A way to model concurrent systems
such as those coded with multithreaded Java code
Developed by J. Magee and J. Kramer of Imperial College
In their book "Concurrency: State Machines and Java Programs" (ISBN: 0471986710)
Supported by a tool: LTSA
Download from www-dse.doc.ic.ac.uk/concurrency
Founded on Hoare’s CSP model with simplifications to allow automated analysis

Design approach
- Produce a high-level design
  - much more abstract than code
  - captures key patterns of interaction
- Explore its properties
  - check it avoids errors
    - (normally, it doesn’t; re-design and re-check!)
- Write code that follows the design

Discrete models
- Real systems can change continuously
  - eg: furnace’s temperature
  - eg: car’s position
  - eg (effectively): thread’s program counter
- Model concentrates on major alterations
  - happen in events at widely separated and clearly demarcated times
  - eg: furnace switched on
  - eg: car brake applied
  - eg thread is dispatched

Overview of an FSP Process
- Key concepts
  - action
  - trace
  - set of traces
  - process syntax (FSP)
  - state machine (LTS)

Action
- Each possible event of importance can be identified and named
  - interaction of system with its environment
    - environment does something to system
    - environment observes some aspect of system
  - system does something to environment
    - internal transformation of system
  - Action name starting in lower case
    - later we’ll see structured names too
Action examples

- Timebomb
  - arm
  - disarm
  - tick
  - boom

- drink machine
  - deposit1dollar
  - deposit2dollars
  - pushcancel
  - pushsodabutton
  - pushjuicebutton
  - return1dollar
  - return2dollars
  - providesoda
  - providejuice

Trace

- A system executes by taking actions one after another in a sequence
  - what an observer would notice
  - called a trace of the system
  - may be finite (stops eventually) or infinite
- Eg timebomb does
  - arm -> tick -> tick -> disarm
- Eg drink machine does
  - deposit2dollars -> pushjuicebutton -> providejuice
    -> return1dollar -> deposit1dollar ->
    pushsodabutton -> providesoda -> ...

Set of traces

- The same system may execute in different ways on different occasions
  - eg timebomb may also do
    - arm -> tick -> tick -> tick -> tick -> boom
- We regard the important model as the set of all possible traces that the system could do
  - "system is correct" means that every trace is acceptable
  - system's not correct if some traces are acceptable and some are not!

Why a set?

- In different executions, environment does different things
  - eg user can deposit1dollar or deposit2dollars
  - this leads system to act differently thereafter
- Also, system may have internal non-determinism
  - random choices
  - unpredictable delays, loss etc

Representing a set of traces

- Different ways to describe a set of sequences
  - List the sequences
    - \{ a->b, a->b->a->b, a->b->a->b->a->b \}
    - establish pattern and rely on reader to continue it
  - Explain in words
    - do a then b and repeat up to twice more
  - Use regular expressions
    - Studied in subjects on theory of computation, formal languages
Warnings
- The set may have a finite or infinite number of traces in it
  - It is possible to have a set with just one trace
- Some traces in the set may be finite while others are infinite

Precision matters
- "set of all finite sequences of a’s" includes an infinite number of sequences such as
  - $a \rightarrow a \rightarrow a$, $a \rightarrow a \rightarrow a \rightarrow a$, etc.
  - but it does not include the infinite sequence $a \rightarrow a \rightarrow a \rightarrow a \rightarrow \ldots$

Process syntax
- We represent a system by a process
  - a piece of text in a precise notation (called FSP)
- The LTSA tool from Imperial College can check that the syntax is correct

Overview of FSP
- A process is named, beginning in UPPER CASE
- Give a collection of equations, separated by commas, each defining a process in terms of expressions built up from actions and from other processes
  - recursion is allowed (right-hand side can refer to same process as left-hand side, or several processes can be defined in terms of one another)

Process example
- The following defines the process $P_1$, using $P_2$ and $P_3$ as auxiliary names
  - $P_1 = (a \rightarrow P_2 | b \rightarrow P_3)$
  - $P_2 = (x \rightarrow y \rightarrow \text{STOP})$
  - $P_3 = (z \rightarrow P_1)$

Warning
- The syntax is constrained by subtle rules. The following is NOT legal:
  - $P_1 = (a \rightarrow b | P_2)$
  - $P_2 = a \rightarrow P_3$
  - $P_3 = P_2 \rightarrow \text{STOP}$
State machines
To understand how a process executes, we can build an alternative representation as a state machine or graph (called a labeled transition system or LTS).
- The LTSA tool does this automatically
- Node corresponds to a possible state of the system
- Edge corresponds to the system changing from one state to another in an event
- Label the edge with the name of the action

Example
P1 = (a -> (b -> P1 | c -> STOP) | d -> P4).

“Understanding” the model
You must be able to convert easily between representations (by hand, for easy ones)
- Eg give LTS graph for FSP process
- Eg give FSP process for LTS graph
- Eg give trace set for FSP process
- Eg give FSP process for trace set
- Eg give LTS for trace set
- Eg give trace set for LTS
Also give any representation from a description of a real system

More details of FSP
Some features of the syntax of FSP
- choice
- prefix
- recursion for cycles
- indexing
- Still to come: composition!
Using the LTSA tool
- check syntax
- draw LTS
- produce traces

STOP
STOP is the simplest process
- It represents a system that never does anything
- Eg one in deadlock
P1 = STOP.

Choice
A system with several possible next steps can be represented as a choice
- Each branch has one or more actions and then another process which shows how the system behaves after doing those actions
- Branches separated by vertical bars, enclosed in ()
P1 = (a -> b -> P2 | c -> STOP | d -> P4).
Warnings

- Don’t omit the parentheses around the choice
  - \( P_1 = a \rightarrow b \rightarrow \text{STOP} \mid c \rightarrow P_2 \) is wrong
- Each branch must have some actions before one expression
  - \( P_1 = (P_2 \rightarrow a \mid b \rightarrow P_3) \) is wrong
  - \( P_1 = (a \rightarrow P_2 \rightarrow b \mid c \rightarrow P_3) \) is wrong
  - \( P_1 = (a \rightarrow b \mid c \rightarrow P_2) \) is wrong

Special case: action prefix

- A choice can have a single branch, still need parentheses around it!
- Represents a system whose first action is fixed
  - \( P_1 = (a \rightarrow b \rightarrow \text{STOP}) \)

Recursion

- When a process name occurs after an action, on the right-hand side of an equation, it represents a transition to the start state of the corresponding LTS
- If the process is the same as on the left side, this means a cycle in the LTS
  - \( P_1 = (\text{tick} \rightarrow \text{tock} \rightarrow P_1) \)

Multiprocess Recursion

- Each process name always represents the same state, in a single set of equations
  - \( P_1 = (a \rightarrow P_2 \mid b \rightarrow \text{STOP}) \)
  - \( P_2 = (c \rightarrow P_1 \mid d \rightarrow P_2) \)

Non-deterministic choice

- When several branches start with the same action, then the system can follow either branch
  - no way to know which will be taken in a particular execution
  - This can model randomness or unpredictable failures

Expressiveness of FSP

- For any LTS, you can write expressions in FSP to define it
  - several different expressions can define the same LTS
- Give a name to each node, and define it by a choice, showing all transitions out of that node
  - \( P_1 = (a \rightarrow P_2 \mid b \rightarrow P_3, ...) \)
Expressiveness of FSP

Experience shows that while simple FSP can say anything, it becomes very tedious to have a separate name for each possible action and each possible state.

- eg deposit1dollar, deposit2dollars, deposit10cents, etc
- eg VMWITH1DOLLARSTORED, VMWITH10CENTSSTORED, etc
- Variables that remember state are really useful!

Indexed Actions

A family of related actions can be named as action[v] where v is a variable with a predefined range which is finite.

- range given in the action expression eg pushnumber[b: 0..9]
- or range given before eg … b: 0..9 … pushbutton[b]
- useful trick: define range in one place, so it can be changed easily: range R = 0..9 pushbutton[b:R]

Indexed processes

Similarly, define a family of processes whose names are P[v:R]

- on the righthand side, use v also in other processes, recursive mentions of P, and/or in indexed actions
- P = P[0],
- P[v:0..2] = (count[v] -> P[v+1] | restart -> P[0]),
- P[3] = STOP.

Guarded actions

A branch of a choice can be guarded by an expression involving the variables already defined.

This means: the process will not take that branch unless the expression is true.

- this is used to disable activity
- especially for dealing with boundary of the range while keeping uniform text
- eg P = P[0],
- P[v:0..3] = (when (v<3) count[v] -> P[v+1]).

LTSA features

- Standard file menu
  - stores FSP process definitions in a file
  - open a previously stored file
- Build menu
  - compile: check syntax then convert to LTS graph
- Run menu: animate (produce a trace)
- Check properties

Exercise

Draw LTS, and give three traces, for

P1 = ( a -> b -> P2 | c -> STOP)
| c -> P1),
P2 = (d -> P1).
Exercise

Give FSP, and three traces, for

\[
P_1 = \left(\begin{array}{ccc}
a & b & c \\
b & & c \\
c & & a
\end{array}\right)
\]

Key ideas of composition

- Build a process by combining simpler pieces
  - syntax (FSP text)
  - semantics (LTS representation)
  - leads to set of traces

Issues with alphabets

---

Syntax

- A composite process can be defined from several previously defined processes, separated by double-pipes ("||") and enclosed in parentheses
- The equation for the composition has a double-pipe before the name of the composite process
- Eg \( || \text{COMPOSITE} = (P_1 || P_2 || P_3) \).

Warning

- You can't use a composite process within choice
  - \( P = (a \rightarrow (Q||R)) \), is wrong
  - instead, you must define simple processes, then compose them
  - however, you can use composite process within a composition
  - \( ||P = (P_1 || P_2) \),
  - \( ||Q = (P || P_3) \).

LTS Semantics

- To understand how a composite process behaves, we need to form its LTS (what nodes, what transitions)
  - this is based on the LTS for each component process
- The rules depend on knowing the alphabet of each component
  - that is, what actions are possible for the component

States of the composition

- A state of the composite process is formed by having each component in one of the states for that component
  - The composition state space is potentially a Cartesian product of separate process state spaces
  - not all combinations may actually occur
Example

Non-shared actions

If one component can take a step by an action which is not shared with other components (i.e., not in their alphabet):

- then composite can take this action too
- state of that component changes appropriately
- state of other components does not change

Example

Non-shared actions

Shared actions

If all components that share an action can take it:

- then composite can take this action too
- state of all components that share the action change simultaneously
- state of other components (if any) does not change

Example

Shared actions

Shared actions block

If there is a shared action, but some component can’t take it:

- then composite can’t take it either
- this might block progress in other components which could take this step if they were by themselves
- e.g., there is no c step from state P1, so there is no c step in composite from state <P1,P2>
- even though c takes P2 to Z when P2 is isolated
Interleaving traces

- If we take a trace of the composite process, and ignore all actions except those in alphabet of one component, then what is left is a trace of that component.
- If all processes have disjoint alphabets (i.e., no shared actions at all), then every trace of composite is just an interleaving of traces of the components.

Example

- Start of trace of \((P_1 \parallel P_2)\):
  - \(P_1\): \(X\)
  - \(P_2\): \(Y\)
  - \(P_1\): \(Z\)
  - \(P_2\): \(Z\)

Example

- \(P_A = (u \rightarrow v \rightarrow P_A)\).
  - Trace: \(u \rightarrow v \rightarrow u \rightarrow v \rightarrow u \rightarrow v \ldots\)
- \(P_B = (w \rightarrow P_B | x \rightarrow \text{STOP})\).
  - Traces: \(w \rightarrow w \rightarrow w \rightarrow \ldots w \rightarrow x\)
- \(\parallel P_C = (P_A \parallel P_B)\).
  - Some traces of \(P_C\):
    - \(u \rightarrow w \rightarrow v \rightarrow u \rightarrow v \rightarrow w \rightarrow u \rightarrow v \rightarrow v \ldots\)
    - \(w \rightarrow u \rightarrow v \rightarrow w \rightarrow u \rightarrow w \rightarrow x \rightarrow v \rightarrow v \ldots\)

Exercise

- \(Q_A = (a \rightarrow (b \rightarrow QA | c \rightarrow \text{STOP}) | b \rightarrow \text{STOP})\).
- \(Q_B = (b \rightarrow \text{STOP} | c \rightarrow d \rightarrow \text{STOP})\).
- \(\parallel Q = (Q_A \parallel Q_B)\).
- Is \(a \rightarrow c \rightarrow d\) a trace of \(Q\)?
- Is \(a \rightarrow b \rightarrow a \rightarrow c\) a trace of \(Q\)?

Issues with alphabets

- The semantics of composition depend on the alphabet of the components.
  - Shared actions are treated quite differently than non-shared actions.
- We sometimes need to “tweak” process definitions to get the composition we intend.
- We also want to build systems with multiple components described by a common definition.

Summary I

- Extension:
  - Puts an action in the alphabet even though not mentioned in the process text.
- Relabeling:
  - Changes one action name to another.
- Hiding:
  - Removes an action from the alphabet.
  - Gives corresponding transitions a new private name.
Summary II

• Process Labeling
  - puts extra identifier in front of each action
  - allows multiple components from same definition

• Parameterised processes
  - allow easy change of the scale of the system

Process Alphabet

• Normally the alphabet of a process is all the action names that are mentioned in the definition
  - this includes those mentioned in “local processes” (i.e., descriptions of states within the process; separated by commas in FSP text)
  - expand indexed actions/processes to see what action names occur in the text
  - not all these actions need be present in traces (if some states are unreachable)
  - The alphabet of a composite is the union of the alphabets of the components

Alphabet example

P1 = (a[i:0..2] -> P[i] | b -> Q),
P[i:0..2] = (when (i<2) c[i+1] -> P[i+1]),
Q = (d -> P[0] | d -> STOP).

Alphabet is {a[0], a[1], a[2], b, c[1], c[2], d}
- note: c[0] is not in alphabet

Extension

To make a process with more actions in alphabet than are explicit in the definitions
- why? to allow the process a veto on these actions in a composition

Syntax: P + {x, y}
- process with same LTS as P, but alphabet also has x and y

Extension example

VAR = VAR[0],
VAR[v:0..4] = (read[v] -> VAR[v] [write[w:0..4] -> VAR[w]).
||SYS = (INC || VAR).

If we omit the extension, then SYS can have write[0] step at any time
- This would not be a good model of reality of a thread which repeatedly increments a variable
Relabeling

- Replace one action name by another, everywhere within a process
  - why? usually, to take a predefined process and bring action names to match those needed for composition
- Syntax: \( P \{\text{newname/oldname}\} \)
  - multiple renamings can be done together
  - even give multiple new names for one old
  - note: in composition, relabeling is done first

Relabeling example

\[
P = (a \to b \to P) \{c/b, d/b\}.
\]

- equivalent to
  \[
P = (a \to (c \to P | d \to P)).
\]

Hiding

- Change name on all transitions of a given action, to something that is not shown in traces
  - new name written as "tau" but actually is a private name not used in any other process
  - thus can't be shared in composition
  - why? to ignore internal interactions between components
- Syntax: \( P \{a,b\} \)
  - All a and b transitions are changed to tau
  - Alternative syntax: \( P \{c,d\} \)
  - All transitions except c and d are changed to tau

Multiple similar components

- When we model a system, we often want to describe a type of component, and then build the system by composing several components of that type (perhaps with other types too)
  - eg network has several PCs and one switch and one LAN and one printer
  - eg bank has several tellers, several accounts, several customers

Motivation

- We need someway to produce several different components on the same pattern
- NB what's wrong with saying
  \[
  ||SYS = (PC || PC || LAN || SWITCH ...)
  \]
  - answer: PCs would run in lock step; all transitions in two PCs would occur simultaneously due to shared action names!

Process labeling

- Make a component which is like \( P \) but change the action names in a consistent way
- Syntax: \( a:P \)
  - a component which behaves just like \( P \) except that all actions have names with added "a." in front
  - eg action "read" in \( P \) becomes "a.read" in \( a:P \)
  - So ||SYS = (pc1:PC || pc2:PC || LAN ...)
  - warning: labeling can only be done within composition (not in simple process)
Component families

- Use indexed labels for a family of similar components
- \(|\text{SYS} = (\text{pc}[0]:\text{PC} \ | \ \text{pc}[1]:\text{PC} \ | \ \text{pc}[2]:\text{PC} \ | \ ...)|
- Equivalent is
- \(\text{SYS} = (\forall i:0..3 \ \text{pc}[i]:\text{PC} \ | \ ...)|

Parameterised processes

- Sometimes a system consists of an unknown number of similar components
- So we define a system with a parameter determining the number of components
- LTSA insists that we give the parameter an exact value before we use it
- Eg \(\text{SYS}(N=3) = (\forall i:0..N \ \text{pc}[i]:\text{PC} \ | \ ...)|

Families and relabeling

- Sometimes, some actions should be shared between members of a family, in order to be performed simultaneously in all components
  - Use process labels which makes separate action names
  - Then relabel actions that should be shared to something common

Example

- THREAD = (doit -> THREAD | cancel -> STOP).
- Suppose we want each thread to have its separate doit, but all should take cancel step together
- \(\text{SYS} = (\forall i:R \ \text{thread}[i]:\text{THREAD}) / \{ \forall i:R \ \text{cancelall}/\text{thread}[i].\text{cancel} \} \).