EAST WEST UNIVERSITY

Department of Computer Science and Engineering

CSE430 - Software Testing and Quality Assurance

Lab Topic: LTSA - Labelled Transition System Analyser

1. Introduction

In software engineering, understanding the behavior of concurrent systems before implementation is essential for preventing design errors such as deadlocks, livelocks, or synchronization issues.

The Labelled Transition System Analyser (LTSA) is a verification tool used to model, analyze, and validate system behavior using Finite State Processes (FSP).

LTSA allows developers and students to specify a system in a formal notation and visualize its behavior through state transition diagrams (LTS). It supports animation for exploring possible sequences of actions, and model checking for verifying safety and liveness properties.

LTSA Download Link: https://www.doc.ic.ac.uk/ltsa/#Downloads

2. Objectives

After completing this lab session, students will be able to:

- Understand the theoretical foundation of Labelled Transition Systems and FSP.
- Model simple and concurrent systems using LTSA.
- Use LTSA to detect deadlocks, synchronization errors, and property violations.

3. Overview of LTSA

3.1 Definition

Labelled Transition System (LTS) is a mathematical model representing system states and the actions that cause transitions between those states.

Each node in the LTS corresponds to a state, and each edge corresponds to an action (event) that moves the system from one state to another.

3.2 The Finite State Process (FSP)

FSP is a formal language used by LTSA to describe behaviors of systems. An FSP specification defines:

- Actions (events that occur)
- Processes (behavioral entities)
- Composition (how multiple processes interact)

Once written, the FSP model is compiled by LTSA into its corresponding LTS diagram.

4. Importance of LTSA

The LTSA tool plays an important role in the design and verification of complex systems:

- It helps identify deadlocks and race conditions early in the design phase.
- It enables students to visualize concurrency and understand interleaving of events.
- It assists in verifying safety properties (undesired states never occur) and progress properties (desired states eventually occur).
- It bridges the gap between theoretical modeling and real-world implementation.

5. Working with LTSA

Step-by-step Procedure

1. **Open LTSA Tool**
   Launch the LTSA editor and create a new FSP specification file.

2. **Write the FSP code**
   Enter the process definitions in the text editor window.

3. **Compile the Specification**
   Click the “Compile” button to generate the LTS model.

4. **Draw the LTS Diagram**
   From the compiled processes list, select the desired process and click “Draw”.

5. **Animate the System**
   Use the “Animate” option to simulate and observe system execution traces.

6. **Verify Properties**
   Use the **Check** option to verify safety and progress properties.

6. Writing FSP Programs

Below are common types of FSP constructs along with examples and explanations.

6.1 Sequential Process (Action Prefix and Recursion)

```
SWITCH = (on -> off -> SWITCH).
```

- This model defines a simple toggle switch that turns on, then off, and repeats indefinitely.
- The arrow (->) represents an action followed by the next state. The process name SWITCH at the end indicates recursion, meaning it restarts from the beginning after completing the sequence.

6.2 Deterministic Choice

```
DRINKS = (red -> coffee -> DRINKS
       | blue -> tea    -> DRINKS).
```

- Here, the process allows a deterministic choice between two separate starting actions — either red or blue.
- Once an action is selected, it determines the following sequence (coffee or tea).
- This is useful for modeling user choices in a system.

6.3 Non-Deterministic Choice

```
COIN  = (toss -> HEADS | toss -> TAILS),
HEADS = (heads -> COIN),
TAILS = (tails -> COIN).
```

- The same action (toss) may lead to different outcomes (HEADS or TAILS).
- Non-deterministic choice models uncertain or random behaviors that depend on external factors.

6.4 Indexed Actions

```
BUFFER(N=3) = (in[i:0..N] -> out[i] -> BUFFER).
```

- Indexed actions allow repetition for multiple instances of similar operations.
- This example defines input-output pairs (in[i] and out[i]) for three buffer positions.
- It helps model data storage or message-passing systems.

6.5 Guarded Actions (Conditional Execution)

```
COUNT(N=3)    = COUNT[0],
COUNT[i:0..N] = ( when(i<N) inc -> COUNT[i+1]
              | when(i>0) dec -> COUNT[i-1] ).
```

- Guards restrict actions based on conditions.

- Here, inc is enabled only when i < N, and dec is enabled only when i > 0.
- This ensures the counter never goes below 0 or above the maximum limit.

6.6 Parallel Composition (Synchronization)

```
MAKER = (make -> ready -> MAKER).
USER  = (ready -> use  -> USER).
||SYSTEM = (MAKER || USER).
```

- Two processes (MAKER and USER) execute in parallel.
- The shared action ready forces synchronization — both must reach it simultaneously.
- LTSA will show possible interleavings before and after synchronization.

6.7 Relabelling (Renaming for Interaction)

```
CLIENT = (call -> wait -> continue -> CLIENT).
SERVER = (request -> service -> reply -> SERVER).
||CS = (CLIENT || SERVER) / { call/request, reply/wait }.
```

- Relabelling aligns actions of different processes for communication.
- Here, CLIENT.call is relabelled to match SERVER.request, and SERVER.reply matches CLIENT.wait.
- As a result, both processes synchronize correctly during simulation.

6.8 Shared Resource Modeling

```
RESOURCE = (acquire -> release -> RESOURCE).
USER     = (printer.acquire -> use -> printer.release -> USER) \ {use}.
||SHARED = (a:USER || b:USER || {a,b}::printer:RESOURCE).
```

- Two users (a and b) share a single printer resource.
- The shared printer can only be acquired by one user at a time, ensuring mutual exclusion.
- Hiding the internal use action (\ {use}) keeps the model abstract.

6.9 Deadlock scenario

```
R1 = (r1.acquire -> r1.release -> R1).
R2 = (r2.acquire -> r2.release -> R2).

P = (r1.acquire -> r2.acquire -> critical -> r2.release -> r1.release -> P).
Q = (r2.acquire -> r1.acquire -> critical -> r1.release -> r2.release -> Q).

||DEADLOCK_SYS = (P || Q || R1 || R2).
```

Model

- This model demonstrates a deadlock situation in a concurrent system where two processes compete for two shared resources (R1 and R2).
- The goal is to visualize how resource allocation order can create a circular wait and cause the system to become permanently blocked.
- Each resource can be acquired and released repeatedly.
- After being released, it becomes available again.
- The labels r1.acquire and r1.release (similarly for r2) represent the synchronization points where processes interact with the shared resources.

Process P

- Process P first acquires Resource R1, then tries to acquire Resource R2.
- After obtaining both locks, it enters a critical section (labeled critical).
- Upon completion, it releases both resources in the reverse order and repeats the cycle.
- This represents a thread that locks resources sequentially in one specific order.

Process Q

- Process Q follows the opposite order: it first acquires R2, then R1.
- The reversed sequence creates a potential for circular dependency.
- If both processes hold one lock and wait for the other, neither can proceed.
- The || operator composes all four processes to execute concurrently.
- Processes P and Q interact with the same resources R1 and R2.
- Synchronization occurs automatically on shared action names (e.g., r1.acquire).

6.9.1 How the Deadlock Occurs

Step	Action	Explanation
1	r1.acquire (by P)	Process P acquires Resource R1.
2	r2.acquire (by Q)	Process Q acquires Resource R2.
3	-	Now P waits for R2, but Q is holding it.
4	-	Q waits for R1, but P is holding it.
5	Deadlock	Both processes are blocked forever, waiting on each other.

This forms the Circular Wait Condition, one of the four classic conditions of deadlock (Mutual Exclusion, Hold and Wait, No Preemption, Circular Wait).

6.9.2 Key Concepts Illustrated

Concept	Description
Mutual Exclusion	Each resource can only be held by one process at a time.
Hold and Wait	Each process holds one resource while requesting another.

No Preemption	Resources are released only voluntarily.
Circular Wait	Each process waits for a resource held by the other.

All four conditions hold simultaneously, creating a classic deadlock situation.

6.9.3 How to Recover (Conceptually)

To prevent such deadlocks, the system must break one or more of the four conditions.
Two common fixes:

1. Global Ordering:
   Force all processes to acquire resources in the same order (e.g., always `r1` before `r2`).

2. Waiter/Coordinator (Mutex):
   Introduce a central controller that allows only one process to acquire multiple resources at a time.

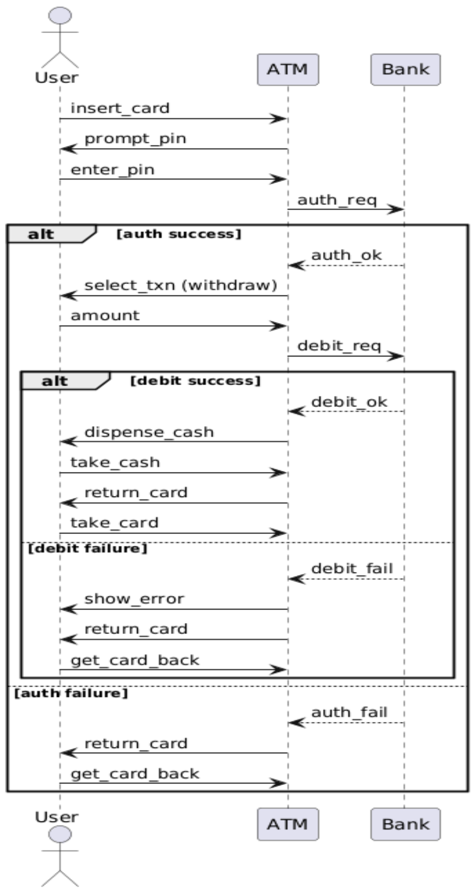
Both solutions remove the circular wait condition, making the system deadlock-free.

7. Case Study: Modeling a User–ATM System in LTSA

7.1 Scenario Description

The ATM system involves three interacting entities: the User, the ATM machine, and the Bank. The process begins when a user inserts a card, enters a PIN, requests a transaction, and ends with either successful cash withdrawal or failure due to authentication or balance issues.

A simplified sequence of interactions is as follows:



1. User inserts card.
2. ATM prompts for PIN.
3. The user enters a PIN.
4. ATM sends authentication request to Bank.
5. The bank replies with success or failure.
6. If successful, the ATM processes transactions and dispenses cash.
7. Finally, the ATM returns the card to the user.

7.2 FSP Model of the User–ATM System

```
USER =
  ( insert_card -> prompt_pin -> enter_pin -> auth_req ->
     ( auth_ok ->
         select_withdraw -> amount ->
           ( dispense_cash -> take_cash -> return_card -> take_card -> USER
           | show_error    -> return_card -> get_card_back -> USER
           )
     | auth_fail ->
         return_card -> get_card_back -> USER
     )
```

```
 ).

ATM =
  ( insert_card -> prompt_pin -> enter_pin -> auth_req ->
     ( auth_ok ->
         select_withdraw -> amount -> debit_req ->
           ( debit_ok  -> dispense_cash -> return_card -> ATM
           | debit_fail-> show_error    -> return_card -> ATM
           )
     | auth_fail -> return_card -> ATM
     )
  ).

BANK =
  ( auth_req  -> (auth_ok   -> BANK
              | auth_fail -> BANK)
  | debit_req -> (debit_ok  -> BANK
              | debit_fail-> BANK)
  ).

||ATM_SYSTEM = ( USER || ATM || BANK ).
```

- The User interacts with the ATM through card insertion, PIN entry, and receiving responses.
- The ATM interacts with both User and Bank for authentication and debit operations.
- The Bank provides responses to authentication and debit requests.
- The Relabelling section (/ { ... }) ensures corresponding actions synchronize between components.
- The resulting model can be animated to display successful and failed transaction paths.

8. Lab Tasks

Students should complete the following tasks in LTSA and submit their results (LTS diagrams, analysis, and observations):

1. Model a toggle switch (SWITCH) that alternates between *on* and *off* states.

2. Develop a simple Traffic Light Controller with states *Red → Green → Yellow → Red.*

3. Extend the Drinks Machine model with a cancel option.

4. Simulate a Coin Toss Machine using non-deterministic choice.

5. Create a Counter model with guard conditions to restrict underflow and overflow.

6. Build a Maker–User synchronization system using shared actions.

7. Implement a Client–Server model using relabelling to synchronize request and reply actions.

8. Construct a Shared Printer model ensuring only one user can use the printer at a time.

9. Analyze the ATM system model for safety: verify that cash is never dispensed without successful authorization.

10. Add a progress property to confirm that the card is always returned after any transaction.