Concurrency and Parallelism

Master 1 International

Andrea G. B. Tettamanzi
Université de Nice Sophia Antipolis
Département Informatique
andrea.tettamanzi@unice.fr
Lecture 5

Theoretical Models
Table of Contents

- Introduction
- Petri Nets
- Actor Model Theory
- Traces and Trace Theory
- Lamport's TLA+ Logic
- Process Calculi:
  - Calculus of Communicating Systems (CCS)
  - Communicating Sequential Processes (CSP)
  - π-Calculus
Introduction

• Fundamental problems:
  – Primary: establishing the equivalence of programs
  – Secondary: proving other interesting properties

• A model provides an abstract view in which the “irrelevant” details are ignored in establishing the equivalence of systems

• A denotational model is one in which the meaning of a system can be derived from its constituent parts (compositionality)

• For sequential programming, computer scientists have been successful in building denotational models of programs which abstract away the operational details

• For concurrent programming, it is harder to come up with such models, mainly due to interleaving
Petri Nets

- Introduced by Carl Adam Petri
- A mathematical modeling language for the description of distributed systems
- A bipartite graph consisting of places, transitions, and arcs
- Places contain a discrete number of tokens.
- A distribution of tokens over the places is called a marking.
- A transition may fire whenever its input places contain sufficient tokens.
- Firing is atomic. Upon firing, a transition
  - consumes tokens in its input places
  - places tokens in its output places.
- Execution of Petri nets is nondeterministic
**Petri Nets: Typical Interpretations**

<table>
<thead>
<tr>
<th>Input Places</th>
<th>Transition</th>
<th>Output Places</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preconditions</td>
<td>Event</td>
<td>Postcondition</td>
</tr>
<tr>
<td>Input data</td>
<td>Computation step</td>
<td>Output data</td>
</tr>
<tr>
<td>Input signals</td>
<td>Signal processor</td>
<td>Output signal</td>
</tr>
<tr>
<td>Resources needed</td>
<td>Task or job</td>
<td>Resources released</td>
</tr>
<tr>
<td>Conditions</td>
<td>Logical clause</td>
<td>Conclusion(s)</td>
</tr>
<tr>
<td>Buffers</td>
<td>Processor</td>
<td>Buffers</td>
</tr>
</tbody>
</table>
Petri Net: Formal Definition

- A Petri net is a 5-tuple PN = (P, T, F, W, M₀) where:
  - P = {p₁, p₂, ..., pₘ} is a finite set of places,
  - T = {t₁, t₂, ..., tₘ} is a finite set of transitions,
  - F ⊆ (P × T) ∪ (T × P) is a set of arcs (flow relation),
  - W : F → {1, 2, 3, ...} is a weigh function,
  - M₀ : P → {0, 1, 2, 3, ...} is the initial marking,
  - P ∩ T = ∅ and P ∪ T ≠ ∅.
- A Petri net structure N = (P, T, F, W) without any specific initial marking is denoted by N
- A Petri net structure N with marking M is denoted (N, M)
Petri Nets: Behavioral Properties

- Reachability: M is reachable iff \( \exists \sigma : M_0 [\sigma > M] \)
  - \( R(M_0) \) is the set of reachable markings
  - Reachability problem: decidable, but EXPSPACE

- Boundedness: no. of tokens in each place is bounded for any reachable marking; k-bounded: \( \leq k \); 1-bounded = safe

- Liveness: at any marking M, any transition may eventually fire
  - A transition is Lk-live if it may fire in k time steps (weaker)

- Reversibility: from any M it is possible to reach a home state \( M' \)

- Synchronic Distance between two transitions:
  - \( d(t_1, t_2) = \max_{\sigma} | N_\sigma(t_1) - N_\sigma(t_2) | \), where \( N_\sigma(t) = \) firings of \( t \) in \( \sigma \)
Actor Model

- A mathematical model of concurrent computation
- Proposed by Carl Hewitt in 1973
- Studied by Gul Agha in his PhD Thesis at MIT (1985)
- Actors are the universal primitives of parallel computation
- Actors = Processes
- Actors exchange messages asynchronously and create other actors
- Abstract actor machine with a minimal programming language
- To send a communication, an actor specifies the target
- Communications are buffered and eventually delivered
- Denotational semantics based on transitions
Simple Actor Language (SAL)

- `<behavior definition> ::= def <beh name> (<acquaintance list>) [<communication list>] <command>* end def`
- `<parameter list> ::= {id | <var list> } | {, id | , <var list> } | ε`
- `<var list> ::= case <tag field> of <variant>+ end case`
- `<variant> ::= <label> : <parameter list>`
- `<command> ::= if <condition> then <command> { else <command> } fi | become <expression> | send <msg> to <target> | <let bindings> "{<command> "}" | `<behavior definition> | <command>*`
Denotational Semantics (1)

- Tasks: communications which are still pending (not yet accepted)
  \[ \text{task} = (\text{tag}, \text{target}, \text{msg} = [\text{value}_1, \text{value}_2, \ldots, \text{value}_n]) \]

- Local states function \( l : \text{target} \rightarrow \text{behavior} \)

- **Configuration** of an actor system: \( c = (\text{local states fn, tasks}) \)

- Behavior: \( \text{msg} \rightarrow (\text{new tasks, new actors, replacement behavior}) \)

- Actor: \( (\text{mail address (= to be used as target), behavior}) \)

- The behavior of an actor whose mail address is \( m \) is a function \( (\text{tag, m, msg}) \rightarrow (\text{set of tasks, set of actors, replacement actor}) \)

- Depending on the incoming communication \( (\text{tag, m, [v}_1, \ldots, \text{v}_n]) \), send communications to specified targets (1) creating new actors and (2) specifying a replacement actor machine
Denotational Semantics (2)

Transition:
\[
\tau \quad \text{from} \quad c \quad \text{to} \quad c'
\]

Task \( \tau = (t, m, k) \)

Configurations

\(
\tau \in \text{tasks}(c)
\)

\(
\text{states}(c)(m) = \beta, \text{where } \beta(t, m, k) = (T, A, \gamma)
\)

\(
\text{tasks}(c') = (\text{tasks}(c) - \{\text{tau}\}) \cup T
\)

\(
\text{states}(c') = (\text{states}(c') - \{(m, \beta)\}) \cup A \cup \{\gamma\}
\)

Subsequent transition:
\[
\tau \quad \text{from} \quad c \quad \text{to} \quad c'
\]

\(
\tau \in \text{tasks}(c) \land c \rightarrow^* c' \land \tau \not\in \text{tasks}(c') \land
\neg \exists c''(\tau \not\in \text{tasks}(c'') \land c \rightarrow^* c'' \land c'' \rightarrow^* c')
\)
Actor Model

• The Actor Model provides solution to three central problems in distributed computing:
  – Divergence (= infinite loops), thanks to the “guarantee of mail delivery”
  – Deadlock (cf. “the five dining philosophers”)
    • No syntactic (= low-level) deadlock possible
    • Semantic deadlocks are possible, but may be detected
    • Solve detected deadlock by negotiation
  – Mutual exclusion: not really a problem for actors
    • An actor can be “accessed” only by sending it some mail
    • An actor accepts just one mail and specifies a replacement that will accept the next mail in queue
Trace Theory

- Finite automata are a convenient model of sequential programs
- Automata admit powerful analysis tools
  - Structural properties: underlying graph-like model
  - Behavioral properties: formal language theory
- Basic idea: use well-developed tools from formal language theory for the analysis of concurrent systems
- A concurrent system is understood as in the theory of Petri nets
- The algebra of dependency graphs (such as Petri nets) is isomorphic to that of trace monoids
- Alternative to concurrency as interleaving non-determinism
- First formulated by Antoni Mazurkiewicz in the 1970s
Traces

- Loosely speaking, a trace is an equivalence class of strings which differ only in the ordering of adjacent independent symbols.
- Dependency relation $D$: if $a \ D \ b$, then $b \ D \ a$ and $a \ D \ a$; $a \equiv_D b$ iff $\neg(a \ D \ b)$: symbols $a$ and $b$ are independent.
- The set $\Sigma^*$ of strings on alphabet $\Sigma$ is a monoid w.r.t. the operation $\cdot$ of concatenation.
- The trace monoid $M(D)$ is the quotient monoid $\Sigma^*_D / \equiv_D$.
- Given a string $w$, $[w]_D$ is the trace represented by string $w$.
- A dependency graph $G(D)$ is a graphical representation of dependency relation $D$. $G(D)$ is isomorphic to $M(D)$.
Histories

- Given $n$ processes, each with its own alphabet $\Sigma_i$
- An elementary history $\pi(a)$ is an $n$-tuple consisting of one-symbol strings $a$ in positions where $a \in \Sigma_i$, the empty string $\varepsilon$ elsewhere
- A history is a concatenation of elementary histories
- The monoid of histories $H(\Sigma_1, \Sigma_2, \ldots, \Sigma_n)$ is isomorphic with the monoid of traces over dependency $\Sigma_1^2 \cup \Sigma_2^2 \cup \ldots \cup \Sigma_n^2$.
- An ordering of events may be established given a history
Trace Languages

- Trace language over D: any set of traces over D
- Trace projection of trace t onto dependency C: $\pi_C(t)$
- The synchronization of string language $L_1$ over $\Sigma_1$ with the string language $L_2$ over $\Sigma_2$ is defined as $(L_1 \parallel L_2)$ over $(\Sigma_1 \parallel \Sigma_2)$, such that $w \in (L_1 \parallel L_2)$ iff $\pi_{\Sigma_1}(w) \in L_1$ and $\land \pi_{\Sigma_2}(w) \in L_2$. 
**TLA+ Logic**

- Temporal Logic of Actions, developed by Leslie Lamport
- TLA+ combines temporal logic with a logic of actions
- TLA+ formulas describe the behavior of a system
- Temporal aspect: primed and non-primed variables:
  - Non primed, x, means “the current value of x”
  - Primed, x', means “the value of x at the next step”
- Action: a Boolean formula containing constants, variables, and primed variables
- State function: an expression containing constants and non-primed variables only
- Action A is enabled in state s iff there exists a state t such that (old-state s, new state t) satisfies A
**TLA Syntax**

- **P**: satisfied iff true for the initial state
- **[A]_f**: satisfied iff every step satisfies A or leaves f unchanged
- **□F**: satisfied if F is always true
- **WF_f(A)**: weak fairness of A: if A \( \land (f' \neq f) \) ever becomes enabled and remains enabled forever, then infinitely many A \( \land (f' \neq f) \) steps occur
- **SF_f(A)**: strong fairness of A: if A \( \land (f' \neq f) \) is enabled infinitely often, then infinitely many A \( \land (f' \neq f) \) steps occur
- **F \rightarrow^{+} G**: G is true for at least as long as F is
- **◊F**: F is eventually true, equivalent to \( \neg \Box \neg F \)
- **F \sim G**: F leads to G: whenever F, eventually G: \( \Box (F \Rightarrow \Diamond G) \)
A process calculus or process algebra is a tool for the formal modeling of a concurrent system.

A process calculus comprises:
- tools for the high-level description of interactions, communications, and synchronizations between processes
- algebraic laws that allow to manipulate descriptions and prove equivalences between processes

Three very influential process calculi:
- Calculus of Communicating Systems (CCS)
- Communicating Sequential Processes (CSP)
- π-Calculus
Calculus of Communicating Systems

- Introduced by Robin Milner around 1980
- Syntax: $P ::= \emptyset | a.P | A | P_1 + P_2 | P_1 | P_2 | P_1[b/a] | P_1\backslash a$
  
  - $\emptyset$ is the empty process
  - Process $a.P$ can perform action $a$ and continue as $P$
  - $A = P$ defines identifier $A$ that refers to process $P$
  - $P_1 + P_2$ is the non-deterministic choice between $P_1$ and $P_2$
  - $P_1 | P_2$ means the two processes are executed concurrently
  - $P[b/a]$ is process $P$ with all actions $a$ replaced by $b$
  - $P\backslash a$ is process $P$ without action $a$
Communicating Sequential Processes

- Introduced by Sir C. A. R Hoare in 1978
- Originally designed as a concurrent programming language
- Then refined into an algebraic theory
- Used for specification and verification of concurrent systems
- Has enjoyed some success in industrial applications
- Focus on dependable and safety-critical systems
- CSP describes systems in terms of component processes that
  - Operate independently
  - Interact through message passing
- Processes may be defined both as sequential processes or as the parallel composition of more primitive processes
CSP Primitives

- Events: communications or interactions:
  - Atomic names (e.g., on, off)
  - Compound names (e.g., valve.open, valve.close)
  - Input events, ? = “reads” (e.g., mouse?xy)
  - Output events, ! = “writes” (e.g., terminal!message)

- Primitive processes, representing fundamental behaviors
  - STOP, the deadlock process
  - SKIP, successful termination
### CSP Algebraic Operators

- **a → P** [prefix]  
  Wait for event a, then proceed as P

- **(a → P) □ (b → Q)** [deterministic choice]  
  - if event a then P, else, if event b then Q

- **(a → P) π (b → Q)** [nondeterministic choice]  
  - either a → P or b → Q

- **P ||| Q** [interleaving]  
  P and Q in parallel with interleaving

- **P |[X]| Q** [interface parallel]  
  - P and Q can proceed only after they both accept the same event in X

- **P \ X** [hiding]  
  - execute P after removing any occurrence of the events in X
CSP Syntax

- Proc ::= STOP | SKIP
  | e → Proc (prefixing)
  | Proc □ Proc (external choice)
  | Proc π Proc (nondeterministic choice)
  | Proc ||| Proc (interleaving)
  | Proc \ X (hiding)
  | Proc ; Proc (sequential composition)
  | if b then Proc else Proc (Boolean conditional)
  | Proc ▷ Proc (timeout)
  | Proc △ Proc (interrupt)
Semantics

• The CSP syntax may be given several different formal semantics
• Operational Semantics: meaning given in terms of operations
• Algebraic semantics
• Denotational semantics
  – Traces model, based on trace theory
  – Stable failures model
  – Failures/divergence model
**$\pi$-Calculus**

- May be regarded as a continuation of CCS
- Parallel counterpart of $\lambda$-calculus
- The syntax of $\pi$-calculus allows one to represent
  - parallel composition of processes,
  - synchronous communication between processes through channels,
  - creation of new channels,
  - replication of processes
  - nondeterminism.
- Process: an abstraction of an independent thread of control
- Channel: an abstraction of a communication link b/w processes
**Syntax of $\pi$-Calculus**

Let $P$ and $Q$ denote processes. Then

- $P | Q$ denotes a process composed of $P$ and $Q$ running in parallel.
- $a(x).P$ denotes a process that waits to read a value $x$ from the channel $a$ and then, having received it, behaves like $P$.
- $\overline{a}<x>.P$ denotes a process that first waits to send the value $x$ along the channel $a$ and then, after $x$ has been accepted by some input process, behaves like $P$.
- $(\nu a)P$ ensures that $a$ is a new channel in $P$.
- $!P$ denotes an infinite number of copies of $P$, all running in parallel.
- $P + Q$ denotes a process that behaves like either $P$ or $Q$.
- $0$ denotes the inert process that does nothing.
\textit{\textit{n-Calculus Example}}

Client-server communication:

\[ !\text{incr}(a, x).\overline{a<x+1>} \mid (\nu \text{res})(\overline{\text{incr}<\text{res}, 17>} \mid \text{res}(y)) \]

Infinite copies of a server accept messages on a channel called “incr” containing a channel name \( a \) and a number \( x \), then send on channel \( a \) the result of computing \( x + 1 \).

In parallel, a client creates a new channel called “res” and sends a message containing channel name “res” and 17 to the channel called “incr”; at the same time, it accepts messages containing the result, \( y \), on channel “res”
Congruence

Structural congruence is the least equivalence relation preserved by the process constructs and satisfying:

- $P \equiv Q$, if $Q$ can be obtained from $P$ by renaming bound names
- $P \mid Q \equiv Q \mid P$
- $P + Q \equiv Q + P$
- $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$
- $P \mid 0 \equiv P$
- $(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$
- $(\nu x)0 \equiv 0$
- $!P \equiv P \mid !P$
- $(\nu x)(P \mid Q) \equiv (\nu x)P \mid Q$, if $x$ is not a free name in $Q$
Reduction Semantics

Reduction relation: \( P \rightarrow P' \) means \( P \) can become \( P' \) after performing a computation step.

\( \rightarrow \) is defined as the least relation closed under the rules:

- \( \bar{a} <x>.P | a(y).Q \rightarrow P | Q[x/y] \)
- If \( P \rightarrow Q \), then also \( P | R \rightarrow Q | R \)
- If \( P \rightarrow P' \) and \( Q \rightarrow Q' \), then also \( P + Q \rightarrow P' \) and \( P + Q \rightarrow Q' \)
- If \( P \equiv P' \) and \( P' \rightarrow Q' \), then also \( (\nu x)P \rightarrow (\nu x)Q \)
- If \( P \equiv P' \) and \( P' \rightarrow Q' \) and \( Q' \equiv Q \), then \( P \rightarrow Q \)
Thank you for your attention