Theory of Computing

Проф. Мануель Маццара

Guest lecture: CSP

(Bertrand Meyer)
with material from the ETH course
“Concepts of Concurrent Computation”

CSP purpose

Concurrency formalism

- Expresses many concurrent situations elegantly
- Influenced design of several concurrent programming languages, in particular Occam (Transputer)

Calculus

- Formally specified: laws
- Makes it possible to prove properties of systems

Concurrency calculi

Also known as process calculi

Aim: provide abstract models of concurrent computation

Independent of particular concurrency frameworks, languages, architectures, implementations

Examples:

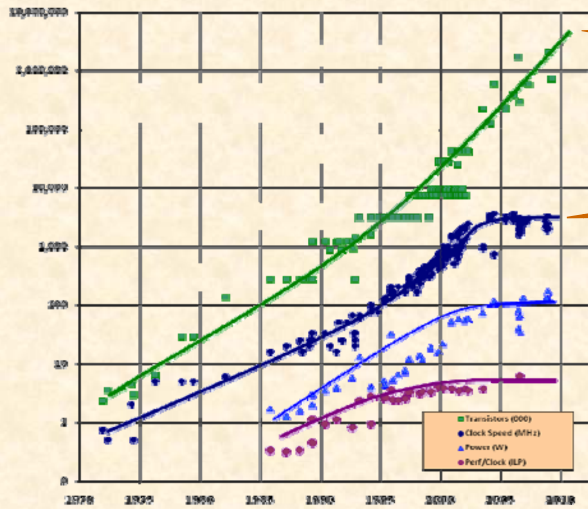
- CSP (this lecture), Tony Hoare
- CCS (Calculus of Communicating Systems), Robin Milner, 1973-1980
- π-calculus, Milner: distribution
- Ambient Calculus, Cardelli/Gordon: mobility

3

Concurrent programming is hard!

4

Эффективность

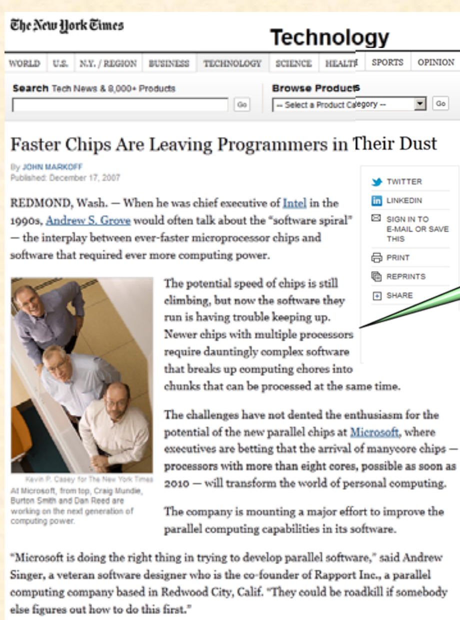


Число транзисторов

Тактовая частота

Источник: Intel

New York Times, декабрь 2007 г.



The New York Times

Technology

WORLD | U.S. | N.Y. / REGION | BUSINESS | TECHNOLOGY | SCIENCE | HEALTH | SPORTS | OPINION

Search Tech News & 8,000+ Products

Browse Products

Faster Chips Are Leaving Programmers in Their Dust

By JOHN MARKOFF
Published: December 17, 2007

REDMOND, Wash. — When he was chief executive of Intel in the 1990s, Andrew S. Grove would often talk about the "software spiral" — the interplay between ever-faster microprocessor chips and software that required ever more computing power.

The potential speed of chips is still climbing, but now the software they run is having trouble keeping up. Newer chips with multiple processors require dauntingly complex software that breaks up computing chores into chunks that can be processed at the same time.

The challenges have not dented the enthusiasm for the potential of the new parallel chips at Microsoft, where executives are betting that the arrival of manycore chips — processors with more than eight cores, possible as soon as 2010 — will transform the world of personal computing.

The company is mounting a major effort to improve the parallel computing capabilities in its software.

"Microsoft is doing the right thing in trying to develop parallel software," said Andrew Singer, a veteran software designer who is the co-founder of Rapport Inc., a parallel computing company based in Redwood City, Calif. "They could be roadkill if somebody else figures out how to do this first."

Kevin P. Casey for The New York Times
At Microsoft, from top, Craig Mundie, Burton Smith and Dan Reed are working on the next generation of computing power.

TWITTER
LINKEDIN
SIGN IN TO E-MAIL OR SAVE THIS
PRINT
REPRINTS
SHARE

«Новые процессоры с несколькими ядрами требуют чрезвычайно сложного программного обеспечения»

Что говорят о параллельном программировании

Интел, 2006:

- *Многоядерные вычисления быстрым и захватывающим путем переводят индустрию на абсолютно новую территорию*

Рик Рашид, глава Microsoft Research, 2007:

- *Многоядерные процессоры представляют собой одну из крупнейших смен технологии, с глубокими следствиями в методах разработки программ*

Билл Гейтс:

- *Мы никогда не сталкивались с подобными задачами.* **Здесь нужен прорыв**.

Дэвид Паттерсон, Калифорнийский университет в Беркли, 2007:

- *Вся индустрия, в принципе, сделала отчаянный выбор. Она делает ставку на параллельные вычисления. Ставка сделана, но большая проблема - добиться выигрыша*

7

US National Academy of Science, 2011

Heroic programmers *can exploit vast amounts of parallelism…*

However, ***none of those developments comes close to the ubiquitous support for programming parallel hardware*** *that is required to ensure that IT's effect on society over the next two decades will be as stunning as it has been over the last half-century*

8

Programming for heroes: dining philosophers

Listing 4.33: Variables for Tanenbaum's solution

```
state = ['thinking'] * 5
sem = [Semaphore(0) for i in range(5)]
mutex = Semaphore(1)
```

The initial value of state is a list of 5 copies of 'thinking'. sem is a list of 5 semaphores with the initial value 0. Here is the code:

Listing 4.34: Tanenbaum's solution

```
def get_fork(i):
    mutex.wait()
    state[i] = 'hungry'
    test(i)
    mutex.signal()
    sem[i].wait()

def put_fork(i):
    mutex.wait()
    state[i] = 'thinking'
    test(right(i))
    test(left(i))
    mutex.signal()

def test(i):
    if state[i] == 'hungry' and
    state (left (i)) != 'eating' and
    state (right (i)) != 'eating':
        state[i] = 'eating'
        sem[i].signal()
```

Allen Downey: The Little Green Book of Semaphores, greenteapress.com/semaphores/

Bank transfer

transfer (*source*, *target*: *ACCOUNT*;
    amount: *INTEGER*)
        -- If enough funds, transfer amount from source to target.
    do
        if *source*.*balance* >= *amount* **then**
            source.*withdraw* (*amount*)
            target.*deposit* (*amount*)
        end
    end

transfer (*Jane*, *Jill*, 100) 1

transfer (*Jane*, *Joan*, 100) 2

Jane	Jill	Joan
100	0	0
0	100	0
-100	0	100

Bank transfer (better version)

```
transfer (source, target:                    ACCOUNT;
            amount: INTEGER)
        -- Transfer amount from source to target.
    require
        source.balance >= amount
    do
        source.withdraw (amount)
        target.deposit  (amount)
    ensure
        source.balance = old source.balance – amount
        target.balance = old target.balance + amount
    end
```


The inability to reason from APIs

if acc1.balance >= 10( **then** transfer (acc1, **acc2**, 100) **end**

if acc1.balance >= 100 **then** transfer (acc1, **acc3**, 100) **end**

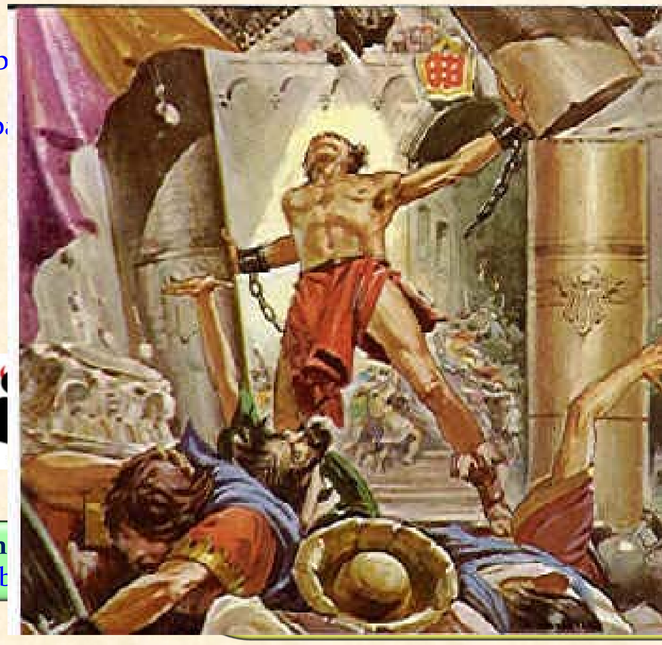
```
transfer (source, target:                    ACCOUNT;
            amount: INTEGER)
        -- Transfer amount from source to target.
    require
        source.balance >= amount
    do
        ...
    ensure
        source.balance = old source.balance – amount
        target.balance = old target.balance + amount
    end
```

```
invariant
    balance >= 0
```

The inability to reason from APIs

if acc1.b)) end

if acc1.ba end



ount
unt

invarian
b

CSP purpose

Concurrency formalism

- Expresses many concurrent situations elegantly
- Influenced design of several concurrent programming languages, in particular Occam (Transputer)

Calculus

- Formally specified: laws
- Makes it possible to prove properties of systems

Traces

A trace is a sequence of events, for example

<coin, coffee, coin, coffee>

Many traces of interest are infinite, for example

<coin, coffee, coin, coffee, …>

(Can be defined formally, e.g by regular expressions, but such traces definition are not part of CSP; they are descriptions of CSP process properties.)

Events come from an *alphabet*. The alphabet of all possible events is written Σ in the following.

15

Processes and their traces

A CSP process is characterized (although not necessarily defined fully) by the set of its traces. For example a process may have the trace set

{<>,
<coin, coffee>,
<coin, tea>}

The special process STOP has a trace set consisting of a single, empty trace:

{<>}

16

Basic CSP syntax

P ::=

STOP	|	-- Does not engage in any events
$a \rightarrow Q$	|	-- Engages in a, then acts like Q
$Q \sqcap R$	|	-- Internal choice
$Q \square R$	|	-- External choice
$Q \parallel_E R$	|	-- Concurrency (E: subset of alphabet)
$Q \parallel R$	|	-- Lock-step concurrency (same as $Q \parallel_\Sigma R$)
$Q \setminus E$	|	-- Hiding
$\mu Q \bullet f(Q)$		-- Recursion

17

Generalization of → notation

Basic:

$a \rightarrow P$

Generalization:

$x: E \rightarrow P(x)$

Accepts any event from E, then executes P (x) where x is that event

Also written

$?\, x: E \rightarrow P(x)$

Note that if E is empty then $x: E \rightarrow P(x)$ is STOP for any P

18

Some laws of concurrency

1. $P \mid\mid Q = Q \mid\mid P$
2. $(P \mid\mid (Q \mid\mid R)) = ((P \mid\mid Q) \mid\mid R)$

3. $P \mid\mid STOP = STOP$

4. $(c \rightarrow P) \mid\mid (c \rightarrow Q) = (c \rightarrow (P \mid\mid Q))$
5. $(c \rightarrow P) \mid\mid (d \rightarrow Q) = STOP$ -- If $c \neq d$

6. $(x: A \rightarrow P(x)) \mid\mid (y: B \rightarrow Q(y)) =$
   $(z: (A \cap B) \rightarrow (P(z) \mid\mid Q(z))$

Basic notions

Processes engage in events

Example of basic notation:

CVM = (coin → coffee → coin → coffee → STOP)

Right associativity: the above is an abbreviation for

CVM = (coin → (coffee → (coin → (coffee → STOP))))

Trace set of CVM: {<coin, coffee, coin, coffee>}

The events of a process are taken from its alphabet:

α(CVM) = {coin, coffee}

STOP can engage in no events

Traces

traces (e → Q) = {<e> + s | s ∈ traces (Q)}

Exercises: determine traces

P ::=

STOP	|	-- Does not engage in any events
a → Q	|	-- Engages in a, then acts like Q
Q ⊓ R	|	-- Internal choice
Q □ R	|	-- External choice
$Q \mathbin{\|\|_E} R$	|	-- Concurrency (E: subset of alphabet)
Q || R	|	-- Lock-step concurrency (same as $Q \mathbin{\|\|_\Sigma} R$)
Q \ E	|	-- Hiding
μQ • f (Q)		-- Recursion

Recursion

CLOCK = (tick → CLOCK)

This is an abbreviation for

CLOCK = μP • (tick → P)

A recursive definition is a fixpoint equation. The μ notation denotes the fixpoint

Accepting one of a set of events; channels

Basic notation:

? x: A → P (x)

Accepts any event from A, then executes P (x) where x is that event

Channel names

Example:

? y: c.A → d.y'

(where c.A denotes {c.x | x ∈ A} and y' denotes y deprived of its initial channel name, e.g. (c.a)' = a)

More convenient notation for such cases involving channels:

$\boxed{\text{c? x: A → d!x}}$

A simple buffer

COPY = c?x: A → d!x → COPY

External choice

COPYBIT = (in.0 → out.0 → COPYBIT

□

in.1 → out.1 → COPYBIT)

External choice

COPY1 = in? x: A → out1!x → COPY1

COPY2 = in? x: B → out2!x → COPY2

COPY3 = COPY1 □ COPY2

27

External choice

Consider

CHM1 = (in1r → out50k → out20k → out20k → out10k)
CHM2 = (in1r → out50k → out50k)

CHM = CHM1 □ CHM2

28

Lock-step concurrency

Consider

$P \quad = \quad ?x: A \rightarrow P'$

$Q \quad = \quad ?x: B \rightarrow Q'$

Then

$P \,||\, Q = ?\,x \rightarrow$

- $(P' \,||\, Q')$ if $x \in A \cap B$
- STOP otherwise

(to be generalized soon)

29

More examples

VMC =

(in2f →
((large → VMC) □
(small → out1f → VMC))
□
(in1f →
((small→ VMC) □
(in1f → large → VMC))

FOOLCUST = (in2f → large → FOOLCUST □
in1f → large → FOOLCUST)

FV = FOOLCUST || VMC =
μP • (in2f →large → P □ in1f → STOP)

30

Hiding

Consider

$P = a \rightarrow b \rightarrow Q$

Assuming Q does not involve b, then

$P \setminus \{b\} = a \rightarrow Q$

More generally:

$(a \rightarrow P) \setminus E =$

- $P \setminus E$ if $a \in E$
- $a \rightarrow (P \setminus E)$ if $a \notin E$

31

Hiding introduces internal non-determinism

Consider

$R = (a \rightarrow P) \;\square\; (b \rightarrow Q)$

Then

$R \setminus \{a, b\} = P \sqcap Q$

32

Internal non-deterministic choice

CH1F = (in1f →
((out20rp → out20rp →
out20rp →out20rp →out20rp → CH1F)
⊓
(out50rp → out50rp → CH1F)))

33

Non-deterministic internal choice: another application

TRANSMIT (x) = in?x → LOSSY (x)

LOSSY (x) =
out!x → TRANSMIT (x)
⊓ out!x → LOSSY (x)
⊓ TRANSMIT (x)

34

The general concurrency operator

Consider

$P \;=\; ?x: A \rightarrow P'$

$Q \;=\; ?x: B \rightarrow Q'$

Then

$P \underset{E}{||} Q = ?\, x \rightarrow$

- $P' \underset{E}{||} Q'$ if $x \in E \cap A \cap B$
- $P' \underset{E}{||} Q$ if $x \in A\text{-}B\text{-}E$
- $P \underset{E}{||} Q'$ if $x \in B\text{-}A\text{-}E$
- $(P' \underset{E}{||} Q) \sqcap (P \underset{E}{||} Q')$ if $x \in (A \cap B) - E$

35

Special cases of concurrency

Lock-step concurrency:

$P \,||\, Q \quad = \; P \underset{\Sigma}{||} Q$

Interleaving:

$P \,|||\, Q = \; P \underset{\varnothing}{||} Q$

36

Lock-step concurrency (reminder)

Consider

$P = ?x: A \rightarrow P'$

$Q = ?x: B \rightarrow Q'$

Then

$P \mid\mid Q = ?\, x \rightarrow$

- $(P' \mid\mid Q')$ if $x \in E \cap A \cap B$
- $STOP$ otherwise

37

Laws of non-deterministic internal choice

$P \sqcap P = P$

$P \sqcap Q = Q \sqcap P$

$P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R$

$x \rightarrow (P \sqcap Q) = (x \rightarrow P) \sqcap (x \rightarrow Q)$

$P \mid\mid (Q \sqcap R) = (P \mid\mid Q) \sqcap (P \mid\mid R)$

$(P \sqcap Q) \mid\mid R = (P \mid\mid R) \sqcap (Q \mid\mid R)$

The recursion operator is not distributive; consider:

$P = \mu X \bullet ((a \rightarrow X) \sqcap (b \rightarrow X))$

$Q = (\mu X \bullet (a \rightarrow X)) \sqcap (\mu X \bullet (b \rightarrow X))$

38

Note on external choice

From previous slide:

$x \rightarrow (P \sqcap Q) = (x \rightarrow P) \sqcap (x \rightarrow Q)$

The question was asked in class of whether a similar property also applies to external choice $\square$

The conjectured property is

$x \rightarrow (P \square Q) = (x \rightarrow P) \square (x \rightarrow Q)$

It does not hold, since

$(x \rightarrow P) \square (x \rightarrow Q) = x \rightarrow (P \sqcap Q)$

(As a consequence of rule on next page)

39

General property of external choice

$(?x: A \rightarrow P) \square (?x: B \rightarrow Q) =$

$?x: A \cup B \rightarrow$

- P if $x \in A\text{-}B$
- Q if $x \in B\text{-}A$
- $P \sqcap Q$ if $x \in A \cap B$

40

Traces

traces (e → P) = {<e> + s | s ∈ traces (P)}

Exercise: determine traces

P ::=

STOP	|	-- Does not engage in any events
a → Q	|	-- Engages in a, then acts like Q
Q ⊓ R	|	-- Internal choice
Q □ R	|	-- External choice
$Q \parallel_E R$	|	-- Concurrency (E: subset of alphabet)
Q || R	|	-- Lock-step concurrency (same as $Q \parallel_\Sigma R$)
Q \ E	|	-- Hiding
μQ • f (Q)		-- Recursion

Refinement

Process Q *refines* (specifically, *trace-refines*) process P if

$$\text{traces}(Q) \subseteq \text{traces}(P)$$

For example:

P *refines* $P \sqcap Q$

The trace model is not enough

The traces of and are the same:

$$\text{traces}(P \,\Box\, Q) = \text{traces}(P) \cup \text{traces}(Q)$$
$$\text{traces}(P \sqcap Q) = \text{traces}(P) \cup \text{traces}(Q)$$

But the processes can behave differently if for example:

$$P = a \rightarrow b \rightarrow STOP$$
$$Q = b \rightarrow a \rightarrow STOP$$

Traces define what a process may do, not what it may refuse to do

Refusals

For a process P and a trace t of P:

- An event set es ∈ ℙ (∑) is a *refusal set* if P can forever refuse all events in es
- Refusals (P) is the set of P's refusal sets
- Convention: keep only maximal refusal sets
  (if X is a refusal set and Y ⊆ X, then Y is a refusal set)

This also leads to a notion of "failure":

- Failures (P, t) is Refusals (P / t)

where P/t is P *after* t :

traces (P / t) = {u | t + u ∈ traces (P))

45

Comparing failures

Compare

- P = a → STOP □ b → STOP
- Q = a → STOP ⊓ b → STOP

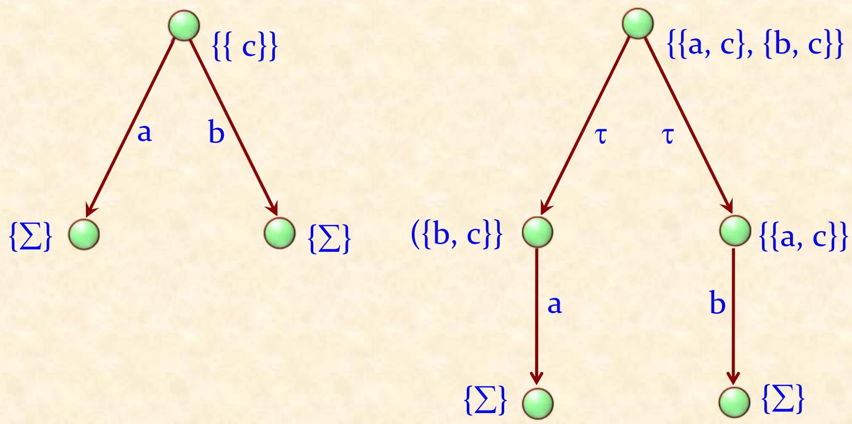
Same traces, but:

- Refusals (P) = ∅
- Refusals (Q) = {{a}, {b}}

46

Refusal sets (from labeled transition diagram)

$\Sigma = \{ a, b, c\}$



a → STOP ☐ b → STOP

a → STOP ⊓ b → STOP

A more complete notion of refinement

Process Q *failures-refines* process P if both

traces (Q) $\subseteq$ traces (P)
failures (Q) $\subseteq$ failures (P)

Makes it possible to distinguish between ☐ and ⊓

Divergence

A process diverges if it is not refusing all events but not communicating with the environment

This happens if a process can engage in an infinite sequence of τ transitions

An example of diverging process:

$$(\mu\, p.a \rightarrow p) \setminus a$$

49

The divergence model (Brookes, Roscoe)

CSP semantics is often expressed through a failures set

A failure is of the form

$$[s, X]$$

where s is a trace (sequence of events) and X a finite set of events

A failure set must satisfy the following properties:

- $[\langle\rangle, \emptyset] \in F$
- $[s + t, \emptyset] \in F \;\Rightarrow\; [s, \emptyset] \in F$
- $[s, X] \in F \wedge Y \subseteq X \;\Rightarrow\; [s, Y] \in F$
- $[s, X] \in F \wedge [s + \langle c\rangle, \emptyset] \notin F \;\Rightarrow\; [s, X \cup \{c\}] \in F$

50

Basic CSP syntax

P ::=

STOP	|	-- Does not engage in any events
$a \rightarrow Q$	|	-- Engages in a, then acts like Q
$Q \sqcap R$	|	-- Internal choice
$Q \,\square\, R$	|	-- External choice
$Q \parallel_E R$	|	-- Concurrency (E: subset of alphabet)
$Q \parallel R$	|	-- Lock-step concurrency (same as $Q \parallel_\Sigma R$)
$Q \setminus E$	|	-- Hiding
$\mu Q \bullet f(Q)$		-- Recursion

CSP laws in the divergence model (1/2)

$$
\begin{aligned}
P \,\square\, P &\equiv_M P \\
P \,\square\, Q &\equiv_M Q \,\square\, P \\
P \,\square\, (Q \,\square\, R) &\equiv_M (P \,\square\, Q) \,\square\, R \\
P \,\square\, (Q \sqcap R) &\equiv_M (P \,\square\, Q) \sqcap (P \,\square\, R) \\
P \sqcap (Q \,\square\, R) &\equiv_M (P \sqcap Q) \,\square\, (P \sqcap R) \\
P \,\square\, \mathrm{STOP} &\equiv_M P \\
(a \rightarrow (P \sqcap Q)) &\equiv_M (a \rightarrow P) \sqcap (a \rightarrow Q) \\
(a \rightarrow P) \,\square\, (a \rightarrow Q) &\equiv_M (a \rightarrow P) \sqcap (a \rightarrow Q) \\
P \sqcap P &\equiv_M P \\
P \sqcap Q &\equiv_M Q \sqcap P \\
P \sqcap (Q \sqcap R) &\equiv_M (P \sqcap Q) \sqcap R \\
P \parallel Q &\equiv_M Q \parallel P \\
P \parallel (Q \parallel R) &\equiv_M (P \parallel Q) \parallel R \\
P \parallel (Q \sqcap R) &\equiv_M (P \parallel Q) \sqcap (P \parallel R) \\
(a \rightarrow P) \parallel (b \rightarrow Q) &\equiv_M \mathrm{STOP} \qquad \text{if } a \neq b \\
&\equiv_M (a \rightarrow (P \parallel Q)) \quad \text{if } a = b \\
P \parallel \mathrm{STOP} &\equiv_M \mathrm{STOP}
\end{aligned}
$$

(From: Brooks & Roscoe 85)

CSP laws (2/2)

$$
\begin{aligned}
P|||Q &\equiv_M Q|||P \\
(P|||Q)|||R &\equiv_M P|||(Q|||R) \\
P|||(Q\sqcap R) &\equiv_M (P|||Q)\sqcap(P|||R) \\
(a\to P)|||(b\to Q) &\equiv_M (a\to(P|||(b\to Q)))\,\Box\,(b\to((a\to P)|||Q)) \\
P;(Q;R) &\equiv_M (P;Q);R \\
\mathrm{STOP}|||Q &\equiv_M Q \\
\mathrm{SKIP};Q &\equiv_M Q \\
\mathrm{STOP};Q &\equiv_M \mathrm{STOP} \\
P;(Q\sqcap R) &\equiv_M (P;Q)\sqcap(P;R) \\
(P\sqcap Q);R &\equiv_M (P;R)\sqcap(Q;R) \\
(a\to P);Q &\equiv_M (a\to P;Q) \qquad \text{if } a\neq\checkmark \\
(P\backslash a)\backslash b &\equiv_M (P\backslash b)\backslash a \\
(P\backslash a)\backslash a &\equiv_M P\backslash a \\
(a\to P)\backslash b &\equiv_M (a\to P\backslash b) \qquad \text{if } a\neq b \\
&\equiv_M P\backslash b \qquad\qquad \text{if } a=b \\
(P\sqcap Q)\backslash a &\equiv_M (P\backslash a)\sqcap(Q\backslash a)
\end{aligned}
$$

53

Some extensions

Non-timed:

- The ✓ event (not in Σ): successful termination
- Skip : successfully terminates
- Sequential composition: P ; Q
- $\perp$: diverging process

Timed:

- $P \overset{t}{\not{}} Q$: interrupt
- $P \overset{t}{\triangleright} Q$: timeout
- $a \overset{!}{\rightarrow} P$: communicate immediately
- WAIT t: same as $\mathrm{STOP} \overset{t}{\triangleright} \mathrm{SKIP}$

54

Example (Ouaknine)

$V1 = coin.in \rightarrow$

$((coke \rightarrow V1) \,\square\, (fanta \rightarrow V1)) \overset{60}{\triangleright} (coin.out \overset{!}{\rightarrow} V1)$

Some laws no longer hold

$P \,||\, STOP = STOP$ if $P \neq \bot$

$\bot \,||\, STOP = \bot$

$(a \rightarrow P) \setminus b \;=\; a \rightarrow (P \setminus b)$ if $a \neq b$

$(a \rightarrow P) \setminus a \;=\; P \setminus a$

CSP: Summary

A calculus based on mathematical laws

Provides a general model of computation based on communication

Serves both as specification of concurrent systems and as a guide to implementation

One of the most influential models for concurrency work