Computations and Interaction

Jos C.M. Baeten

Systems Engineering and Theory of Computing

Joint work with Bas Luttik and Paul van Tilburg

February 10, 2011
What is a computation?

Hand in stack of punched cards at counter
Wait 2 hours
Find dump in pigeon hole
What is a computation?

Hand in stack of punched cards at counter
Wait 2 hours
Find dump in pigeon hole

Modeled by a Turing machine.
Input separated from output
Fixed input string
Input: one click.
Immediate reaction from computer: reactive systems.
Not an input string
Nowadays: interaction

Input: one click.
Immediate reaction from computer: reactive systems.
Not an input string
One Google query has different answers all the time: non-determinism
Nowadays: interaction

Input: one click.
Immediate reaction from computer: reactive systems.
Not an input string
One Google query has different answers all the time: non-determinism
A Turing machine cannot fly an airplane, but a computer can.
Foundations of computing is automata theory and formal languages.
Foundations of computing is automata theory and formal languages. Finite automaton, pushdown automaton, Turing machine: languages.
Foundations of computing is automata theory and formal languages. Finite automaton, pushdown automaton, Turing machine: languages. Interaction: from concurrency theory, process theory.
Differences

- Final state, termination, in concurrency
- Transition systems not necessarily finite
- Language equivalence cannot capture interaction.
  
  \[ a.b + a.c \neq a.(b + c) \]  
  (lady and the tiger)

Branching bisimulation.
Reachable nodes, unnamed nodes, layout
Language equivalence throws away a lot of information.
An algebra, for calculation

0 1  \[ a \cdot 0 + b \cdot a \cdot 1 \]
Structural Operational Semantics

\[ 1 \Downarrow \]
\[ a.x \xrightarrow{a} x \]
\[ \frac{x \xrightarrow{a} x'}{x + y \xrightarrow{a} x'} \]
\[ \frac{y + x \xrightarrow{a} x'}{x + y \xrightarrow{a} x'} \]
\[ \frac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y} \]
\[ \frac{y \xrightarrow{a} y'}{x \cdot y \xrightarrow{a} y'} \]
\[ \frac{y \xrightarrow{a} y'}{x \cdot y \xrightarrow{a} y'} \]
\[ \frac{x \Downarrow \quad y \Downarrow \quad x \cdot y \Downarrow}{x + y \Downarrow \quad y + x \Downarrow} \]
### Structural Operational Semantics

- \( x^* \downarrow \quad x \xrightarrow{a} x' \quad x \downarrow \quad y \downarrow \quad x \parallel y \downarrow \)
  
  \[ x \xrightarrow{a} x' \quad x^* \xrightarrow{a} x' \cdot x^* \quad x \xrightarrow{a} x' \quad \]
  
  \[ x \parallel y \xrightarrow{a} x' \parallel y \quad \text{and v.v.} \]
  
- \( x \xrightarrow{a} x' \quad x \parallel y \xrightarrow{a} x' \parallel y \)
  
  \[ x \xrightarrow{c!d} x' \quad y \xrightarrow{c?d} y' \quad x \parallel y \xrightarrow{c?d} x' \parallel y' \quad \text{and v.v.} \]
  
- \( x \xrightarrow{a} x' \quad \partial_c(x) \xrightarrow{a} \partial_c(x') \quad x \xrightarrow{c!d} x' \quad \partial_c(x) \downarrow \)
  
  \[ x \xrightarrow{a} x' \quad a \neq c!d, c?d \quad \partial_c(x) \xrightarrow{a} \partial_c(x') \]
  
  \[ x \xrightarrow{c?d} x' \quad \tau_c(x) \xrightarrow{\tau} \tau_c(x') \quad \tau_c(x) \xrightarrow{a} \tau_c(x') \quad \]
  
  \[ x \xrightarrow{a} x' \quad a \neq c?d \quad \tau_c(x) \xrightarrow{a} \tau_c(x') \quad x \xrightarrow{c?d} x' \quad \tau_c(x) \xrightarrow{\tau} \tau_c(x') \quad \tau_c(x) \downarrow \]
Presentation of a Grammar
From automaton to recursive specification

\[
S = a.T + a.W \\
T = a.U + b.W \\
U = b.V + b.R \\
V = 0 \\
W = a.R \\
R = b.W + 1
\]
From Recursive Specification to Automaton

\[
\begin{align*}
  t \xrightarrow{a} x & \quad P = t \\
  P \xrightarrow{a} x & \\
\end{align*}
\]

\[
\begin{align*}
  t \downarrow & \quad P = t \\
  P \downarrow & \\
\end{align*}
\]
Algebraic Laws

\[
\begin{align*}
  x + y & \equiv y + x \\
  (x + y) + z & \equiv x + (y + z) \\
  x + x & \equiv x \\
  x + 0 & \equiv x
\end{align*}
\]

\[
\begin{align*}
ax + ay & \approx a(x + y) \\
 a0 & \approx 0
\end{align*}
\]

- First four laws: isomorphic automata. But not a congruence!
- Distributive law: language preserving, removing non-determinism
- 0 is right-zero: only successful termination counts
Definition of Bisimulation

Bisimulation is the strongest congruence containing isomorphism.

Every congruence on automata containing isomorphism must contain bisimilarity.
Process Algebra

- 0  inaction, unsuccessful termination, deadlock
- 1  empty process, skip, successful termination
- $a._$  action prefix
- $_+_$  alternative composition, choice
- $_∥_$  parallel composition, merge, with communication
- $∂_c(_)$ encapsulation
- Recursive specifications, guarded
- $τ_c(_)$ abstraction
- Axiomatizations

Process Algebra

- 0 inaction, unsuccessful termination, deadlock
- 1 empty process, skip, successful termination
- $a._$ action prefix
- _ + _ alternative composition, choice
- _ || _ parallel composition, merge, with communication
- $\partial_c(_)$ encapsulation
- Recursive specifications, guarded
- $\tau_c(_)$ abstraction
- Axiomatizations

Removing non-determinism and totalizing

LNCS 5065, Montanari festschrift.
Removing non-determinism and totalizing

\[
\begin{align*}
S & = aU + aV \\
U & = bS \\
V & = bW + 1 \\
W & = 0.
\end{align*}
\]

\[
\begin{align*}
S & \approx aU + aV \approx a(U + V) + bX \\
U + V & \approx bS + bW + 1 \approx b(S + W) + 1 + aX \\
S + W & \approx aU + aV + 0 \approx a(U + V) + bX \\
X & = aX + bX
\end{align*}
\]
Removing silent steps
Removing silent steps

\[ S = aW + \tau T \]
\[ T = aU \]
\[ U = \tau U + bV + \tau T \]
\[ V = 0 \]
\[ W = 1 \]

\[ S \approx aW + \tau T \approx aW + T \approx aW + aU \]
\[ U \approx \tau U + bV + \tau T \approx U + bV + T \approx bV + aU \]
\[ V = 0 \]
\[ W = 1 \]
Regular expressions

\[ S \approx aT + 1 \approx a(bS + 1) + 1 \approx abS + a1 + 1 \approx (ab1)^* \cdot (a1 + 1) \]

\[ T \approx bS + 1 \approx b(aT + 1) + 1 \approx baT + b1 + 1 \approx (ba1)^* \cdot (b1 + 1) \]
A regular language is a language equivalence class of a finite (non-deterministic) automaton.
A regular language is a language equivalence class of a finite (non-deterministic) automaton. A regular process is a bisimulation equivalence class of a finite, non-deterministic automaton.
A regular language is a language equivalence class of a finite (non-deterministic) automaton.
A regular process is a bisimulation equivalence class of a finite, non-deterministic automaton.
A regular process is given by a recursive specification over the signature \(0, 1, a, _, +\). So right-linear but not left-linear.
A regular language is a language equivalence class of a finite (non-deterministic) automaton. A regular process is a bisimulation equivalence class of a finite, non-deterministic automaton. A regular process is given by a recursive specification over the signature $0, 1, a, \_+, +$. So right-linear but not left-linear. Processes given by deterministic automata, and by regular expressions, form a subclass. (B, Flavio Corradini, Clemens Grabmayer, JACM 2007.)
Regular expressions revisited

\[ s = (ts?b.(st!a.1 + 1))^* \]
\[ t = (st?a.(ts!b.1 + 1))^* \]
\[ \partial_{st,ts}((st!a.1 + 1) \cdot s \parallel 1 \cdot t) \]
Pushdown Process

\[ a[\varepsilon/1] \rightarrow \bigcirc \rightarrow a[1/11] \]

\[ b[1/\varepsilon] \rightarrow \bigcirc \rightarrow b[1/\varepsilon] \]

\[ a \rightarrow \bigcirc \rightarrow a \rightarrow \bigcirc \rightarrow a \rightarrow \bigcirc \rightarrow a \]

\[ b \rightarrow \bigcirc \rightarrow b \rightarrow \bigcirc \rightarrow b \rightarrow \bigcirc \rightarrow b \]
\[
S = 1 + \sum_{d \in \mathcal{D}} i?d.S \cdot o!d.S
\]
Example: $S = 1 + S \cdot a1$ has infinite branching. Has head recursion.
Example: $S = 1 + S \cdot a1$ has infinite branching. Has head recursion.
Unbounded branching

\[ X = aX \cdot Y + b1 \]
\[ Y = 1 + c1 \]

Variable \( Y \) is transparent.
This push-down automaton has no sequential recursive specification.
This push-down automaton has no sequential recursive specification. Using parallel composition, it has:

\[ S = c.1 + a.(S \parallel b.1) \]
A process is a popchoice-free pushdown process terminating only on empty
iff it is definable by a transparency-restricted sequential recursive
specification without head recursion.
Removing restrictions

When there is head recursion, can still find a push-down automaton in some cases. When there is transparency, we cannot. However, can get rid of both restrictions when we move to *contra-simulation*.

\[ ax + ay = a(\tau x + \tau y) \]

Theorem

For every pushdown automaton $M$ terminating only on empty there is a regular process $p$ and for every regular process $p$ there is an pushdown automaton $M$ terminating only on empty such that the transition system of $M$ is branching bisimilar with the transition system of $\tau_{i,o}(\partial_{i,o}(p \parallel S))$. 
Theorem

For every pushdown automaton $M$ terminating only on empty there is a regular process $p$ and for every regular process $p$ there is an pushdown automaton $M$ terminating only on empty such that the transition system of $M$ is branching bisimilar with the transition system of

$$\tau_{i,o}(\partial_{i,o}(p \parallel S))$$

The stack is the prototypical pushdown process. Constitutes a grammar for pushdown processes.
A parallel pushdown automaton gives a parallel pushdown process. A process $p$ is parallel pushdown iff there is a regular process $q$ with

$$p \equiv_b \tau_{i,o}(\partial_{i,o}(q \parallel B))$$

$B$ is the bag

$$B = 1 + \sum_{d \in D} i?d.(B \parallel o!d.1).$$

A basic parallel process is given by a rec.spec. over $1$, $0$, $+$, $a_-$, $\parallel$. Any basic parallel process is a parallel pushdown process.
Example

\[ X = c.1 + a.(X \parallel b.1) \]

is basic parallel, parallel pushdown and pushdown but not sequential.
\[ X = c.1 + a.(X \parallel b.1) \]

is basic parallel, parallel pushdown and pushdown but not sequential.

The bag is basic parallel, parallel pushdown but not pushdown, not sequential.
The stack is sequential and pushdown but not basic parallel, parallel pushdown.
Pushdown and parallel pushdown process that is not basic parallel or sequential.
Can be adapted to become pushdown and not parallel pushdown, or parallel pushdown and not pushdown.
Process Classes

- **unbounded**
- **sequential**
- **regular**

- **PDP**
- **PPDP**
- **BPP**

- **executable**
Reactive Turing Machine

$\tau[\square/\square]L \quad \tau[\square/\square]R$

$a[\Box/1]R \quad b[1/\Box]L$

$\tau \quad \tau \quad \tau \quad \tau$

$\tau \quad \tau \quad \tau \quad \tau$

$\tau \quad \Box \quad \Box \quad \Box$
Queue that can always terminate

\[\begin{align*}
o!n[n/\square]L & \quad \tau[n/n]L \\
\tau[n/n]L & \quad \tau[n/n]L \\
\tau[\square/0]R & \quad \tau[\square/1]R \\
i?1[n/n]L & \quad \tau[\square/\square]L \\
i?n[\square/n]R & \quad \tau[\square/\square]L \\
i?0[n/n]L & \quad \tau[\square/0]R
\end{align*}\]
An executable process is the branching bisimulation equivalence class of a transition system of a Reactive Turing Machine.
An executable process is the branching bisimulation equivalence class of a transition system of a Reactive Turing Machine. The queue is executable but not push-down.
An executable process is the branching bisimulation equivalence class of a transition system of a Reactive Turing Machine. The queue is executable but not push-down. A transition system is computable iff it is finitely branching and (with some coding) the set of final states is decidable and for each state, we can determine the set of outgoing transitions.
An executable process is the branching bisimulation equivalence class of a transition system of a Reactive Turing Machine. The queue is executable but not push-down. A transition system is computable iff it is finitely branching and (with some coding) the set of final states is decidable and for each state, we can determine the set of outgoing transitions. A transition system is effective if its set of transitions and set of final states are recursively enumerable.
Results

The transition system defined by a Reactive Turing Machine is computable.

Every effective transition system is branching bisimilar with a transition system of a Reactive Turing Machine.

For every RTM $M$ there is a regular process $p$ and for every regular process $p$ there is an RTM $M$ such that the transition system of $M$ is branching bisimilar with the transition system of

$$\tau_{i,o}(\partial_{i,o}(p \parallel Q^{io}))$$
One queue is two chained queues

\[ Q^{io} = 1 + \sum_{d \in D} i?d.\tau_i(\partial_i(Q^{il} \parallel (1 + o!d.Q^{lo}))) \]

\[ Q^{il} = 1 + \sum_{d \in D} i?d.\tau_o(\partial_o(Q^{io} \parallel (1 + l!d.Q^{ol}))) \]

\[ Q^{lo} = 1 + \sum_{d \in D} l?d.\tau_i(\partial_i(Q^{li} \parallel (1 + o!d.Q^{io}))) \]

\[ Q^{ol} = 1 + \sum_{d \in D} o?d.\tau_i(\partial_i(Q^{oi} \parallel (1 + l!d.Q^{il}))) \]

\[ Q^{li} = 1 + \sum_{d \in D} l?d.\tau_o(\partial_o(Q^{lo} \parallel (1 + i!d.Q^{oi}))) \]

\[ Q^{oi} = 1 + \sum_{d \in D} o?d.\tau_l(\partial_l(Q^{ol} \parallel (1 + i!d.Q^{li}))) \]
Integration of automata theory and process theory is beneficial for both theories.
Integration of automata theory and process theory is beneficial for both theories. This integrated theory can be a first-year course in any academic bachelor program in computer science (or related subjects).
Integration of automata theory and process theory is beneficial for both theories. This integrated theory can be a first-year course in any academic bachelor program in computer science (or related subjects). Draft syllabus available.
Integration of automata theory and process theory is beneficial for both theories. This integrated theory can be a first-year course in any academic bachelor program in computer science (or related subjects). Draft syllabus available. This provides a basic course in theoretical computer science as always, but at the same time gives a foundation for later courses on formal methods and model checking.
Integration of automata theory and process theory is beneficial for both theories.
This integrated theory can be a first-year course in any academic bachelor program in computer science (or related subjects).
Draft syllabus available.
This provides a basic course in theoretical computer science as always, but at the same time gives a foundation for later courses on formal methods and model checking.
We defined a grammar for all executable processes.
Computer science is the study of discrete behavior of interacting information processing agents.