

# Teaching Concepts of Compositional Concurrency with State Machines

Antti Valmari

Tampere University of Technology

2006–06–27

# Contents

1	Introduction	3
2	A system consisting of state machines	8
3	Communication and interaction	11
4	Formal definition of state machines	15
5	More non-abstract compositionality results ...	22
6	Last non-abstract topics	25
7	Conclusions	26

# 1 Introduction

- Two well-know process algebras
    - **CSP** = Communicating Sequential Processes  
[Hoare, Roscoe, ... 1984–]
    - **CCS** = Calculus of Communicating Systems [Milner 1979–]
  - Process algebras feature **full abstraction**
    - Implementation-independent “what”-level representation of behaviour
    - **Compositionality, congruence property:**  
Behaviour of system only depends on the behaviours of its components
- ⇒ Hierarchical construction of (LTS representations of) behaviour
- Many notions of abstract behaviour, for good reasons:  
failure semantics, observational equivalence = weak bisimilarity, ...

# Lack of use of process algebras

- Which process algebra (language) to choose?
- Which semantics to choose (if I happen to realise that I can choose it)?
- Why not good old loops and assignments?

$$\begin{aligned}
 CD = \text{card\_in} \rightarrow \text{request?amount} \rightarrow ( \\
 & \qquad \qquad \qquad \text{card\_out} \rightarrow \text{money!amount} \rightarrow CD \\
 & \qquad \qquad \qquad \sqcap \text{no\_money} \rightarrow \text{card\_out} \rightarrow CD )
 \end{aligned}$$

- How do I understand this?

$$\perp_{\mathcal{N}} = (\Sigma^{*\checkmark} \times \mathbb{P}(\Sigma^{\checkmark}), \Sigma^{*\checkmark})$$

- What observational *congruence*?
- How to present state information?

“in\_critical”  $\sim$  “enter\_critical”, “exit\_critical”

# Lack of rigour of state machines

- Practical engineers use state machines a lot both in SW and HW
  - Semantics of state machines is usually unclear
    - In particular, of communication / interaction
  - Example: SDL input mechanism
    - One input queue / process
    - Runs in trouble when has to listen to  $\geq 2$  directions
- ⇒ Clumsy “save”-mechanism: read first not saved item from the queue
- ⇒ Very few SDL users respect the formal semantics

# Idea: apply process-algebraic theory to interacting state machines

- ⇒ My biennial course on Lotos and CFFD theory gradually changed to a state machine course
- Facilitates presenting ideas from many theories in a uniform framework
    - State space / labelled transition system / Kripke structure
    - Unfolding of data
    - Deadlock, livelock, fairness, . . .
    - Various communication and interaction primitives
    - Strong bisimilarity and isomorphism of behaviours
  - The above can be taught before introducing full abstraction
    - Easier and widely applicable material
    - Strong foundation for studying full abstraction

# The course

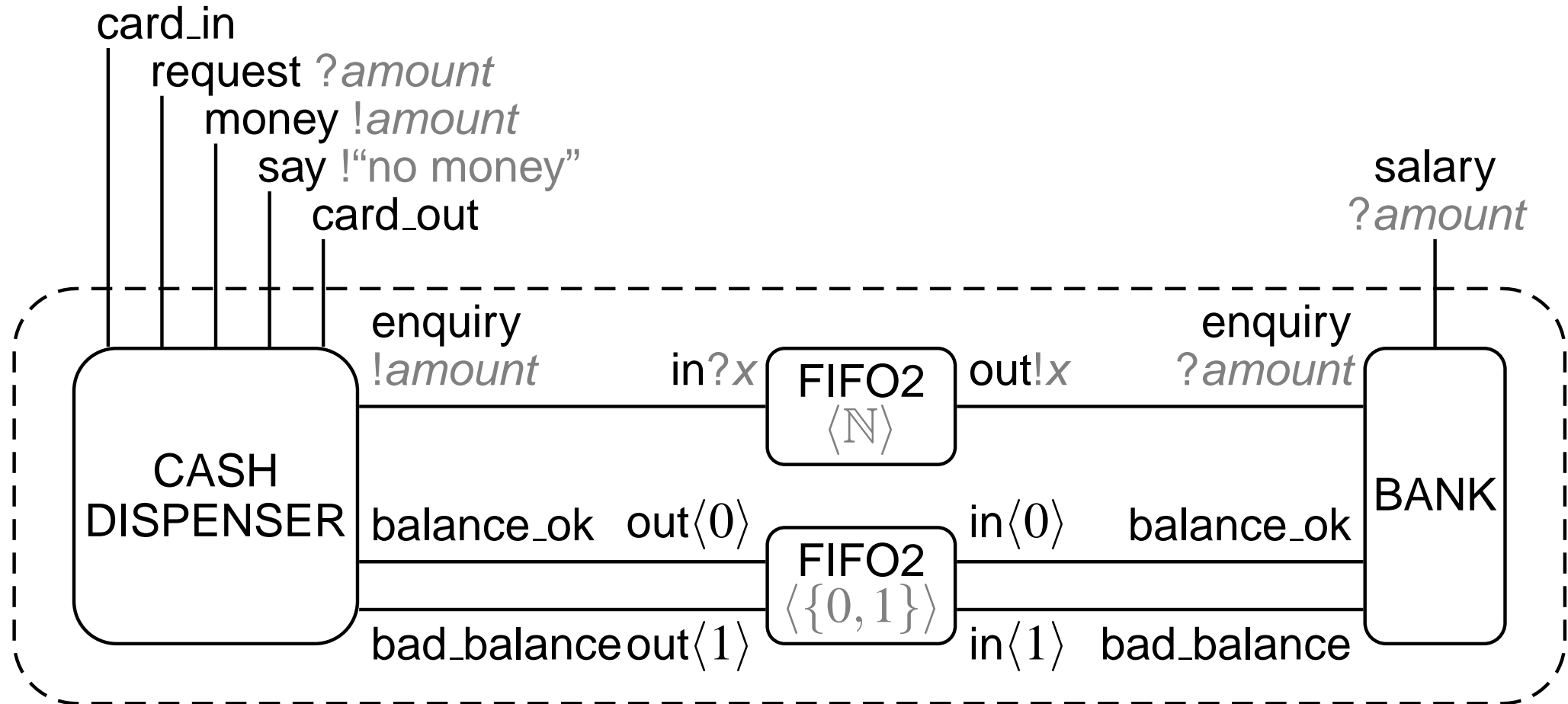
place	TUT	TUT	Hels	TUT	Hels	TUT	TUT
year	1997	1999	2000	2001	2002	2004	2006
passed students	12	13	6	6	9	6	5

- Heterogeneous students
  - General trend: skills decrease
- Implementation changed each time
- State machines since 2004

⇒ Data insufficient for strong conclusions

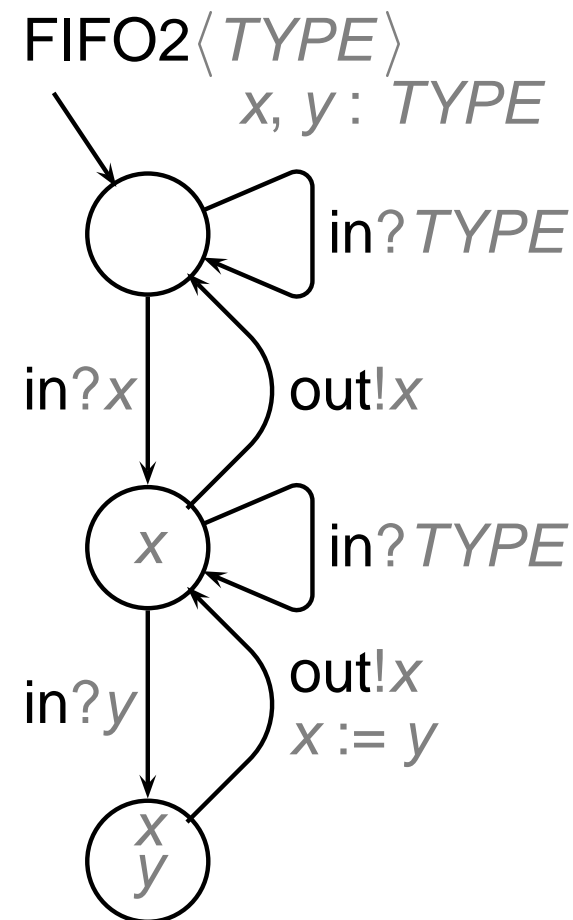
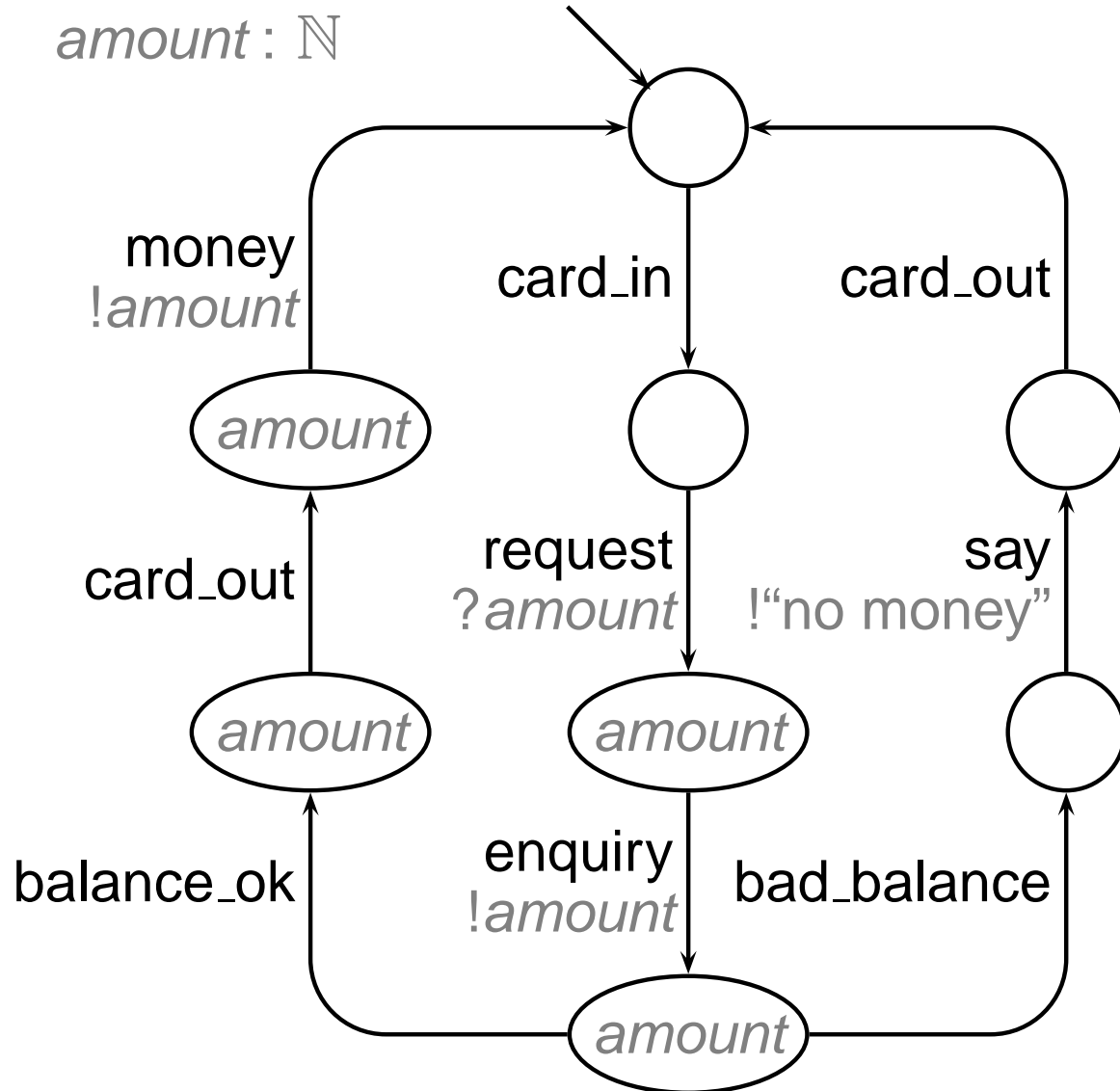
- However, some observations seem clear
  - Mentioned in this talk

## 2 A system consisting of state machines

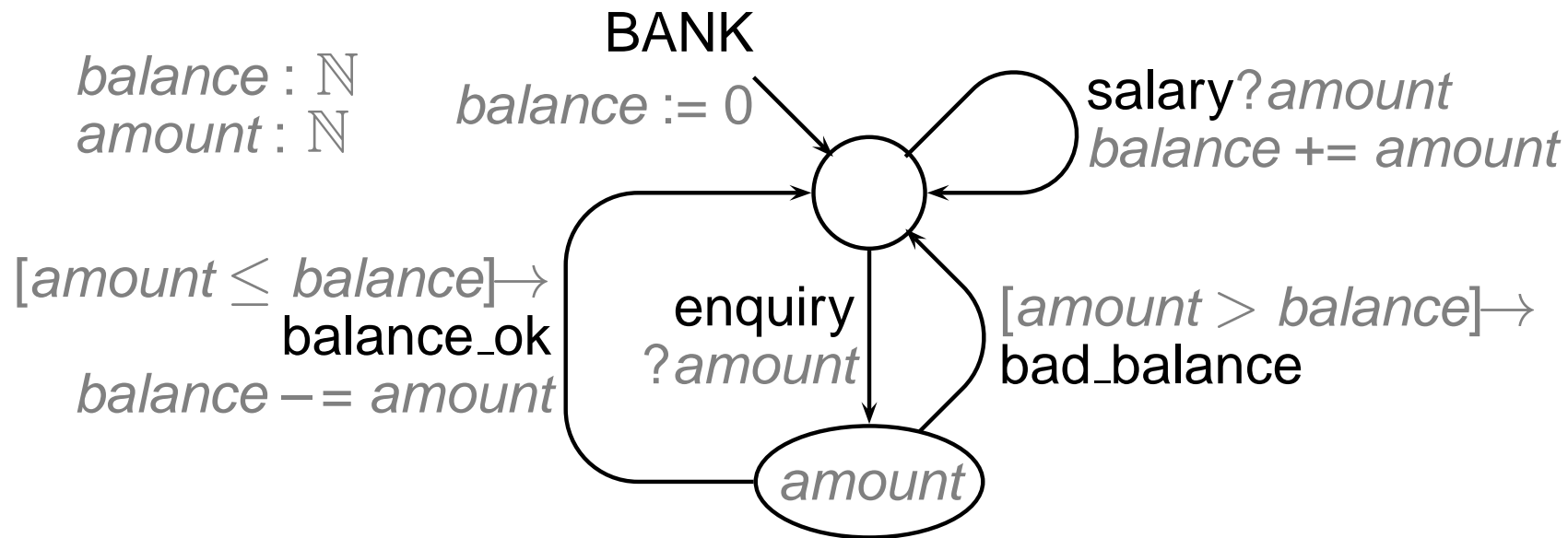


# Individual state machines ...

## CASH DISPENSER



# ... Individual state machines



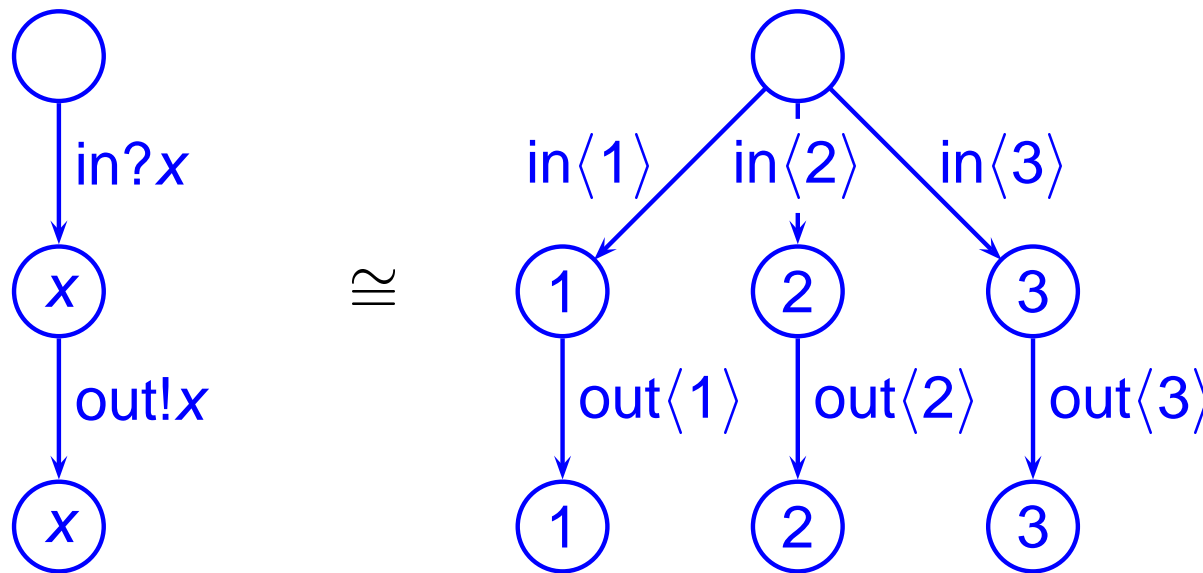
### 3 Communication and interaction

- Synchronous interaction =  
*participants take a transition simultaneously*
- A state machine may be made to wait for alternative synchronisations without restrictions
- Grand ideas of concurrency:  
***All interaction can be expressed in terms of synchronous interaction.***
  - E.g., fifos in the previous system
- Drawing conventions allow flexible synchronisation
- Theorem: can be returned to  
hiding + relational renaming + synchronisation via the same name

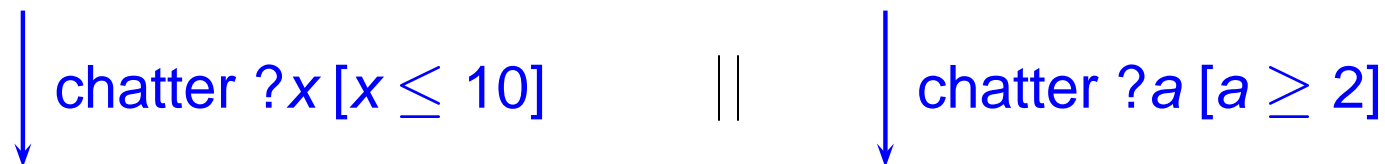
# Input and output ...

- input = environment determines (and state machine remembers) the value

output = state machine determines the value



- Restricting range with postconditions



## ... Input and output

- Communication of several values in one action

$$\begin{array}{ccc} \downarrow & & \downarrow \\ \text{chatter } ?x !y !(y+1) & \parallel & \text{chatter } !3 ?a !2 \cdot b [a \geq 2] \end{array}$$

⇒ Grand ideas of concurrency:

***Input and output are just roles in certain forms of synchronous interaction. They are not always meaningful concepts.***

# Why synchronous interaction?

- Synchronous interaction is **a fundamental theoretical concept**
  - Difficult or impossible to implement in practice
  - All real-life mechanisms can be expressed in terms of it

- Like atoms in engineering

- Course design principle

***The main goal is in understanding concurrency,  
not in learning to design systems.***

- Good goal with the current level of maturity of
  - concurrency theory with various independent formalisms
  - concurrent programming with various alternative primitives

- Students grasp general synchronous interaction and its universality well

## 4 Formal definition of state machines

- Without variables

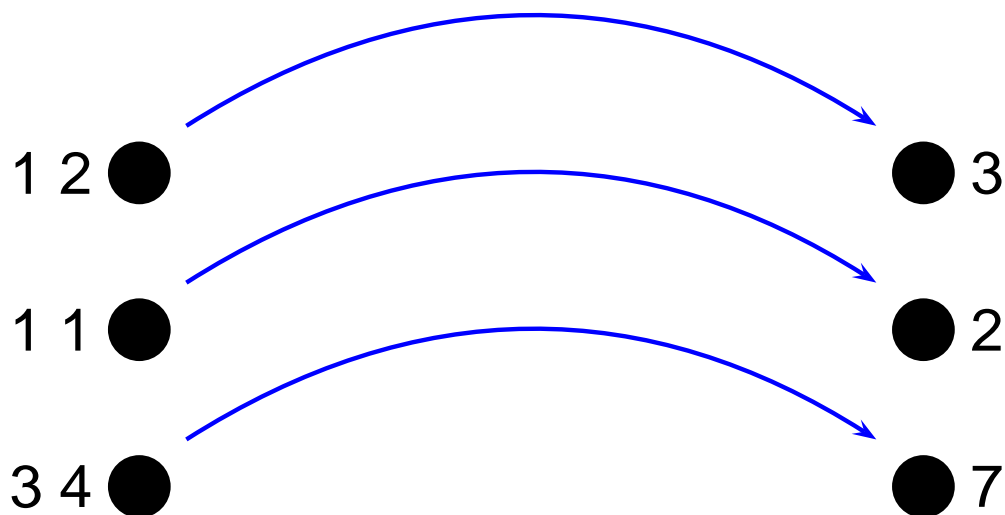
$$(S, \Sigma, \Delta, \hat{S}, \Pi, val)$$

- $\tau \notin \Sigma$ ,  $\tau$  is the ***invisible action***
  - $\Delta \subseteq S \times (\Sigma \cup \{\tau\}) \times S$
  - $\emptyset \neq \hat{S} \subseteq S$
  - $val \subseteq \Pi \times S$
- Multiple initial states are allowed
    - $\Rightarrow$  No choice operator
  - State propositions are second class citizens
    - Cannot be seen by other state machines
    - $\Rightarrow$  Separates interaction aspects from verification aspects
      - Most (all?) communication needs are handled with another mechanism

# Transition relations ...

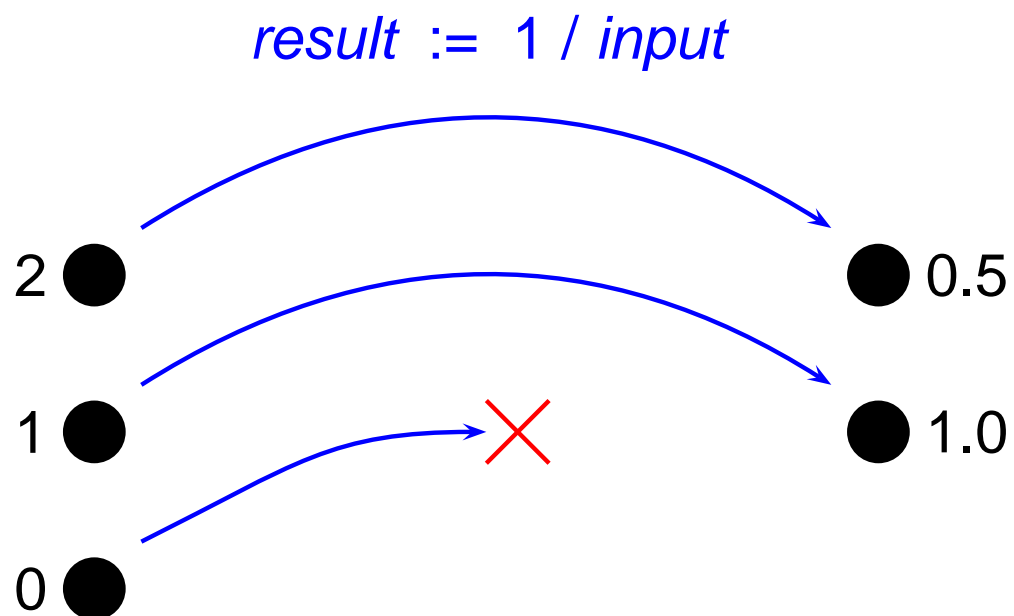
- A sequential program is a function  $\text{input} \mapsto \text{output} \dots$

*result := input1 + input2*



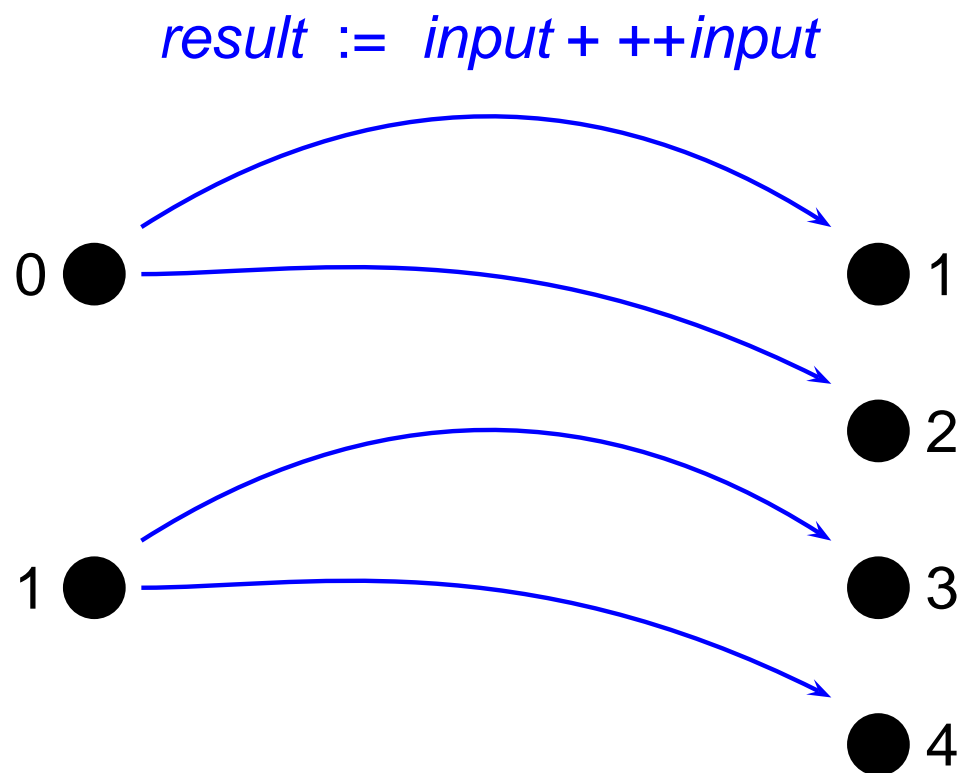
# ... Transition relations ...

- ... oops, it is a partial function ...



## ... Transition relations ...

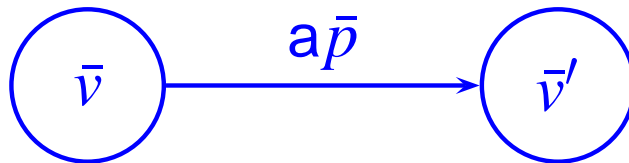
- ... oops, it is a relation  $R(input, output)$



## ... Transition relations

⇒ Important observation:

***Computation with variables in a transition  
can be abstracted as a relation  $R(\bar{v}, \bar{p}, \bar{v}')$***



- Transition is now  $(s, a, R, s')$
- We can forget about programming language syntax and semantics
- They can forget about state machine theory

***An excellent separation of concerns between the theories  
of programming languages and state machines***

# Formal definition with variables

$$(S, VC, \Sigma, \Delta, \hat{S}, Init, \Pi, val)$$

- Some details
  - $VC$  = all potential combinations of variable values
  - $\Sigma$  = gate names
  - $Init \subseteq \hat{S} \times VC$  specifies initial values of variables
- Semantics is obtained via the unfolding construction
  - $U$  = universe of communicated data values
  - $\hat{S}_U = \{ \hat{s} \langle \bar{v} \rangle \mid (\hat{s}, \bar{v}) \in Init \}$
  - $\Sigma_U = \Sigma \times U^*$
  - $\Delta_U$  is those reachable  $s \langle \bar{v} \rangle - a \langle \bar{p} \rangle \rightarrow s' \langle \bar{v}' \rangle$  for which
 
$$\exists (s, a, R, s') \in \Delta : (\bar{v}, \bar{p}, \bar{v}') \in R$$

# Behaviour = state machine

- Grand ideas of concurrency:

*The behaviour of any state machine is  
a state machine without variables.*

⇒ Behaviour becomes something that may be grasped

- Systems may be built hierarchically — also their behaviours may

## 5 More non-abstract compositionality results ...

- Important result:

***The behaviour of one or more interacting state machines is a state machine without variables.***

- The behaviour is just the state space

- Important result:

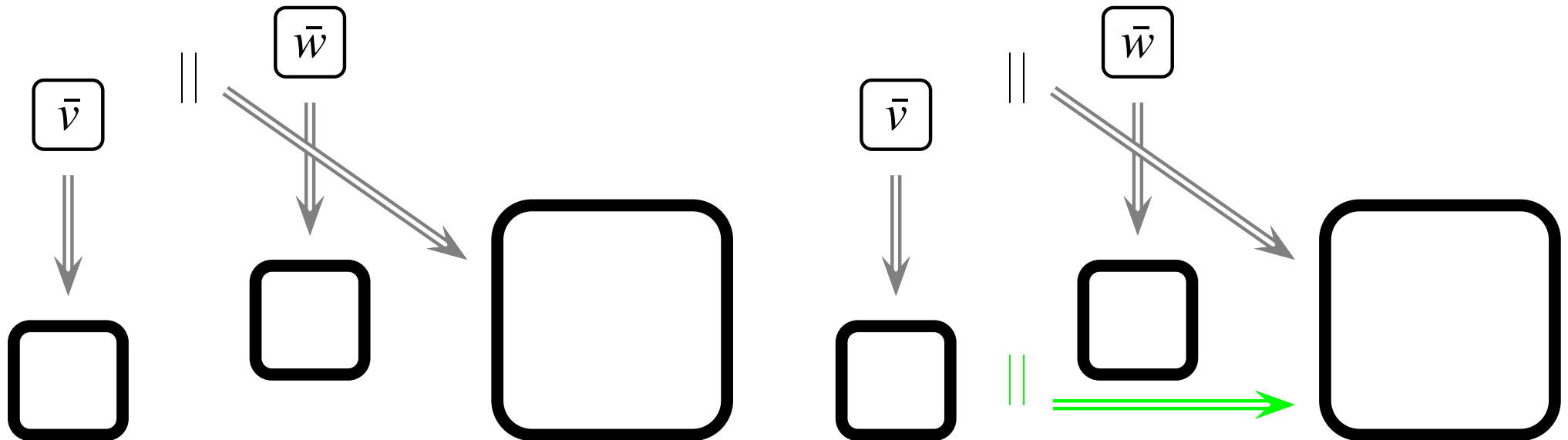
***Unfolding of a variable is equivalent to parallel composition with a state machine that represents the variable.***

⇒ We have got rid of the notion of variable!

# ... More non-abstract compositionality results ...

⇒ Result:

*The behaviour of the system as a whole is the parallel composition of the behaviours of the components.*

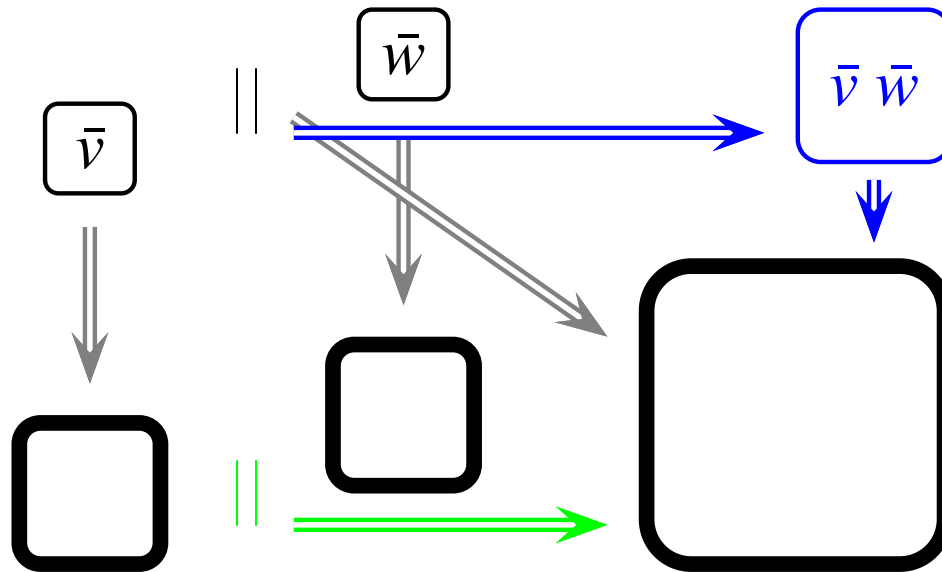


– Compositional state space construction, but still without abstraction

## ... More non-abstract compositionality results

- Result:

*Parallel composition can also be computed with variables present.*



- Little known
- Allows combination of parallel processes into one process at the level of program code
- Students did this easily at the level of pseudocode used above

## 6 Last non-abstract topics

- Strong bisimilarity as the notion of “same behaviour”
    - Students tend to initially prefer isomorphism
    - An example with remnant variable values convinces them
  - Representing synchronisation rules in terms of conventional operators
  - Associativity, commutativity and distributivity theorems about conventional operators
- ⇒ Strong foundation has been laid for teaching abstract semantics
- I do not have much new regarding teaching the abstract semantics
    - ⇒ I skip it in this talk
  - Alternatively, one can continue with Kripke structures and temporal logics

## 7 Conclusions

- The approach focuses on a (usually hidden) common basis of many theories, not on individual theories or design methods
- One's favourite syntax can be used for handling data in examples, while the theory is totally free from syntax issues
- One's favourite communication / interaction primitives can be used, by building them from synchronous interaction
- Economy of concepts:
  - Local variable = parallel state machine without variables
  - Behaviour of anything = state machine without variables
- Compositionality proves to apply very widely and in many orderings
- Treatment of state propositions was not fully satisfactory

**Thank you for attention!**