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 abstration**
  - 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?

\[
CD = \text{card}_\text{in} \rightarrow \text{request}?\text{amount} \rightarrow ( \\
\text{card}_\text{out} \rightarrow \text{money}!\text{amount} \rightarrow CD \\
\cap \text{no}_\text{money} \rightarrow \text{card}_\text{out} \rightarrow CD 
)\]

- How do I understand this?

\[
\bot_\mathcal{N} = (\Sigma^* \times \mathbb{P}(\Sigma^\vee), \Sigma^* \vee)\]

- What observational congruence?

- How to present state information?
  “in\_critical” ~ “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

$\Rightarrow$ Clumsy "save"-mechanism: read first not saved item from the queue
$\Rightarrow$ 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

<table>
<thead>
<tr>
<th>place</th>
<th>TUT</th>
<th>TUT</th>
<th>Hels</th>
<th>TUT</th>
<th>Hels</th>
<th>TUT</th>
<th>TUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>passed students</td>
<td>12</td>
<td>13</td>
<td>6</td>
<td>6</td>
<td>9</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

- 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

- Card_in
- Request \( ? \text{amount} \)
- Money \(! \text{amount} \)
- Say "no money"
- Card_out

- Enquiry \(! \text{amount} \)
- In\( ? \text{x} \)
- Out\( ! \text{x} \)
- Enquiry \(? \text{amount} \)

- FIFO2 \( \langle N \rangle \)
- In\( 0 \)
- Out\( 0 \)
- Balance_ok
- Bad_balance

- FIFO2 \( \langle \{0,1\} \rangle \)
- In\( 1 \)
- Out\( 1 \)
- Balance_ok
- Bad_balance

- CASH DISPENSER
- BANK
- Salary \(? \text{amount} \)

2006–06–27
Individual state machines ...

CASH DISPENSER

\[ \text{amount} : \mathbb{N} \]

- money \( \text{!amount} \)
- card in
- card out
- request \( \?\text{amount} \)
- say \( \text{!"no money"} \)
- enquiry \( \text{!amount} \)
- bad_balance
- balance_ok

\[
\begin{align*}
\text{FIFO2}\langle \text{TYPE} \rangle \\
x, y : \text{TYPE}
\end{align*}
\]
... Individual state machines

\[ \text{balance} : \mathbb{N} \]
\[ \text{amount} : \mathbb{N} \]

- \( \text{enquiry} ? \text{amount} \)
- \( \text{balance} := 0 \)
- \( \text{salary} ? \text{amount} \)
- \( \text{balance} \pm= \text{amount} \)
- \( \text{balance} \leq \text{balance} \rightarrow \text{balance}_\text{ok} \)
- \( \text{balance} \not\leq \text{balance} \rightarrow \text{bad}_\text{balance} \)
- \( \text{balance} := \text{amount} \)
- \( \text{balance} \pm= \text{amount} \)

Diagram:

- START STATE
- Enquiry state
- Salary state
- Balance state
- Bad balance state
- Balance ok state

Transition labels:
- \( \text{amount} \leq \text{balance} \)
- \( \text{amount} > \text{balance} \)

Conditions:
- \( \text{balance} +\text{amount} \)
- \( \text{amount} \leq \text{balance} \)
- \( \text{balance} \not\leq \text{balance} \)

Variables:
- \( \text{balance} \)
- \( \text{amount} \)

Initial state:
- \( \text{balance} := 0 \)

Final state:
- \( \text{balance} \)
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

\[
\text{chatter } ?x \ [x \leq 10] \ || \ \text{chatter } ?a \ [a \geq 2]
\]
... Input and output

- Communication of several values in one action

\[
\text{chatter } ?x !y !(y+1) \parallel \text{chatter } !3 ?a !2 \cdot b [a \geq 2]
\]

⇒ 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} \rightarrow \text{output} . . .

\[ \text{result} := \text{input1} + \text{input2} \]

Diagram:
- Nodes: 1 2, 1 1, 3 4
- Edges:
  - 1 2 to 3
  - 1 1 to 2
  - 3 4 to 7
... Transition relations ...

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

\[
\text{result} := \frac{1}{\text{input}}
\]

Diagram showing transition relations with states 0, 1, 2, and results 0.5 and 1.0.
... Transition relations ...

- ... oops, it is a relation $R(\text{input}, \text{output})$

```
result := input + ++input
```

![Diagram showing transition relations between states](image-url)
... 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

\[(S, VC, \Sigma, \Delta, \hat{S}, \text{Init}, \Pi, \text{val})\]

- Some details
  - \( VC \) = all potential combinations of variable values
  - \( \Sigma \) = gate names
  - \( \text{Init} \subseteq \hat{S} \times VC \) specifies initial values of variables

- Semantics is obtained via the unfolding construction
  - \( U \) = universe of communicated data values
  - \( \hat{S}_U = \{ \hat{s}\langle \bar{v} \rangle | (\hat{s}, \bar{v}) \in \text{Init} \} \)
  - \( \Sigma_U = \Sigma \times U^* \)
  - \( \Delta_U \) is those reachable \( s\langle \bar{v} \rangle \stackrel{a\langle \bar{p} \rangle \rightarrow s'\langle \bar{v}' \rangle}{\text{for which}} \exists (s, a, R, s') \in \Delta : (\bar{v}, \bar{p}, \bar{v}') \in R \)
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:

\[ \text{Parallel composition can also be computed with variables present.} \]

- 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!