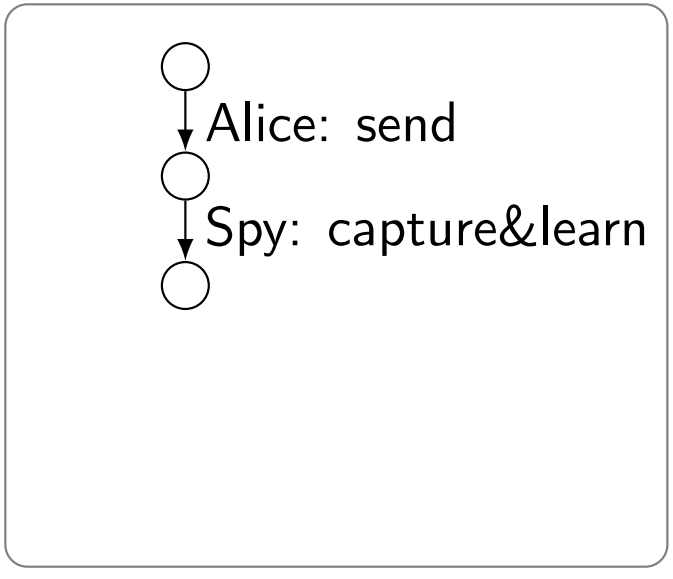
Improving the computation

- merge the two first steps
  - initially put M_0 at each cpu
- increase classes sizes
  - hash on a subset of places
- distribute enough work
  - chosen places must have enough different markings
- consider models structure: security protocols
  - hash after receive events
  - hash on accumulated received values
  - drop known states at each step



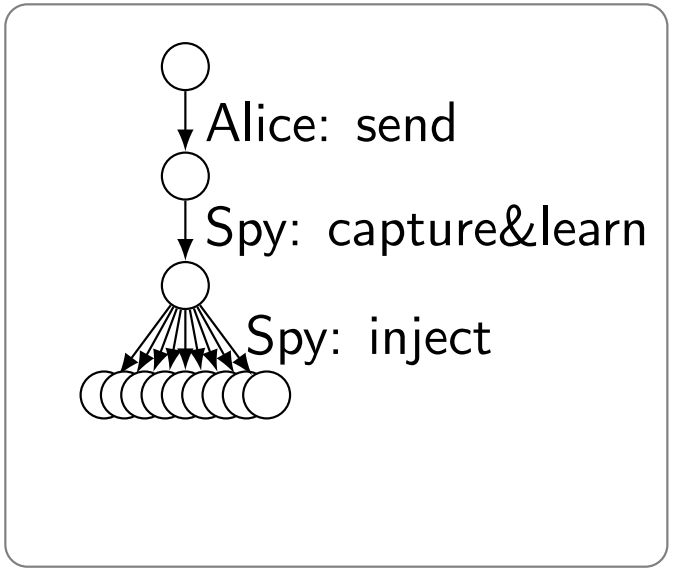
Improving the computation

- merge the two first steps
  - initially put M_0 at each cpu
- increase classes sizes
  - hash on a subset of places
- distribute enough work
  - chosen places must have enough different markings
- consider models structure: security protocols
  - hash after receive events
  - hash on accumulated received values
  - drop known states at each step



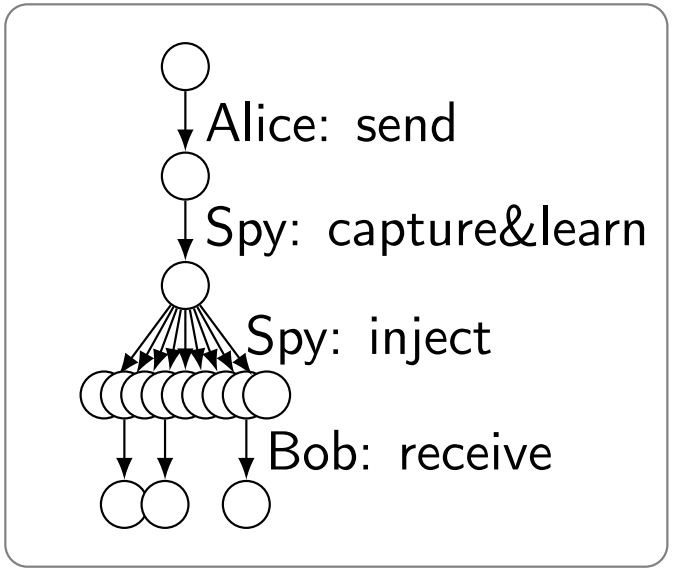
Improving the computation

- merge the two first steps
  - initially put M_0 at each cpu
- increase classes sizes
  - hash on a subset of places
- distribute enough work
  - chosen places must have enough different markings
- consider models structure: security protocols
  - hash after receive events
  - hash on accumulated received values
  - drop known states at each step



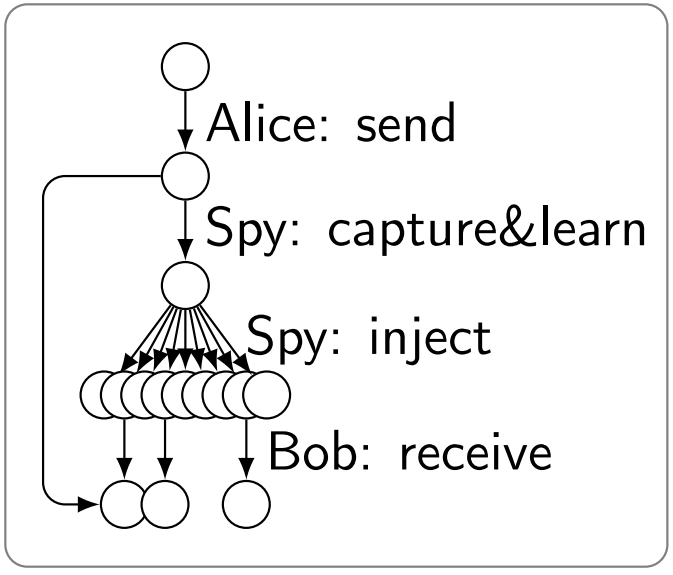
Improving the computation

- merge the two first steps
  - initially put M_0 at each cpu
- increase classes sizes
  - hash on a subset of places
- distribute enough work
  - chosen places must have enough different markings
- consider models structure: security protocols
  - hash after receive events
  - hash on accumulated received values
  - drop known states at each step



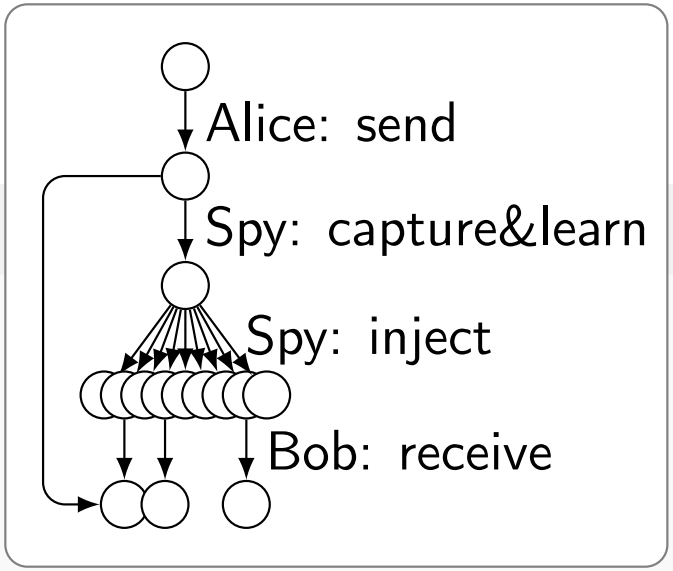
Improving the computation

- merge the two first steps
  - initially put M_0 at each cpu
- increase classes sizes
  - hash on a subset of places
- distribute enough work
  - chosen places must have enough different markings
- consider models structure: security protocols
  - hash after receive events
  - hash on accumulated received values
  - drop known states at each step



Improving the computation

- merge the two first steps
  - initially put M_0 at each cpu
- increase classes sizes
  - hash on a subset of places
- distribute enough work
  - chosen places must have enough different markings
- consider models structure: security protocols
  - hash after receive events
  - hash on accumulated received values
  - drop known states at each step



Experiments

- speedups from 2 to ∞
- much less communications
- work in progress

Outline

A modular framework

ABCD of modelling

More than simulation

Three application domains

Verification issues
- Optimised compilation
- Parallel computation
- Process identifiers
- Clock counters

Conclusion

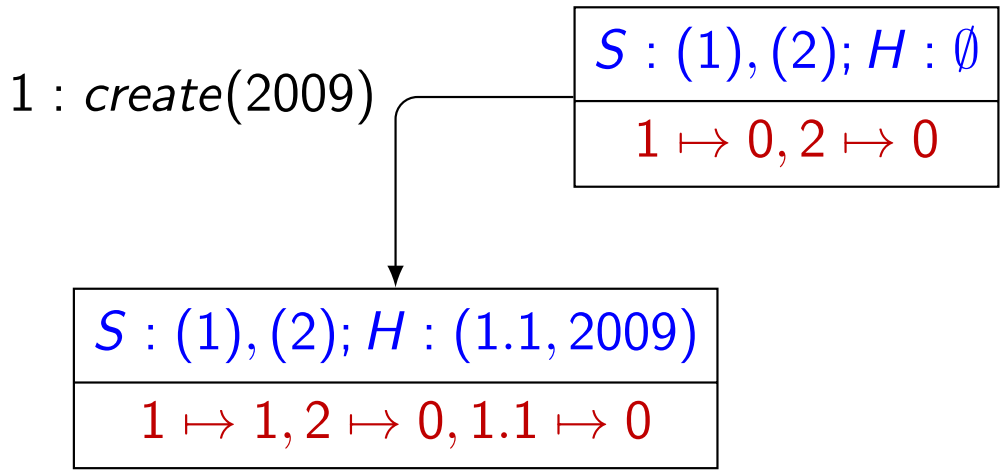
Verification of multi-threaded systems

Identifying equivalent states

$S : (1), (2); H : \emptyset$
$1 \mapsto 0, 2 \mapsto 0$

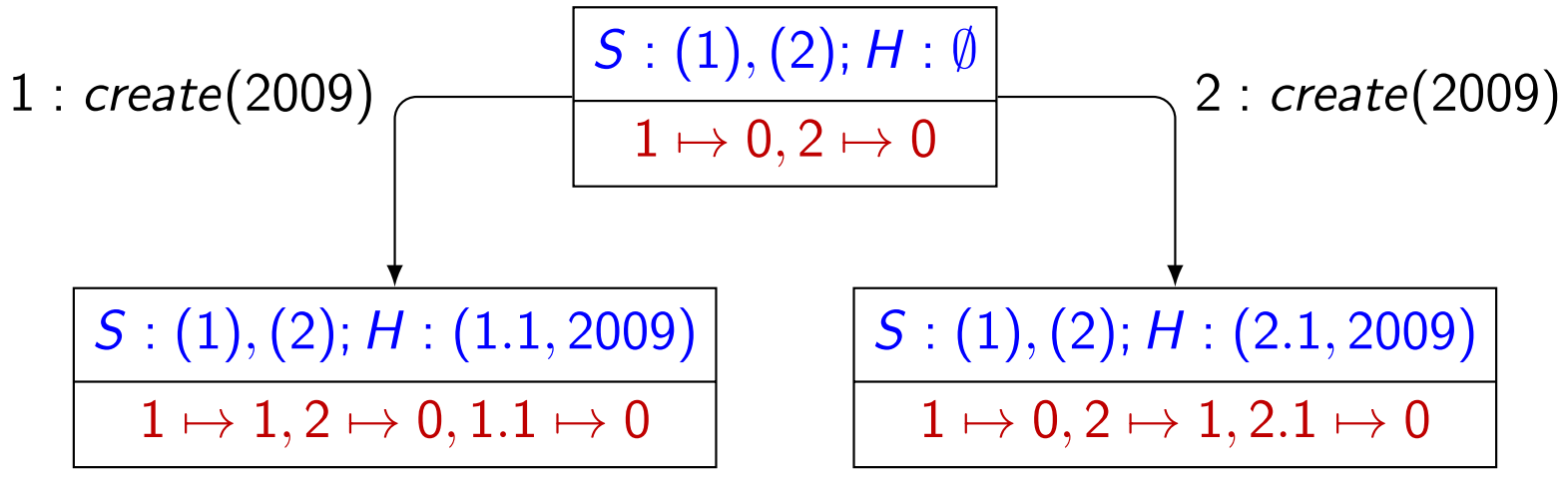
Verification of multi-threaded systems

Identifying equivalent states



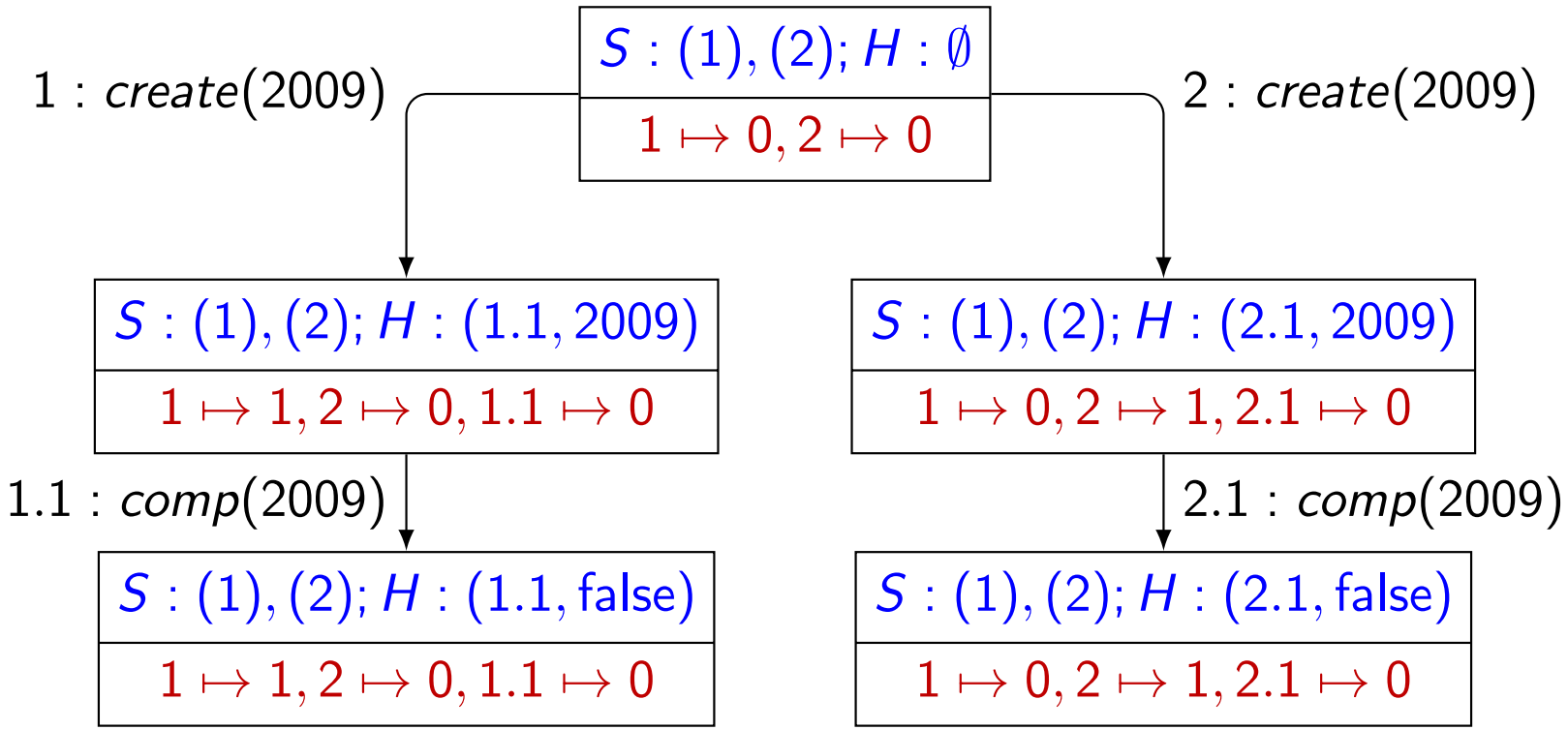
Verification of multi-threaded systems

Identifying equivalent states



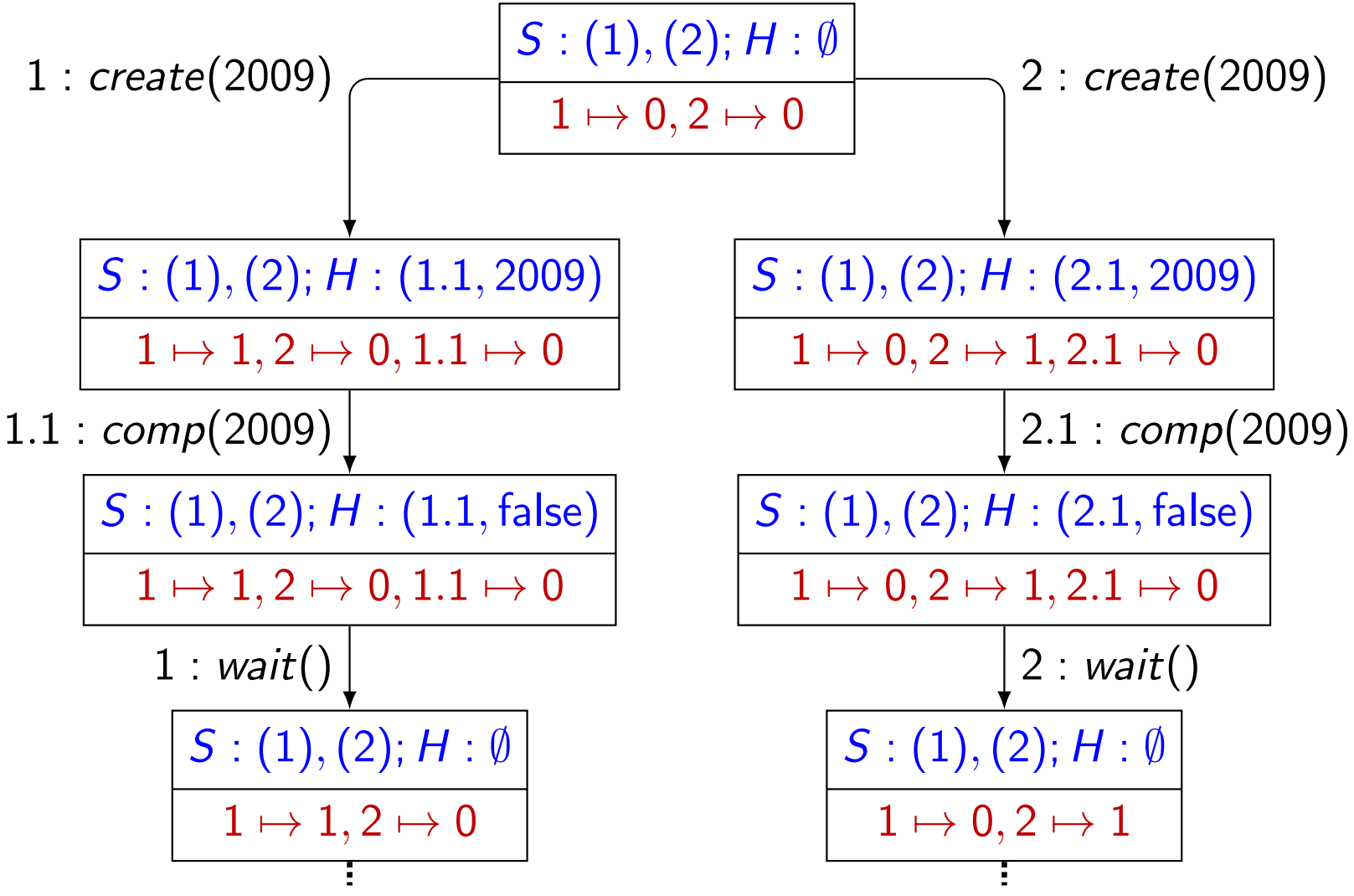
Verification of multi-threaded systems

Identifying equivalent states



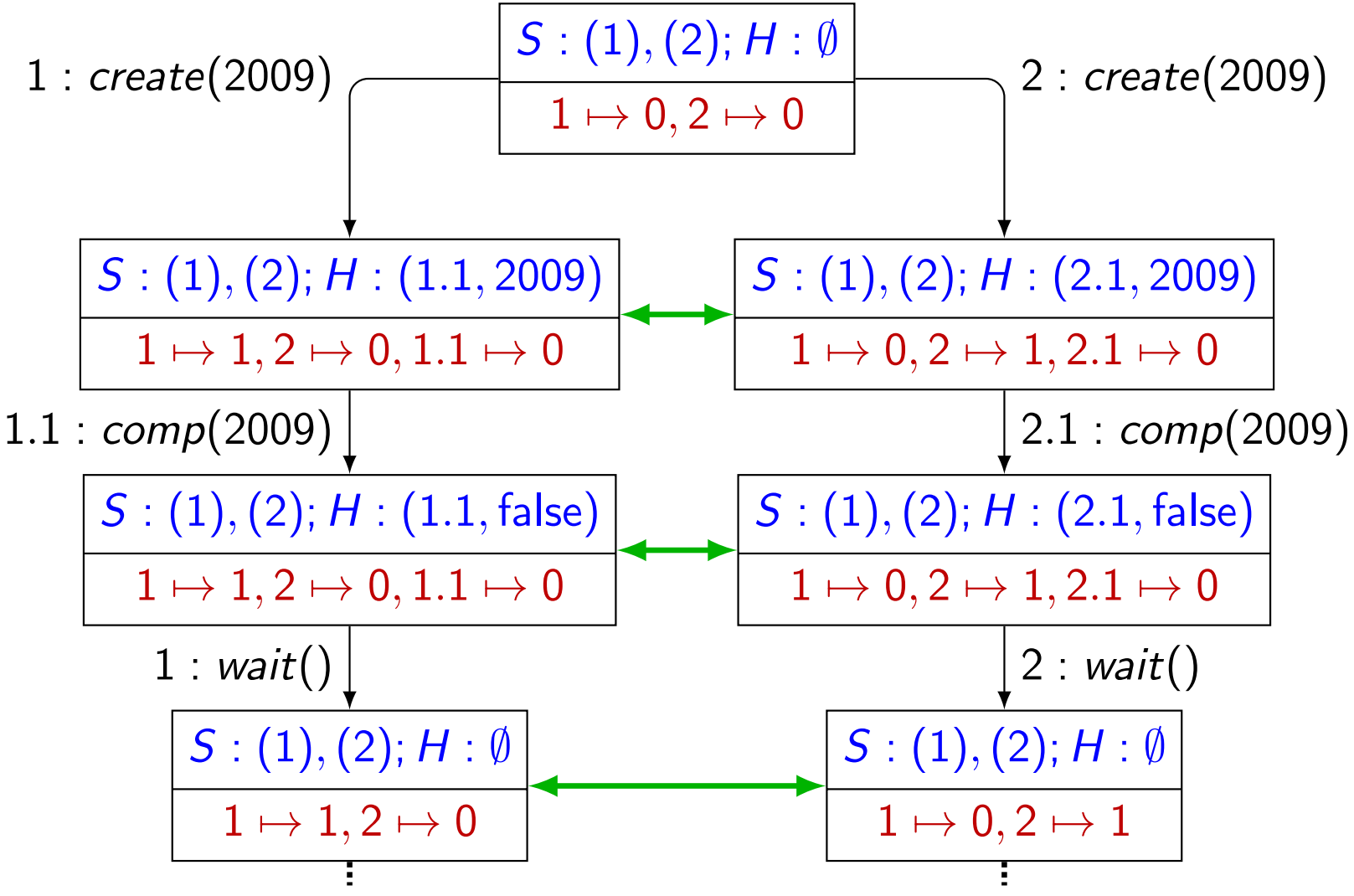
Verification of multi-threaded systems

Identifying equivalent states



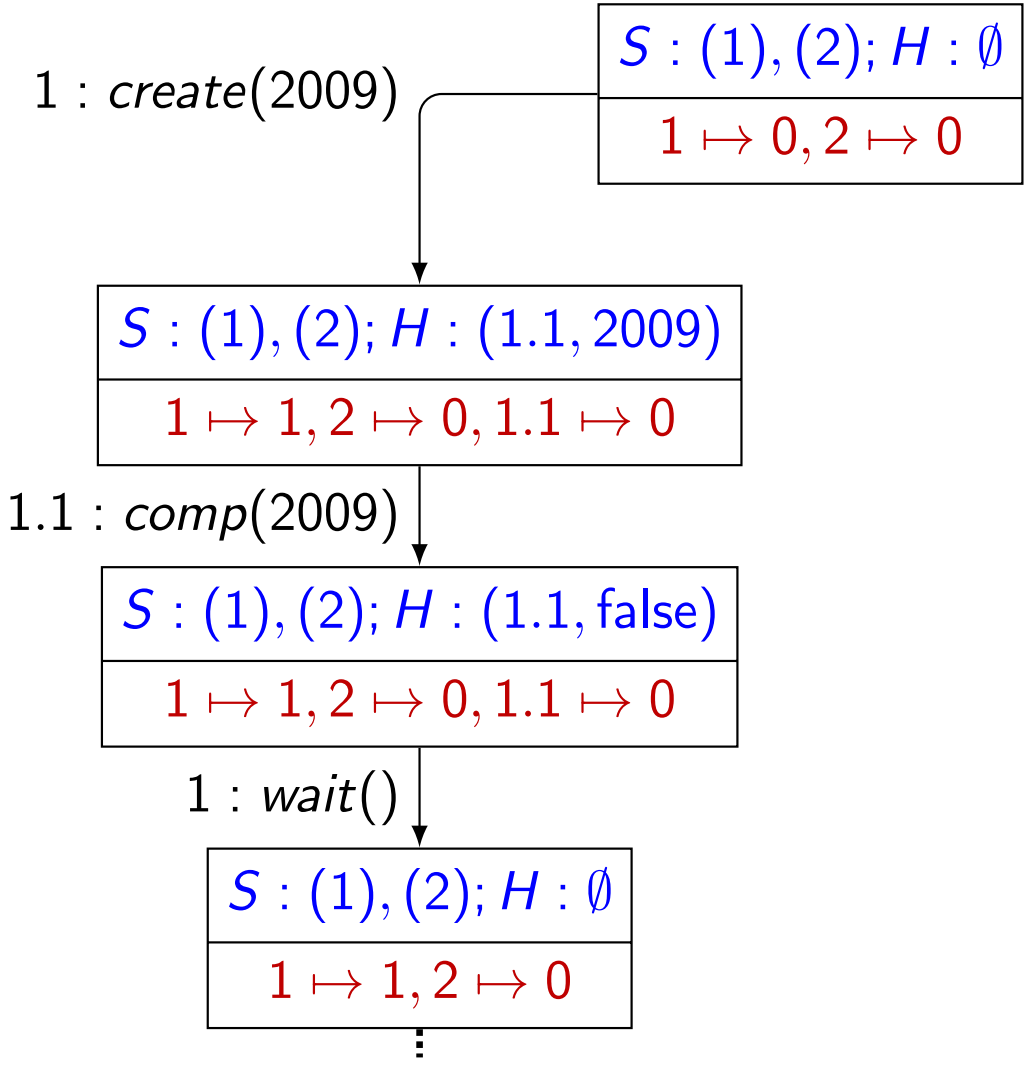
Verification of multi-threaded systems

Identifying equivalent states



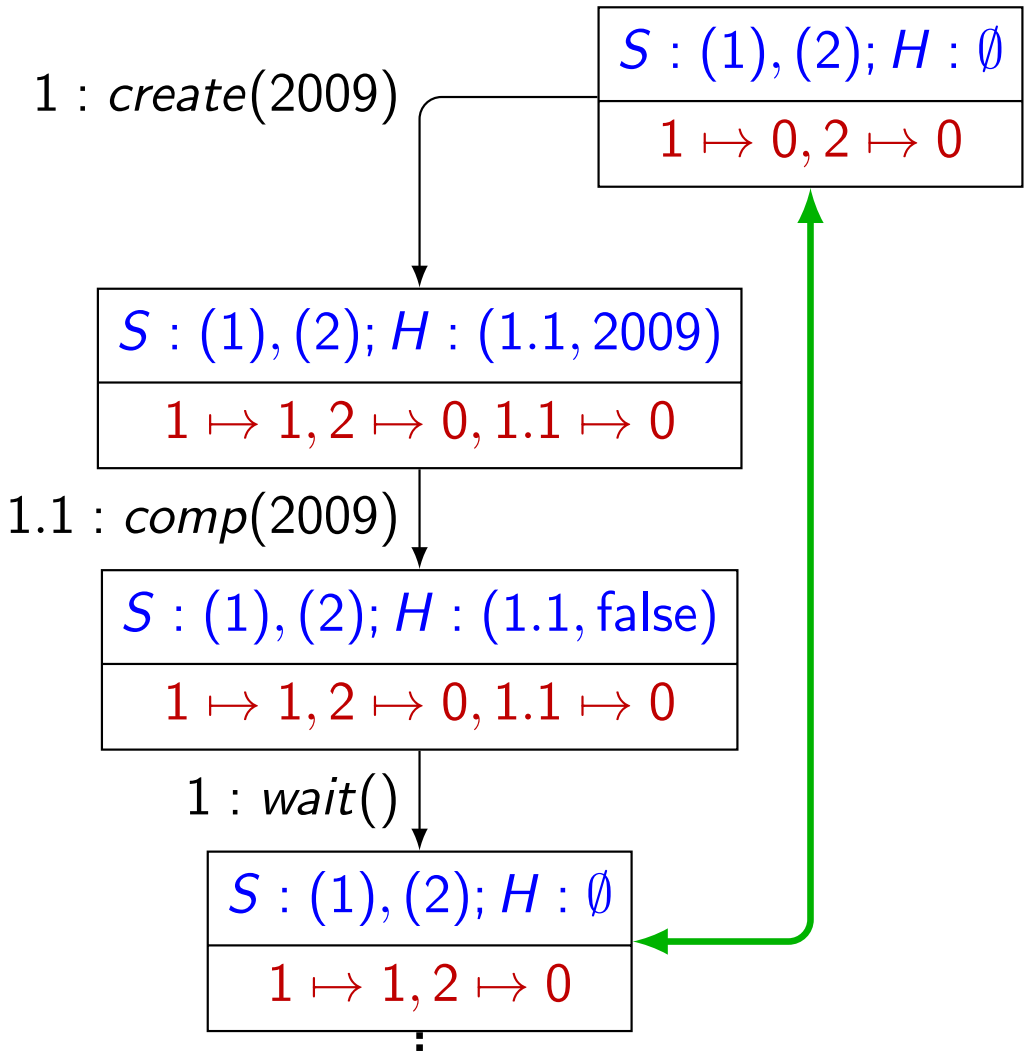
Verification of multi-threaded systems

Identifying equivalent states



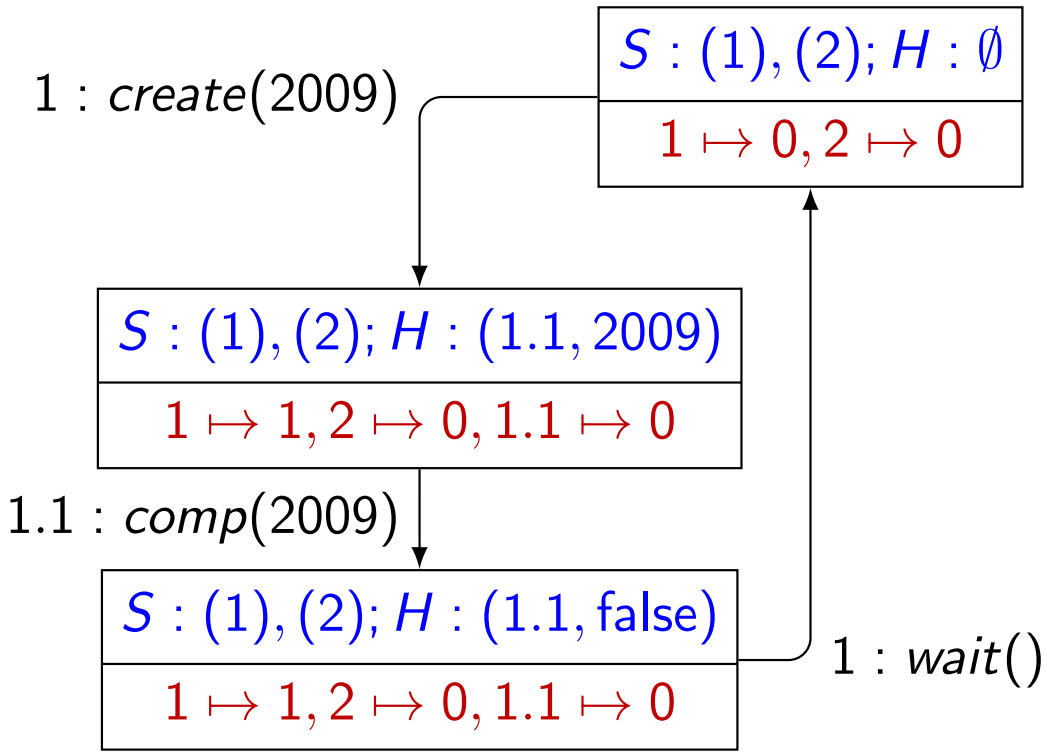
Verification of multi-threaded systems

Identifying equivalent states



Verification of multi-threaded systems

Identifying equivalent states



Checking state equivalence

$S : (1), (2); H : (1.1, \text{false})$
$1 \mapsto 1, 2 \mapsto 0, 1.1 \mapsto 0$

$S : (1), (2); H : (2.1, \text{false})$
$1 \mapsto 0, 2 \mapsto 1, 2.1 \mapsto 0$

Checking state equivalence

$S : (1), (2); H : (1.1, \text{false})$
$1 \mapsto 1, 2 \mapsto 0, 1.1 \mapsto 0$

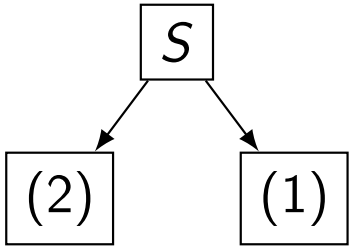
S H

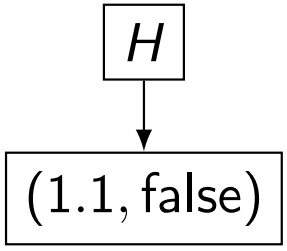
$S : (1), (2); H : (2.1, \text{false})$
$1 \mapsto 0, 2 \mapsto 1, 2.1 \mapsto 0$

S H

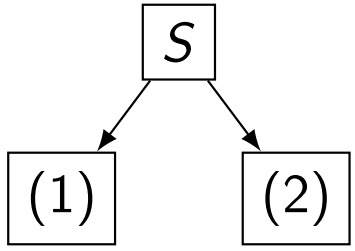
Checking state equivalence

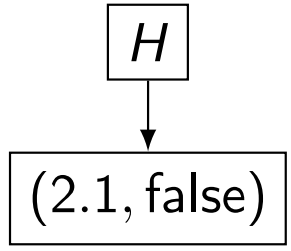
$S : (1), (2); H : (1.1, \text{false})$
$1 \mapsto 1, 2 \mapsto 0, 1.1 \mapsto 0$





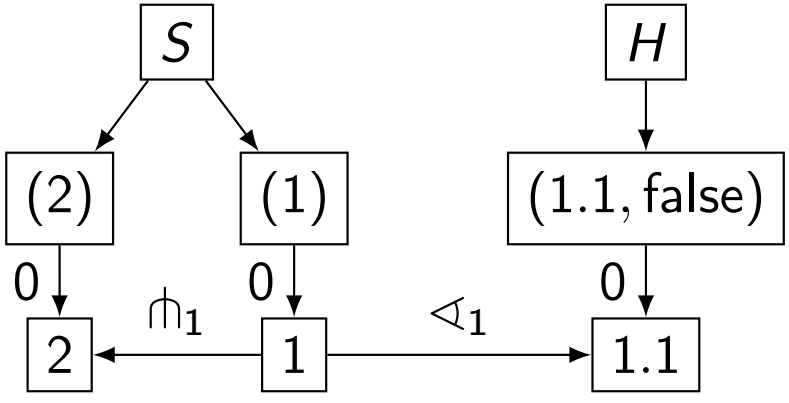
$S : (1), (2); H : (2.1, \text{false})$
$1 \mapsto 0, 2 \mapsto 1, 2.1 \mapsto 0$



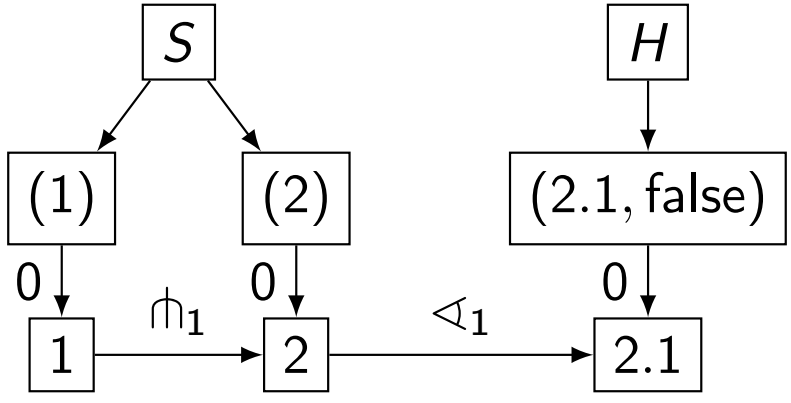


Checking state equivalence

$S : (1), (2); H : (1.1, \text{false})$
$1 \mapsto 1, 2 \mapsto 0, 1.1 \mapsto 0$

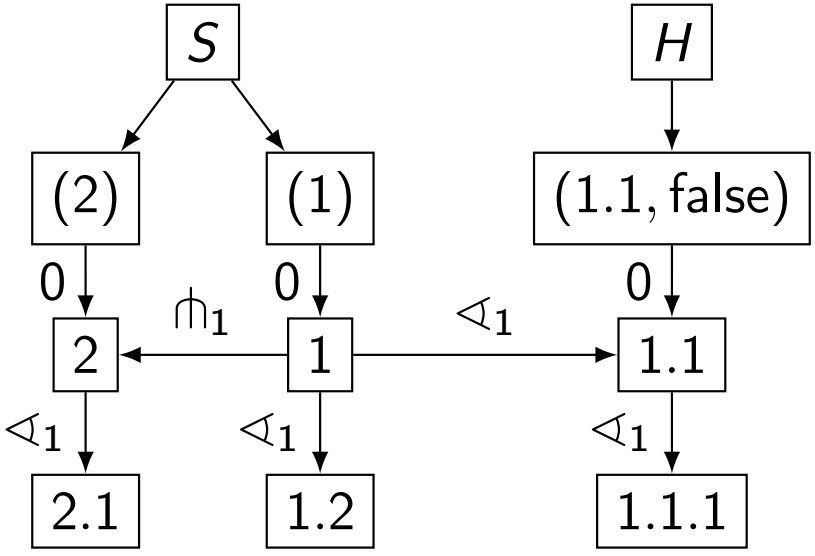


$S : (1), (2); H : (2.1, \text{false})$
$1 \mapsto 0, 2 \mapsto 1, 2.1 \mapsto 0$

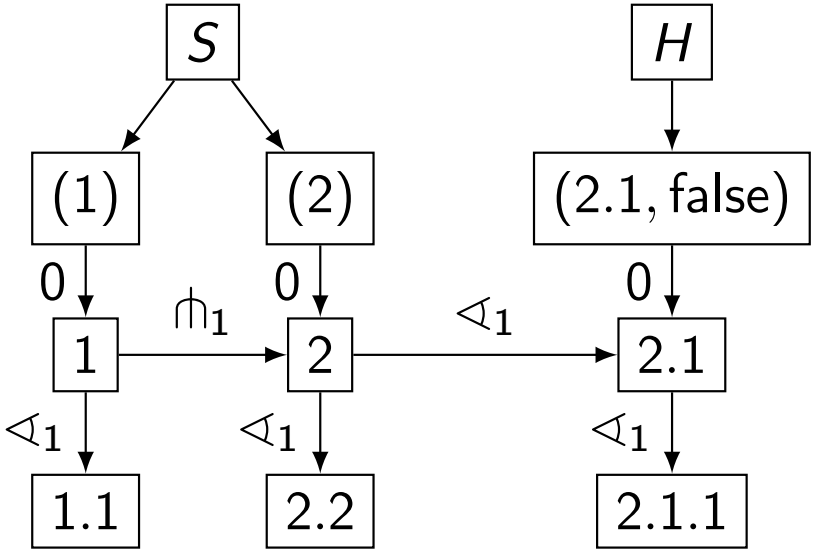


Checking state equivalence

$S : (1), (2); H : (1.1, \text{false})$
$1 \mapsto 1, 2 \mapsto 0, 1.1 \mapsto 0$



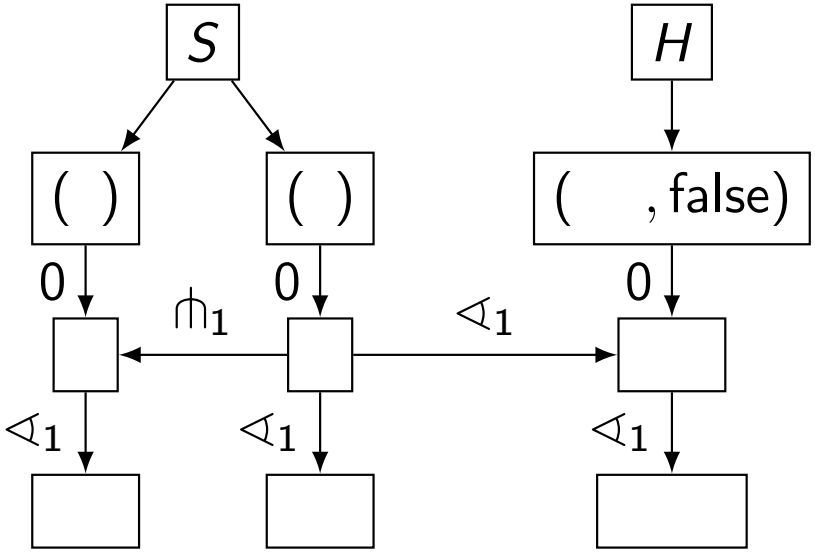
$S : (1), (2); H : (2.1, \text{false})$
$1 \mapsto 0, 2 \mapsto 1, 2.1 \mapsto 0$

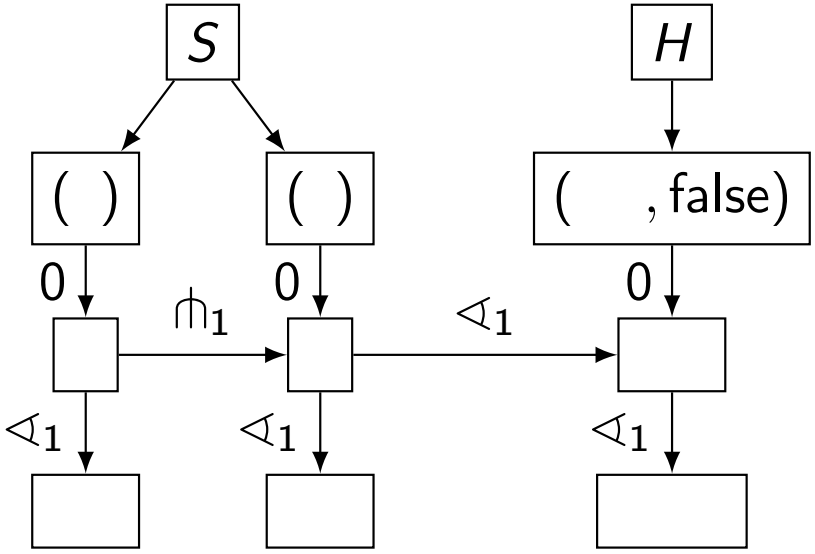


Checking state equivalence

$S : (1), (2); H : (1.1, \text{false})$
$1 \mapsto 1, 2 \mapsto 0, 1.1 \mapsto 0$

$S : (1), (2); H : (2.1, \text{false})$
$1 \mapsto 0, 2 \mapsto 1, 2.1 \mapsto 0$

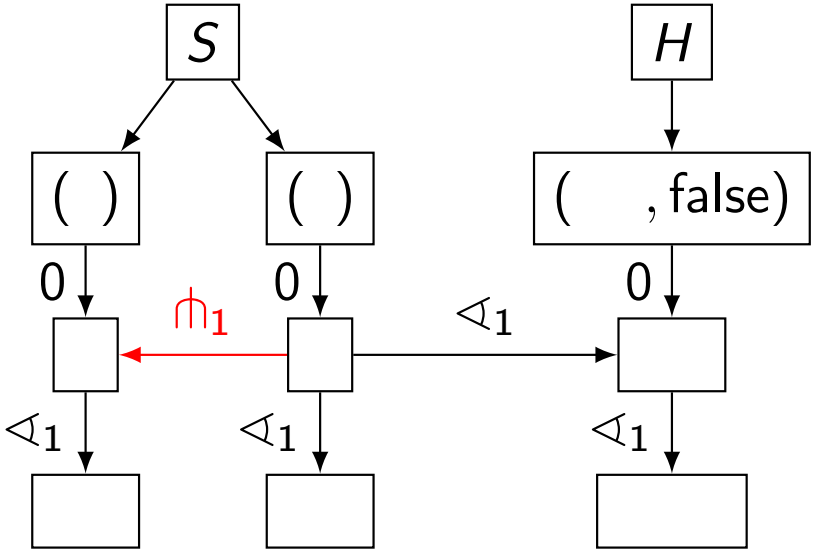


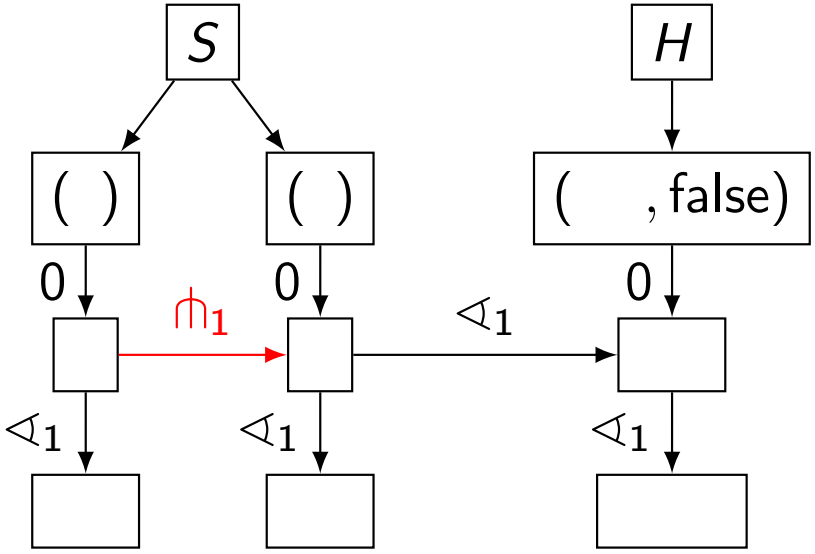


Checking state equivalence

$S : (1), (2); H : (1.1, \text{false})$
$1 \mapsto 1, 2 \mapsto 0, 1.1 \mapsto 0$

$S : (1), (2); H : (2.1, \text{false})$
$1 \mapsto 0, 2 \mapsto 1, 2.1 \mapsto 0$

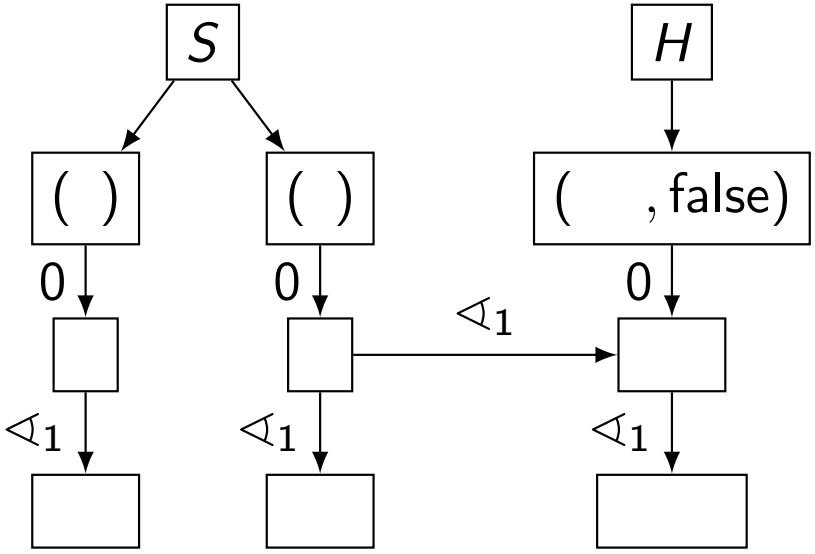


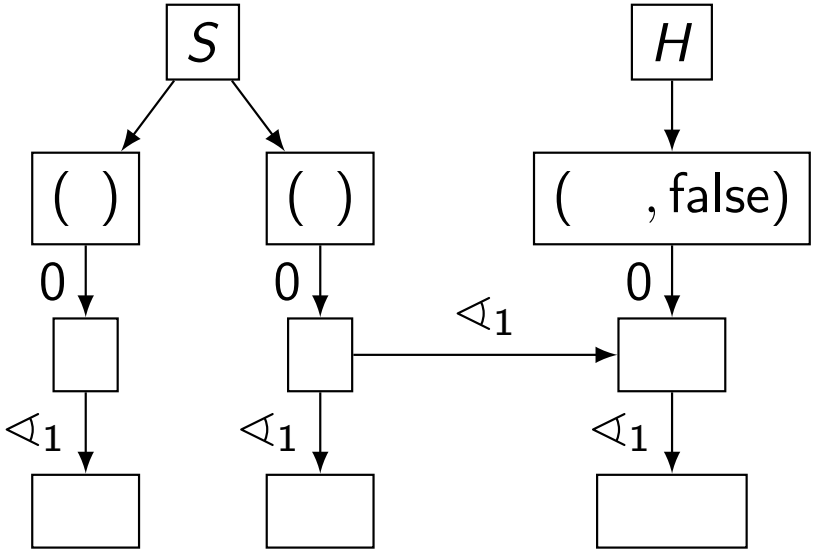


Checking state equivalence

$S : (1), (2); H : (1.1, \text{false})$
$1 \mapsto 1, 2 \mapsto 0, 1.1 \mapsto 0$

$S : (1), (2); H : (2.1, \text{false})$
$1 \mapsto 0, 2 \mapsto 1, 2.1 \mapsto 0$





Evaluating the approach

- consistent abstraction along traces
  - compatible with unfolding
- subsumes symmetry reduction
- graph isomorphism has unknown complexity
  - symmetry reduction is as hard as graph isomorphism
- two millions random states tested using Nauty and VF2
  - time grows with % of pids vs data
  - time grows with $p \stackrel{\text{df}}{=}$ number of distinct pids
    looks $O(p)$ with Nauty and $O(p \log p)$ with VF2
  - time looks $O(n^2)$ where $n \stackrel{\text{df}}{=}$ number of distinct tokens

Outline

A modular framework

ABCD of modelling

More than simulation

Three application domains

Verification issues

- Optimised compilation
- Parallel computation
- Process identifiers
- Clock counters

Conclusion

Efficient representation of counters

Exploiting dedicated libraries

- extract counters from the Petri net
  - (S, T, ℓ, X, L)
  - X: counters
  - $L \stackrel{\text{df}}{=} (C, U)$: condition and update for each $t \in T$
  - dual states: (M, v) with $v \in \mathbb{Z}^X$
  - $(M, v)\,[t, \beta\rangle (M', U_t(v))$ iff $M\,[t, \beta\rangle\, M'$ and $C_t(v)$

Efficient representation of counters

Exploiting dedicated libraries

- extract counters from the Petri net
  - (S, T, ℓ, X, L)
  - X: counters
  - $L \stackrel{\text{df}}{=} (C, U)$: condition and update for each $t \in T$
  - dual states: (M, v) with $v \in \mathbb{Z}^X$
  - $(M, v)\,[t, \beta\rangle (M', U_t(v))$ iff $M\,[t, \beta\rangle\, M'$ and $C_t(v)$
- consider sets of states (possibly infinite)
  - $(M, V)\,[t, \beta\rangle (M', L_t(V))$ iff $M\,[t, \beta\rangle\, M'$ and

$$L_t(V) \stackrel{\text{df}}{=} \{U_t(v) \mid C_t(v) \wedge v \in V\} \neq \emptyset$$

Efficient representation of counters

Exploiting dedicated libraries

- extract counters from the Petri net
  - (S, T, ℓ, X, L)
  - X: counters
  - $L \stackrel{\text{df}}{=} (C, U)$: condition and update for each $t \in T$
  - dual states: (M, v) with $v \in \mathbb{Z}^X$
  - $(M, v)\,[t, \beta\rangle (M', U_t(v))$ iff $M\,[t, \beta\rangle\, M'$ and $C_t(v)$
- consider sets of states (possibly infinite)
  - $(M, V)\,[t, \beta\rangle (M', L_t(V))$ iff $M\,[t, \beta\rangle\, M'$ and

$$L_t(V) \stackrel{\text{df}}{=} \{U_t(v) \mid C_t(v) \wedge v \in V\} \neq \emptyset$$

- identify cycles in the behaviour and compress them
  - meta-transitions

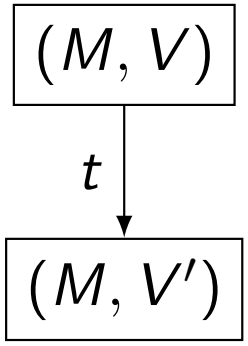
Compressing cycles

saturate loops

$\boxed{(M, V)}$

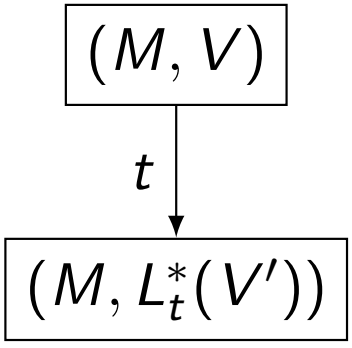
Compressing cycles

saturate loops



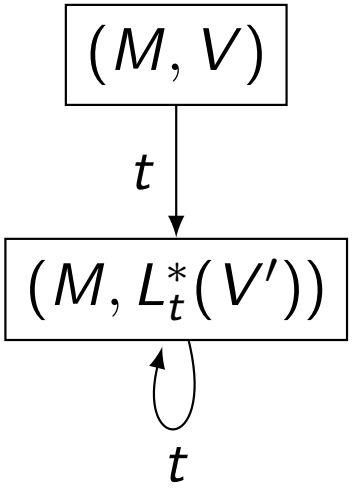
Compressing cycles

saturate loops



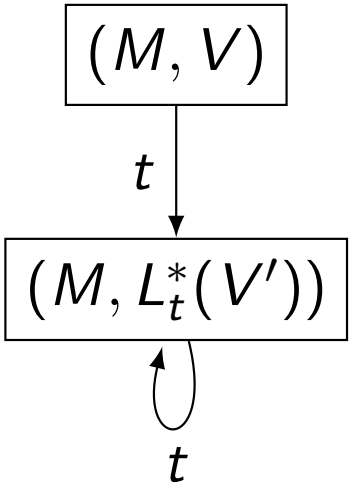
Compressing cycles

saturate loops

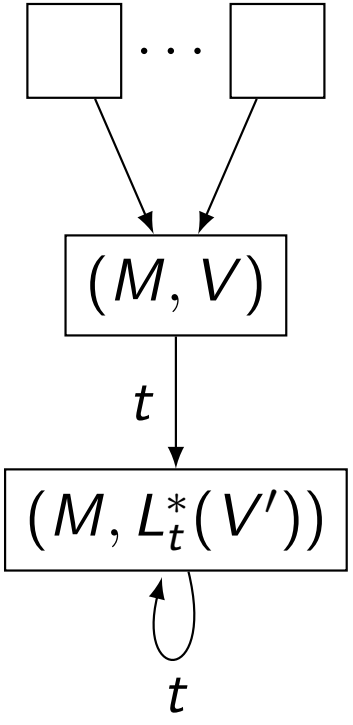


Compressing cycles

saturate loops

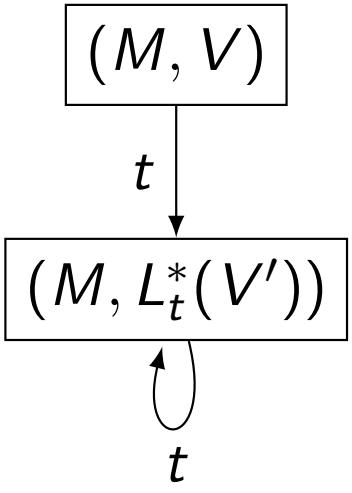


fold loops

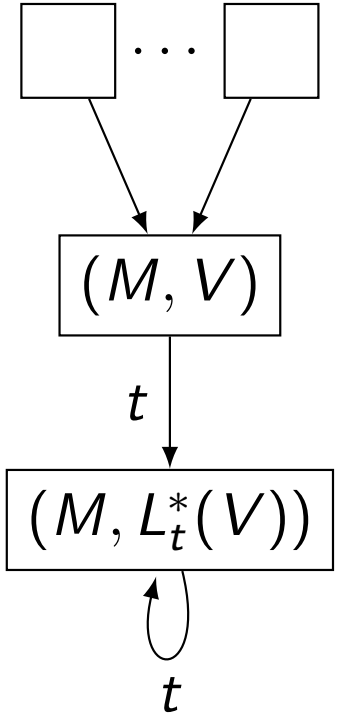


Compressing cycles

saturate loops

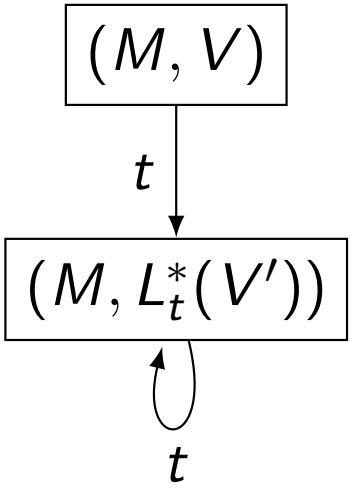


fold loops

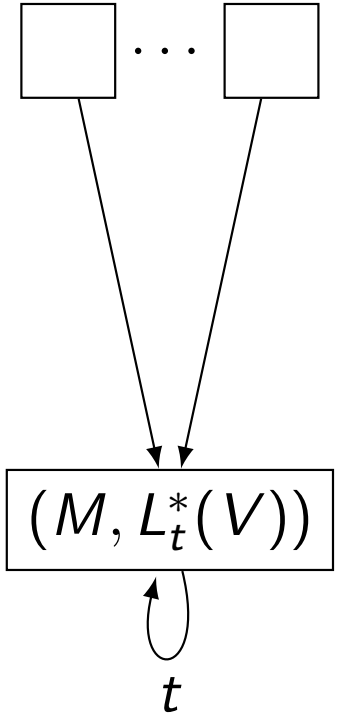


Compressing cycles

saturate loops

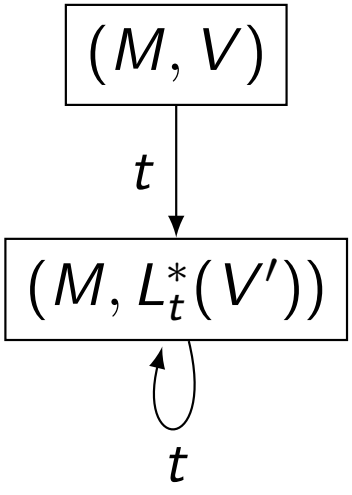


fold loops

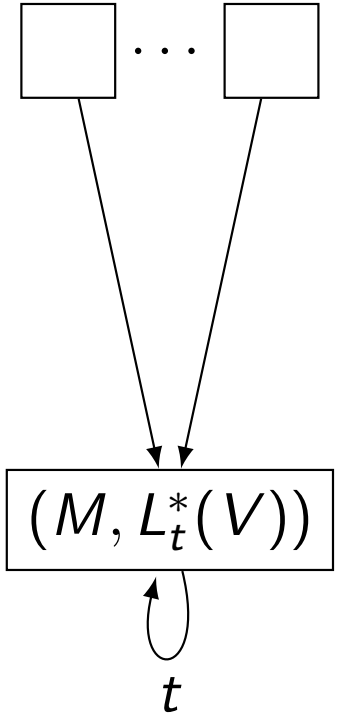


Compressing cycles

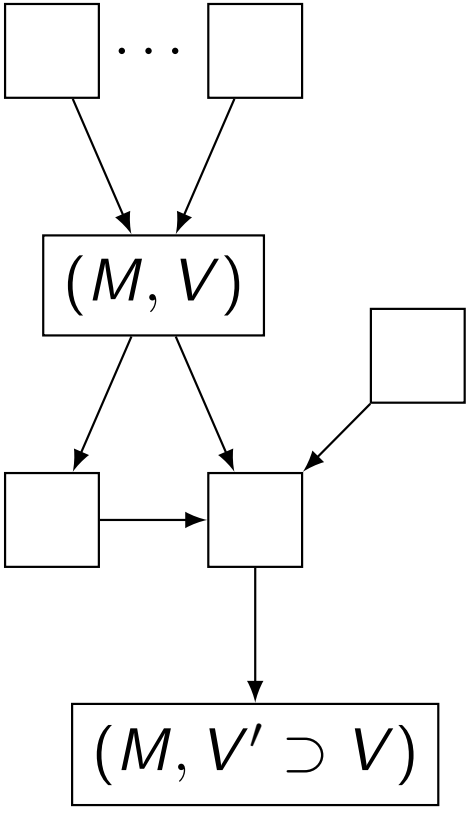
saturate loops



fold loops

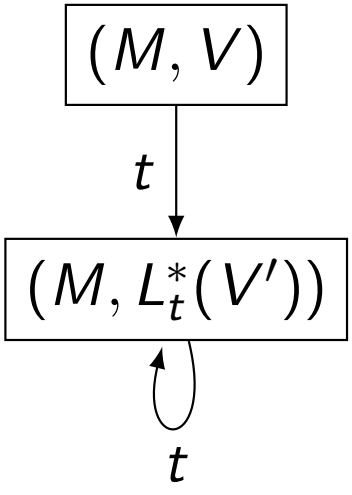


saturate cycles

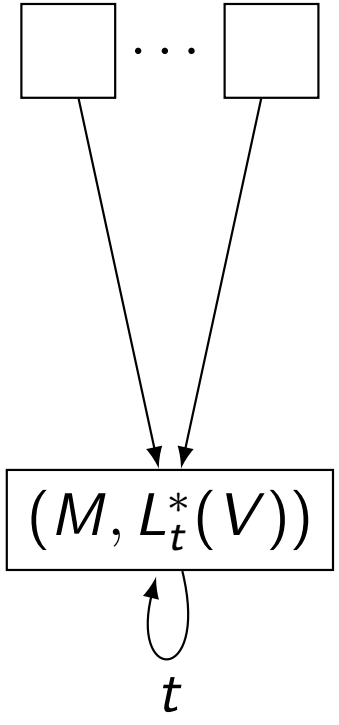


Compressing cycles

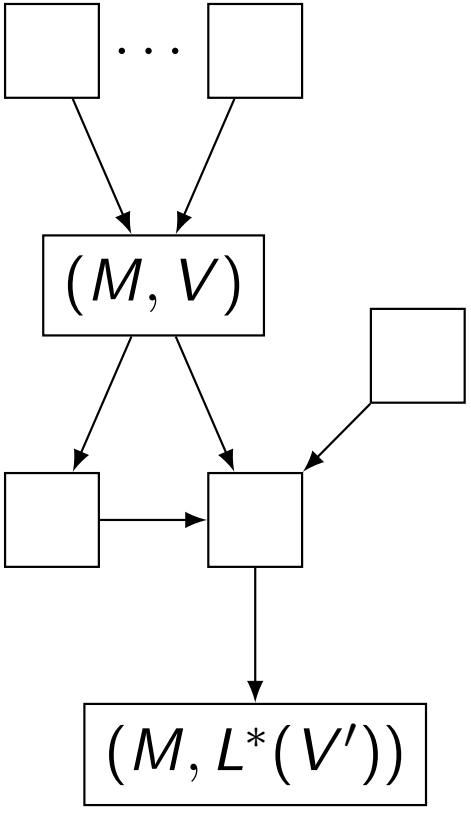
saturate loops



fold loops

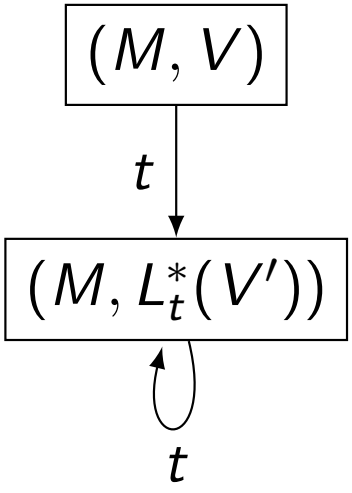


saturate cycles

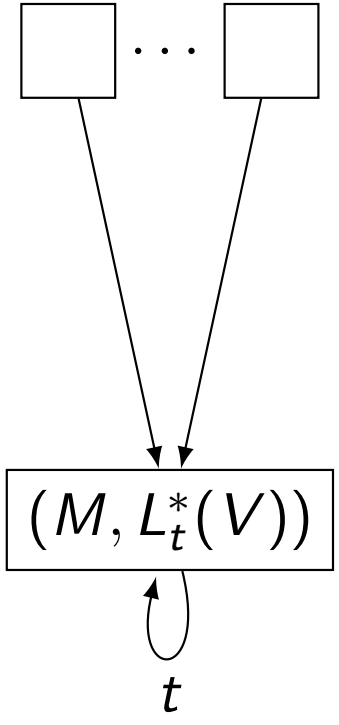


Compressing cycles

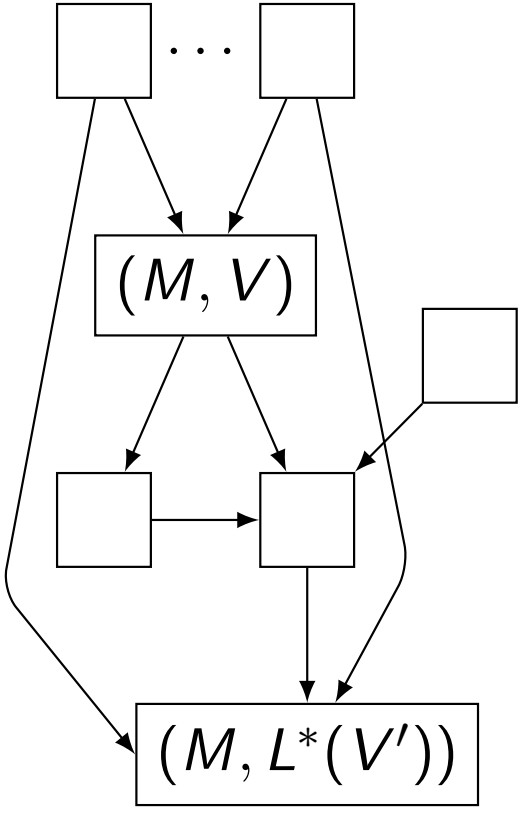
saturate loops



fold loops

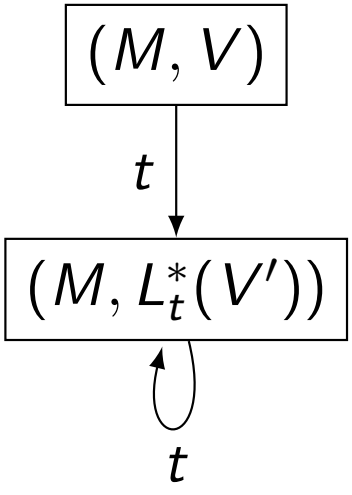


saturate cycles

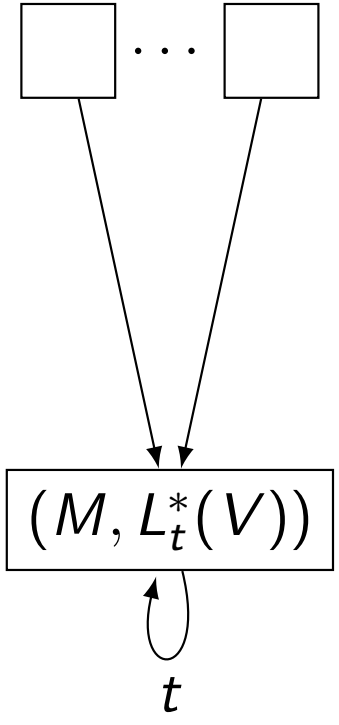


Compressing cycles

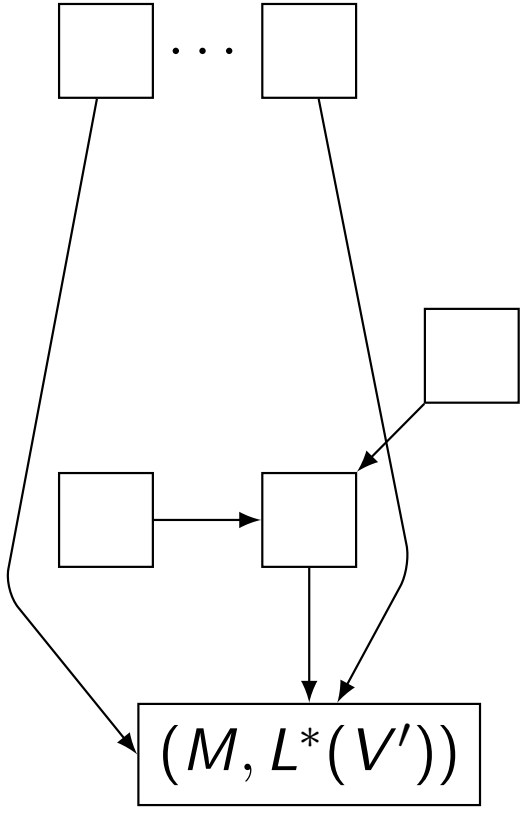
saturate loops



fold loops

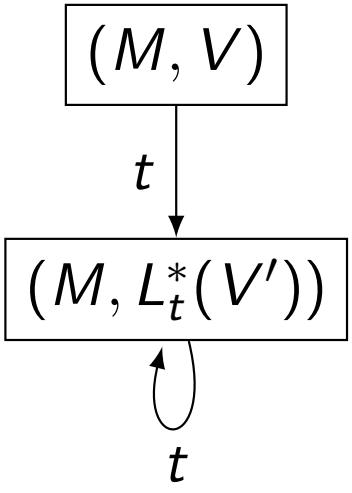


saturate cycles

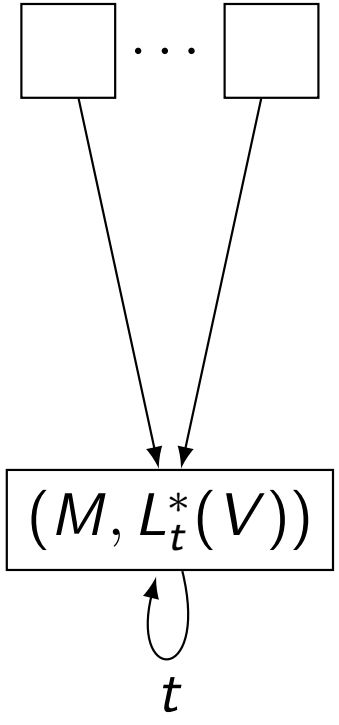


Compressing cycles

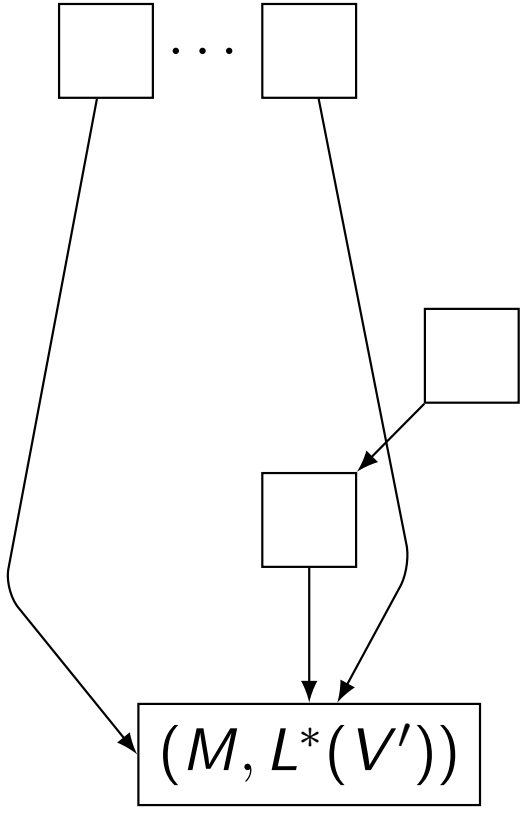
saturate loops



fold loops



saturate cycles



Compressed state spaces

- exact reachability
- over-approximates traces
  - abstract traces can be checked for realisability
  - on-the-fly and on the abstract model
- does not guarantee termination
  - worked with causal time examples (± 1)
- implemented with Lash within SNAKES
  - other tools should be usable
    FAST, MONA, Omega, SATAF/PresTaf, ...

Outline

A modular framework

ABCD of modelling

More than simulation

Three application domains

Verification issues

Conclusion

Summary

- a modular framework of coloured Petri nets
- various composition operators
  - using special annotations...
  - ... that can be discarded
- a user friendly syntax
  - ABCD
- easy to use tools
  - simulation
  - model-checking
- varied applications
- dedicated verification techniques

Conclusion

You can:

- pretend you make no error
- hope that your system will never crash

Conclusion

You can:

- pretend you make no error
- hope that your system will never crash

Or:

- known that everybody makes errors
  - including yourself
- use adequate tools to avoid them
- coloured Petri nets and variants may help
  - sophisticated and smart methods
  - easy to use tools
  - no excuse!

Conclusion

theory

applications

implementation

You can:

- pretend you make no error
- hope that your system will never crash

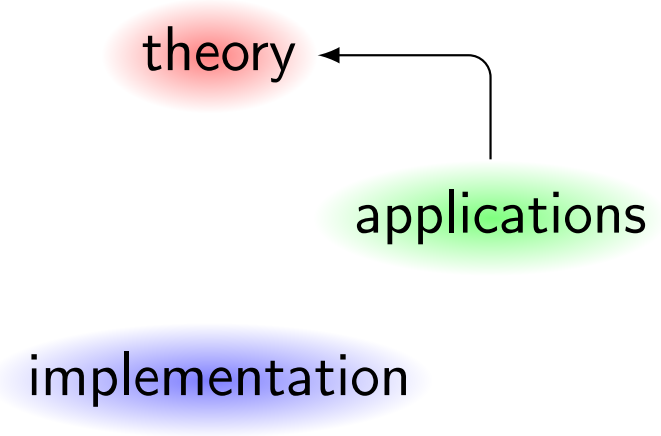
Or:

- known that everybody makes errors
  - including yourself
- use adequate tools to avoid them
- coloured Petri nets and variants may help
  - sophisticated and smart methods
  - easy to use tools
  - no excuse!

Conclusion

You can:

- pretend you make no error
- hope that your system will never crash



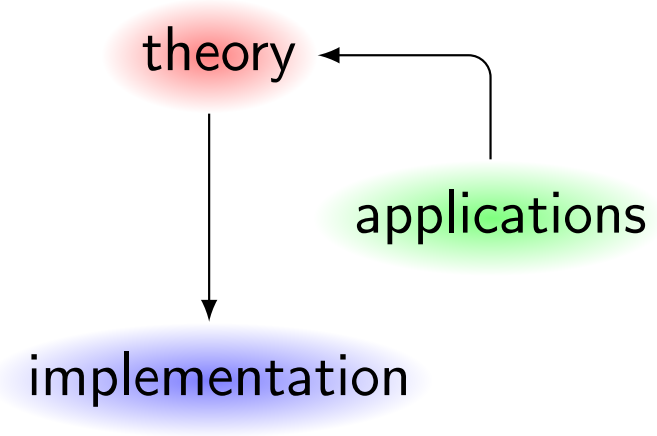
Or:

- known that everybody makes errors
  - including yourself
- use adequate tools to avoid them
- coloured Petri nets and variants may help
  - sophisticated and smart methods
  - easy to use tools
  - no excuse!

Conclusion

You can:

- pretend you make no error
- hope that your system will never crash



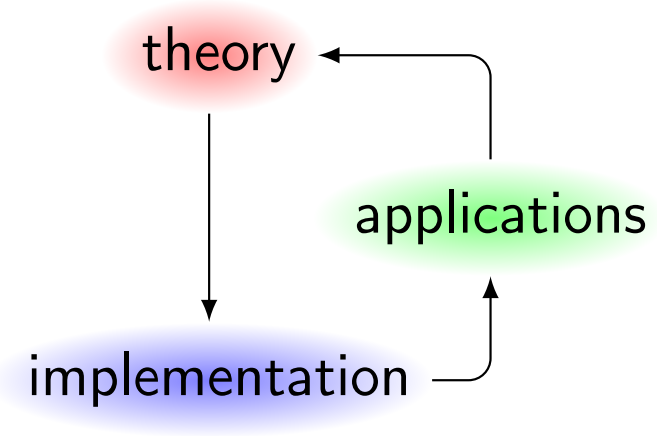
Or:

- known that everybody makes errors
  - including yourself
- use adequate tools to avoid them
- coloured Petri nets and variants may help
  - sophisticated and smart methods
  - easy to use tools
  - no excuse!

Conclusion

You can:

- pretend you make no error
- hope that your system will never crash



Or:

- known that everybody makes errors
  - including yourself
- use adequate tools to avoid them
- coloured Petri nets and variants may help
  - sophisticated and smart methods
  - easy to use tools
  - no excuse!