Introduction to Process Algebra

Moonzoo Kim
CS Dept. KAIST
Fall 2008
We have seen tragic accidents due to software and specification bugs.

These bugs are hard to find because those bugs occur only in “exceptional” cases.

Informal system specification and requirement specification makes automatic analysis infeasible, which results in incomplete coverage.

To provide better coverage, we need:
- Formal requirement specification
- Formal system model

\[ \square (\Phi \rightarrow \Diamond \Omega) \]
Outline

- Requirement specification problems
- Viewpoint on “meaning” (semantics) of system
- Complexity of a system
- Formal modeling v.s. programming
- Introduction to process algebra
Requirement Specification Problems

- **Ambiguity**
  - Expression does not have unique meaning, but can be interpreted as several different meaning.
    - Ex. `long` type in C programming language

- **Incompleteness**
  - Relevant issues are not addressed, e.g. what to do when user errors occur or software faults show.
    - Ex. See next slides

- **Inconsistency**
  - Contradictory requirements in different parts of the specification.
Example (retail chain management software)

If the sales for the current month are below the target sales, then a report is to be printed,

unless the difference between target sales and actual sales is less than half of the difference between target sales and actual sales in the previous month

or if the difference between target sales and actual sales for the current month is under 5 percent.
A system execution $\sigma$ is a sequence of states $s_0s_1\ldots$

- A state has an environment $\rho_s: \text{Var} \rightarrow \text{Val}$

A system has its semantics as a set of system executions
The complexity of a system is sometimes more accurately expressed using semantic viewpoint (number of reachable states) rather than syntactic viewpoint (line number of source code).

- The number of different states a system can reach.
  - Ex: An integer has $2^{32}$ (~4000000000) possible values.
active type A() {
 byte x;
 again:
   x++;
   goto again;
}

active type A() {
 byte x;
 again:
   x++;
   goto again;
}

active type B() {
 byte y;
 again:
   y++;
   goto again;
}
## Formal Modeling V.S. Programming

<table>
<thead>
<tr>
<th>Static Aspects</th>
<th>Formal Modeling</th>
<th>Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstraction Level</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Development Time</td>
<td>Short</td>
<td>Long</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dynamic Aspects</th>
<th>Formal Modeling</th>
<th>Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executable</td>
<td>Yes (model checking) No (theorem proving)</td>
<td>Always</td>
</tr>
<tr>
<td>System Semantics</td>
<td>Mathematically defined</td>
<td>Usually given by examples</td>
</tr>
<tr>
<td>Environment Semantics (i.e. testbeds)</td>
<td>Mathematically defined</td>
<td>Usually given by examples</td>
</tr>
<tr>
<td>Program State Space</td>
<td>Manageable (i.e. tractable state space)</td>
<td>Unmanageable (i.e. beyond computing power)</td>
</tr>
<tr>
<td>Validation</td>
<td>By exhaustive exploration or deductive proof</td>
<td>By testing (incomplete coverage)</td>
</tr>
</tbody>
</table>
You may not need to model a simple system such as +, *, or HelloWorld.

However, you must have a scientific way of abstracting/modeling a system with complex structure, e.g.,

- Hierarchy
- Concurrency
- Communication

Also, you need to have a systematic way to analyze the correctness of your design.
A process algebra consists of:
- a set of operators and syntactic rules for constructing processes
- a semantic mapping which assigns meaning or interpretation to every process
- a notion of equivalence or partial order between processes

Advantages: A large system can be broken into simpler subsystems and then proved correct in a modular fashion.
- A hiding or restriction operator allows one to abstract away unnecessary details.
- Equality for the process algebra is also a congruence relation; and thus, allows the substitution of one component with another equal component in large systems.

Note that the model is constructed in a component-based way, but the analysis is not.
Calculus of Communicating Systems (CCS)

- Developed by R. Milner (Univ. of Cambridge)
  - ACM Turing Award 1991
- Provides many interesting paradigms
  - Emphasis on communication and concurrency
    - Provides compact representation on both communication and concurrency
      - Ex> a (receive) and a’ (send)
      - Ex> | (parallel operator)
  - Provides observation based abstraction
    - Hiding internal behaviors using \ (restriction) operator, i.e., considering all internal behaviors as an invisible special action τ
  - Provides correctness claim based on equivalence
    - Branching time based equivalence
      - Strong equivalence v.s. weak equivalence
Overview on CCS Syntax and Semantics

- CCS describes a system as a set of communicating Processes.
- Behavior of a process is expressed using actions:
  - \( \text{Act} = \text{input\_actions} \cup \text{output\_actions} \cup \{\tau}\)
- Each process is built based on the following 7 operators:
  - Nil (null-ary operator): \(0\)
  - Prefix: \(a.P\)
  - Definition: \(P = a.b.Q\)
  - Choice: \(a.P + b.P\)
  - Parallel: \(P | Q\)
  - Restriction: \(P \setminus \{a,b\}\)
  - Relabelling: \(P[a/b]\)
- Each operator has a clear formal semantics via inference rules (premises-conclusion rules).
  - Based on these inference rules, a meaning/semantics of a process is given as a labelled transition system.
A set of actions Act = \{a, a', b, \tau\}

We define a CCS system Sys as

Sys = (a.E + b.0) \mid a'.F

Sys can executes one of the following 4 actions

- Sys \xrightarrow{a} E \mid a'F
- Sys \xrightarrow{a'} (a.E + b.0)|F
- Sys \xrightarrow{b} 0 \mid a'.F
- Sys - \tau \rightarrow E|F

Prefix: \(a.E \xrightarrow{a} E\)

Choice\(_L\): \((a.E + b.0)) \xrightarrow{a} E\)

Par\(_L\): \((a.E + b.0)) \mid a'.F \xrightarrow{a} E \mid a'.F\)