Outline

• Introduction
• Modeling
• Specifying properties and Verification
• An example
• Project assignment
• References, links
Labeled Transition System Analyzer (LTSA)

- Animate and check the behavior of the overall system before it is implemented
  - focus on an aspect of interest - concurrency
  - model animation to visualise a behaviour
  - mechanical verification of properties (safety & progress)
- by Jeff Magee and Jeff Kramer
The Modeling Approach

- Equivalent graphical and textual representations
  - State machines
    - $LTS$ - Labeled Transition Systems
  - Process algebra
    - $FSP$ - Finite State Processes
FSP – action prefix and recursion

\[ \text{SWITCH} = \text{OFF}, \]
\[ \text{OFF} = (\text{on} \rightarrow \text{ON}), \]
\[ \text{ON} = (\text{off} \rightarrow \text{OFF}). \]

Substituting to get a more succinct definition:

\[ \text{SWITCH} = \text{OFF}, \]
\[ \text{OFF} = (\text{on} \rightarrow (\text{off} \rightarrow \text{OFF})). \]

And again:

\[ \text{SWITCH} = (\text{on} \rightarrow \text{off} \rightarrow \text{SWITCH}). \]
SWITCH = OFF,
OFF    = (on -> ON),
ON     = (off-> OFF).

If \( x \) is an action and \( P \) a process then \((x -> P)\)
describes a process that initially engages in the action \( x \) and then behaves exactly as described by \( P \).
DRINKS = (red->coffee->DRINKS | blue->tea->DRINKS).
FSP – choice

If $x$ and $y$ are actions then $(x \rightarrow P \mid y \rightarrow Q)$ describes a process that initially engages in either of the actions $x$ or $y$. After the first action has occurred, the subsequent behaviour is described by $P$ if the first action was $x$ and $Q$ if the first action was $y$.

Who or what makes the choice?

Is there a difference between input and output actions?
DOuble-sided flowchart:  

**FSP – nondeterministic choice**

\[
\text{COIN} = (\text{toss}\rightarrow\text{HEADS}|\text{toss}\rightarrow\text{TAILS}),  \\
\text{HEADS} = (\text{heads}\rightarrow\text{COIN}),  \\
\text{TAILS} = (\text{tails}\rightarrow\text{COIN}).
\]

Who makes the choice?

Process \((x\rightarrow P \mid x \rightarrow Q)\) describes a process which engages in \(x\) and then behaves as either \(P\) or \(Q\).
FSP – indexed processes & actions

Single slot buffer that inputs a value in the range 0 to 3 and then outputs that value:

\[
BUFF = (in[i:0..3]->out[i]->BUFF).
\]

equivalent to

\[
BUFF = (in[0]->out[0]->BUFF
|in[1]->out[1]->BUFF
|in[2]->out[2]->BUFF
|in[3]->out[3]->BUFF
).
\]

or using a process parameter with default value:

\[
BUFF(N=3) = (in[i:0..N]->out[i]->BUFF).
\]
Using index expressions to model calculation:

const N = 1
range T = 0..N
range R = 0..2*N

SUM        = (in[a:T][b:T]->TOTAL[a+b]),
TOTAL[s:R] = (out[s]->SUM)
The choice *(when B \times \rightarrow P \mid y \rightarrow Q)* means that when the guard B is true then the actions x and y are both eligible to be chosen, otherwise if B is false then the action x cannot be chosen.
FSP – guarded actions

What is the following FSP process equivalent to?

\[
\text{const False} = 0 \\
P = (\text{when (False) doanything} \rightarrow P).
\]

Answer:

STOP
Shangzhu Weng

FSP – parallel composition

Modeling concurrency:

ITCH = (scratch->STOP).
CONVERSE = (think->talk->STOP).

||CONVERSEITCH = (ITCH || CONVERSE).

Commutative: \((P || Q) = (Q || P)\)
Associative: \((P || (Q || R)) = ((P || Q) || R)\)
\(= (P || Q || R)\).

Shangzhu Weng
FSP - action interleavings

ITCH
0
2 states

CONVERSE
0
think
1
talk

CONVERSE_ITCH
0
1
2
3
4
5

(0,0) (0,1) (0,2) (1,2) (1,1) (1,0)

2 x 3 states

from ITCH

from CONVERSE

2 x 3 states
**FSP - shared actions**

Modeling process interactions:

\[
\text{MAKER} = (\text{make} \rightarrow \text{ready} \rightarrow \text{MAKER}). \\
\text{USER} = (\text{ready} \rightarrow \text{use} \rightarrow \text{USER}).
\]

\[
\lvert \lvert \text{MAKER}_\text{USER} = (\text{MAKER} \lvert \lvert \text{USER}).
\]

A composite process

While unshared actions may be arbitrarily interleaved, a shared action must be executed at the same time by all processes that participate in the shared action.
Two instances of a switch process:

\[
\text{SWITCH} = (\text{on} \rightarrow \text{off} \rightarrow \text{SWITCH}).
\]

\[
||\text{TWO\_SWITCH} = (a:\text{SWITCH} || b:\text{SWITCH}).
\]

\[
a:P \text{ prefixes each action label in the alphabet of } P \text{ with } a.
\]
Two instances of a switch process:

\[
\text{SWITCH} = \text{(on} \rightarrow \text{off} \rightarrow \text{SWITCH)}.
\]

\[
||\text{TWO\_SWITCH} = (a:\text{SWITCH} || b:\text{SWITCH}).
\]

An array of instances of the switch process:

\[
||\text{SWITCHES (N=3)} = (\text{forall}[i:1..N] \ s[i]:\text{SWITCH}).
\]

\[
||\text{SWITCHES (N=3)} = (s[i:1..N]:\text{SWITCH}).
\]
FSP – process labeling

Processes may also be labelled by a set of prefix labels

Process prefixing is useful for modeling shared resources:

\[
\text{RESOURCE} = (\text{acquire} \rightarrow \text{release} \rightarrow \text{RESOURCE}).
\]

\[
\text{USER} = (\text{acquire} \rightarrow \text{use} \rightarrow \text{release} \rightarrow \text{USER}).
\]

\[
\text{RESOURCE\_SHARE} = (a:USER \mid\mid b:USER \mid\mid \{a,b\}::\text{RESOURCE}).
\]
RESOURCE = (acquire->release->RESOURCE).
USER = (acquire->use->release->USER).
||RESOURCE_SHARE = (a:USER || b:USER
|| {a,b}::RESOURCE).
FSP – action relabeling

Relabeling to ensure that composed processes synchronize on particular actions.

CLIENT = (call->wait->continue->CLIENT).
SERVER = (request->service->reply->SERVER).

CLIENT_SERVER = (CLIENT || SERVER)

/ {call/request, reply/wait}.
FSP – action hiding

Abstraction to reduce complexity:

When applied to a process P, the hiding operator \{a1..ax\} removes the action names a1..ax from the alphabet of P and makes these concealed actions "silent". These silent actions are labeled tau. Silent actions in different processes are not shared.

Sometimes it is more convenient to specify the set of labels to be exposed...

When applied to a process P, the interface operator @{a1..ax} hides all actions in the alphabet of P not labeled in the set a1..ax.
FSP – action hiding

The following definitions are equivalent:

USER = (acquire->use->release->USER)
\{use\}.
USER = (acquire->use->release->USER)
@\{acquire,release\}.

Minimization removes hidden tau actions to produce an LTS with equivalent observable behavior.
Deadlock analysis

- deadlocked state is one with no outgoing transitions
- in FSP: STOP process

\[ \text{MOVE} = (\text{north} \rightarrow (\text{south} \rightarrow \text{MOVE} | \text{north} \rightarrow \text{STOP})) \]

Analysis using \textit{LTSA}: (shortest trace to STOP)

Trace to \textbf{DEADLOCK}: north north

Shangzhu Weng
Deadlock analysis –
the Dining Philosopher example

Deadlock may arise from the parallel composition of interacting processes.
Deadlock analysis –

the Dining Philosophy example

FORK = (get -> put -> FORK).
PHIL = (sitdown -> right.get -> left.get
    -> eat -> right.put -> left.put
    -> arise -> PHIL).

Table of philosophers:

||DINERS(N=5)= forall [i:0..N-1]
    (phil[i]:PHIL ||
    {phil[i].left,phil[((i-1)+N)%N].right}::FORK).

Shangzhu Weng
Deadlock analysis -
the Dining Philosophy example

Trace to DEADLOCK:
phil.0.sitdown
phil.0.right.get
phil.1.sitdown
phil.1.right.get
phil.2.sitdown
phil.2.right.get
phil.3.sitdown
phil.3.right.get
phil.4.sitdown
phil.4.right.get

This system deadlocks!!

Shangzhu Weng
Deadlock analysis -
the Dining Philosophy example

Introduce an asymmetry into our definition of philosophers.

Use the identity I of a philosopher to make even numbered philosophers get their left forks first, odd their right first.

*Other strategies?*

$$\text{PHIL}(I=0) = (\text{when } (I \% 2 == 0) \text{ sitdown}$$

$$\rightarrow \text{left.get} \rightarrow \text{right.get}$$

$$\rightarrow \text{eat}$$

$$\rightarrow \text{left.put} \rightarrow \text{right.put}$$

$$\rightarrow \text{arise} \rightarrow \text{PHIL}$$

$$\text{|when } (I \% 2 == 1) \text{ sitdown}$$

$$\rightarrow \text{right.get} \rightarrow \text{left.get}$$

$$\rightarrow \text{eat}$$

$$\rightarrow \text{left.put} \rightarrow \text{right.put}$$

$$\rightarrow \text{arise} \rightarrow \text{PHIL}$$

$$.}
Safety properties

- Nothing bad happens
- In the model: No reachable \textit{ERROR/STOP} state

\begin{verbatim}
ACTUATOR
  =(command->ACTION | respond->ERROR),
ACTION
  =(respond->ACTUATOR | command->ERROR).
\end{verbatim}
Safety properties

- Nothing bad happens
- In the model: No reachable ERROR/STOP state

```
property SAFE_ACTUATOR = (command -> respond -> SAFE_ACTUATOR).
```

In complex systems, it is usually better to specify safety properties by stating directly what is required.
Safety **property** \( P \) defines a deterministic process that asserts that any trace including actions in the alphabet of \( P \), is accepted by \( P \).

A safety property must be specified so as to include all the acceptable, valid behaviors in its alphabet.

\[
\text{property } \text{CALM} = \text{STOP} + \{\text{disaster}\}.
\]

Shangzhu Weng
const Max = 3
range Int = 0..Max

SEMAPHORE (N=0) = SEMA[N],
SEMA[v:Int] = (up->SEMA[v+1]
 | when(v>0) down->SEMA[v-1]
 ),
SEMA[Max+1] = ERROR.
Safety properties –

a semaphore example

\[
\text{LOOP} = (\text{mutex}.\text{down} \rightarrow \text{enter} \rightarrow \text{exit} \\
\rightarrow \text{mutex}.\text{up} \rightarrow \text{LOOP}).
\]

\[
\text{||SEMADEMO} = (p[1..3]:\text{LOOP} \\
||\{p[1..3]\}::\text{mutex}:\text{SEMAPHORE}(1)).
\]

How do we check that this does indeed ensure mutual exclusion in the critical section?

\[
\text{property MUTEX} = (p[i:1..3].\text{enter} \\
\rightarrow p[i].\text{exit} \\
\rightarrow \text{MUTEX} ).
\]

\[
\text{||CHECK} = (\text{SEMADEMO} || \text{MUTEX}).
\]
Progress properties

Liveness

Something good eventually happens

A progress property in LTSA asserts that it is *always* the case that an action is *eventually* executed. (handles a restricted class of liveness)

Fair Choice: If a choice over a set of transitions is executed infinitely often, then every transition in the set will be executed infinitely often.
Progress properties - toss-a-coin example

COIN = (toss->heads->COIN | toss->tails->COIN).
Progress properties - toss-a-coin example

COIN = (toss->heads->COIN | toss->tails->COIN).

progress HEADS = {heads}
progress TAILS = {tails}

No progress violations detected.
Suppose that there were two possible coins that could be picked up: a trick coin and a regular coin.

\[
\text{TWOICOIN} = (\text{pick} \rightarrow \text{COIN} | \text{pick} \rightarrow \text{TRICK}),
\]
\[
\text{TRICK} = (\text{toss} \rightarrow \text{heads} \rightarrow \text{TRICK}),
\]
\[
\text{COIN} = (\text{toss} \rightarrow \text{heads} \rightarrow \text{COIN} | \text{toss} \rightarrow \text{tails} \rightarrow \text{COIN}).
\]
Suppose that there were two possible coins that could be picked up: a trick coin and a regular coin.

**LTSA** check progress

- **progress HEADS** = \{heads\}
- **progress TAILS** = \{tails\}

**Progress violation**: TAILS

Path to terminal set of states:
- pick

Actions in terminal set:
- \{toss, heads\}

Shangzhu Weng
A terminal set of states is one in which every state is reachable from every other state in the set via one or more transitions, and there is no transition from within the set to any state outside the set.

Terminal sets for TWOCOIN:
{1,2} and {3,4,5}

progress TAILS = \{tails\} is violated.
A progress property is violated if analysis finds a terminal set of states in which none of the progress set actions appear.

Default: given fair choice, for every action in the alphabet of the target system, that action will be executed infinitely often. This is equivalent to specifying a separate progress property for every action.
Progress analysis–toss-a-coin example

Default analysis for TWOCOIN: separate progress property for every action.

Progress violation for actions: \{pick\}
Path to terminal set of states: pick
Actions in terminal set: \{toss, heads, tails\}

Progress violation for actions: \{pick, tails\}
Path to terminal set of states: pick
Actions in terminal set: \{toss, heads\}

If the default holds, then every other progress property holds
Progress – action priorities

Action priority expressions describe scheduling properties, specified with respect to process compositions.

\[ ||C = (P||Q) << \{a_1, \ldots, a_n\} \] specifies a composition in which the actions \(a_1, \ldots, a_n\) have higher priority than any other action in the alphabet of \(P||Q\) including the silent action \(\tau\).

\[ ||C = (P||Q) >> \{a_1, \ldots, a_n\} \] specifies a composition in which the actions \(a_1, \ldots, a_n\) have lower priority than any other action in the alphabet of \(P||Q\) including the silent action \(\tau\).
Progress - action priorities

NORMAL = (work->play->NORMAL | sleep->play->NORMAL).

||HIGH = (NORMAL) << {work}.

||LOW = (NORMAL) >> {work}.
The Gas Station Example
$N$ customers obtain gas by prepaying the cashier at the gas station.

The cashier activates one of $M$ pumps to serve the customer.

The appropriate amount of gas is then delivered to the appropriate customer by a deliver.
The Gas Station Example - the model

const N = 3  // number of customers
const M = 2  // number of pumps
range C = 1..N // customer range
range P = 1..M // pump range
range A = 1..2 // amount of money or gas

CUSTOMER = (prepay[a:A]->gas[x:A]->
    if (x==a) then CUSTOMER else ERROR).

CASHIER =
    (customer[c:C].prepay[x:A]->start[P][c][x]->CASHIER).

PUMP = (start[c:C][x:A]->gas[c][x]->PUMP).

DELIVER =
    (gas[P][c:C][x:A]->customer[C].gas[x]->DELIVER).

||STATION = (CASHIER || pump[1..M]:PUMP || DELIVER)
    /{pump[i:1..M].start/start[i],
        pump[i:1..M].gas/gas[i]}@{customer}.

||GASSTATION = (customer[1..N]:CUSTOMER || STATION).
The Gas Station Example - reachability analysis

Does a customer *always* get the correct amount of gas?

Not *always* !!

Reachability analysis
Performs an exhaustive search of the state space to detect ERROR and deadlock states

```
property customer.3: CUSTOMER violation.
property customer.2: CUSTOMER violation.
property customer.1: CUSTOMER violation....
States Composed: 3409 Transitions: 11862 in 1468ms
Trace to property violation in customer.2: CUSTOMER:
customer.1.prepay.1 →
pump.1.start.1.1
customer.2.prepay.2 →
pump.1.gas.1.1
customer.2.gas.1 →
```
The Gas Station Example -
corrected model

const N = 3  // number of customers
const M = 2  // number of pumps
range C = 1..N // customer range
range P = 1..M // pump range
range A = 1..2  // amount of money or gas

CUSTOMER = (prepay[a:A]->gas[x:A]->
    if (x==a) then CUSTOMER else ERROR).

CASHIER =
    (customer[c:C].prepay[x:A]->start[P][c][x]->CASHIER).
PUMP = (start[c:C][x:A]->gas[c][x]->PUMP).

DELIVER =
    (gas[P][c:C][x:A]->customer[c].gas[x]->DELIVER).

||STATION = (CASHIER || pump[1..M]:PUMP || DELIVER)
    /{pump[i:1..M].start/start[i],
        pump[i:1..M].gas/gas[i]}@{customer}.

||GASSTATION = (customer[1..N]:CUSTOMER || STATION).
The Gas Station Example
- check safety property

Safety property:
If a customer pays first, it should get gas first. (FIFO)

\[
\begin{align*}
\text{range } T &= 1..2 \\
\text{property} \quad \\
\text{FIFO} &= (\text{customer}[i:T].\text{prepay}[A] \rightarrow \text{PAID}[i]), \\
\text{PAID}[i:T] &= (\text{customer}[i].\text{gas}[A] \rightarrow \text{FIFO} \\
&\quad | \text{customer}[j:T].\text{prepay}[A] \rightarrow \text{PAID}[i][j]) \\
\text{PAID}[i:T][j:T] &= (\text{customer}[i].\text{gas}[A] \rightarrow \text{PAID}[j]).
\end{align*}
\]

Does this system satisfy the property?
No, if we have more than one pumps!!
The Gas Station Example
- check safety property

A counterexample trace:

Composing
  property FIFO violation.
States Composed: 617 Transitions: 1398 in 94ms
Trace to property violation in FIFO:
  customer.1.prepay.1
  pump.1.start.1.1
  customer.2.prepay.1
  pump.2.start.2.1
  pump.2.gas.2.1
  customer.2.gas.1

Although a pump is activated for customer1 first,
the gas is given to customer 2 first!
The Gas Station Example - check liveness

Liveness property:
Customers will eventually get served.

Holds!

Try this?

\[ ||\text{GASSTATION} = (\text{customer[1..N]}:\text{CUSTOMER} || \text{STATION}) \]
\[ \gg\{\text{customer[1]})\}. \]

Customer 1 will never be served!
References and links

- *Concurrency: state models & Java programs*
  [Jeff Magee and Jeff Kramer] Wiley, 1999
  http://www-dse.doc.ic.ac.uk/concurrency/