Operating Systems Lab
Dept of CSE, LNMIIT, Jaipur
15 March 2021
Lab Session 05

Introduction

This lab session is about an interesting tool for analysing process behaviours. First, we build an abstract model of the process to present to the analyser. The specific analyser we have chosen for this lab session is called: Labelled Transition System Analyzer (LTSA) and you can find a generous collection of resources for this tool at: https://www.doc.ic.ac.uk/~jnm/book/ We have downloaded some of these resources here into our OS classroom for easy access.

We do not expect students to be expert LTSA model builders. Instead, we use a few example models to help the students be familiar with the model analyser. Students can also use the help tab on LTSA to get supporting information.

Students will also be guided to use the analyser software. We explain the kinds of questions the analyser answers.

Only two simple runs of action sequences are needed as submissions for this lab work. All OS students must complete this lab session by the end of week 15-19 Feb – before the start of the mid-term examinations period.

Task 01: Setting up LTSA for Dining Philosophers Problem

Download LTSA package from OS Classroom site and install it on your computer. This is a Java-based system and can be run on any computer with Java (Download and install Java on your native computer operating system, if necessary). I am working on my Windows system for this exercise.

Unzip ltsatool. A minor change is needed in file ltsa.bat. Correct line: `javaw.exe -Xmx512m -jar D:\JeffDoc\java\Ltsa\ltsa.jar` to match the location of your ltsa.jar file.

You are ready to run the tool. Double click newly edited ltsa.bat icon. The analyser can also be started by a double click on ltsa.jar file. Under LTS Analyse window that pops us do the following:

1. File >> Examples >> Chapter6 >> DiningPhilosophers. The Examples section has all models discussed in a book written by tool developers.
2. Let us explore philosophers' problem a bit before we read the code: On LTSA select tab: Check >> Run >> DEFAULT
3. It pops a big Animator window. The window lists the alphabet of the model. ***Alphabet*** is set of all the actions that the model has. Some entries (action names) have a checked box in front. Others are unchecked. Only actions with checked boxes in front are enabled. Other actions are not ready.
4. It should be easy for you to make sense of the action names in the animator. For example, the animator on your screen is showing `phil.0` can `sitdown`, and `phil.1` can `sitdown` … Click on `phil.2.sitdown` to do that action.

5. A new set of enabled (and, disabled) action names appear on the animator window. Run the system to your hearts content by clicking on enabled action names and let the system progress.
6. History of your completed actions is shown on the left column of the Animator window. ***Deadlock*** is the condition where no choice of an enabled action name is available. Can you steer the animator to a deadlock?
7. If you cannot find actions leading to a deadlock, do not worry. The Analyser can do so for you. Close the Animator window.
8. Select Check >> Safety. The system determines that the model can lead to a deadlock and it also determines the action sequence leading to a deadlock.
9. In fact, you will get similar message through Check >> Progress.
10. We will learn the difference between *Safety* and *Progress* little later. For the present remember, **Safety** means *No bad event should happen*. **Progress** means *Good event(s) should happen*.

Let us explore the analyser a bit more. This time we explore its wonderful ability to draw diagrams!

1. Once again, let us Check >> Safety
2. Notice we are under Tab Output. What is under tab Edit? Well, it shows the input LTS model presented by the programmer.
3. What is under tab Draw?
4. Left Panel has Processes – Subprocess of the overall model. The Right panel is for displaying corresponding Transition diagrams. Middle tool bar shows edit options for altering diagram size – Zoom in and Zoom out.
5. Reveal the Transition diagram for phil.0 (this is done by clicking on left panel items). Left panel tells us it is PHIL (philosopher process). Click once on it. Resize diagram to suit your screen size.
6. You notice that the subprocess (phil.0) repeats a fixed set of actions in an order for ever. Other philosophers also live the same lifecycle forever.
7. Forks have even more boring lifecycle. But their naming convention is more sophisticate. `{phil.0.left, phil.4.right}::FORK`.
8. This fork process is to be understood as follows: each fork process has two aliases (names or references). That is `phil.0.left` is a fork, and this same fork is also called `phil.4.right`. You already know each adjacent philosopher-pair shares a common fork; each fork is on the left of one philosopher and on the right of the other philosopher.
9. You may run the animator window again with draw panel open. Each action is displayed through a red transition as it occurs.
10. Also notice that there is **step** option under Run to animate reported safety problems. Use this animation with draw window to see progress in pictures.

Menu Bar tab Window

1. Select option Transitions under tab Window. Then select Transitions (in the submenu Edit, Output, Transitions)
2. This time you view a text-based description of processes in the model. We take the example, `phil.0:PHIL`

3. It reads: Q0 is a subprocess. Q0 does action `phil.0.sitdown` and then does what subprocess Q1 does. Q1 does action `phil.0.right.get` and the does actions of subprocess Q2. So on.
4. Note `||DINERS` is a concurrent process composed of all the subprocesses. Let us now quickly list some core properties of processes and actions.

What are Process and Actions?

1. NO MORE THAN ONE ACTION can occur in a concurrent process at any instance of time. For example, two different philosophers cannot pick their forks at the same instance of time.

   Some of you may say, how then two philosophers can eat simultaneously? Well, the modelled action "eat" represents "begin eating" event. The procedures of actual eating are not in the abstracted model. Similarly, action "left.put" is the event/action representing the end of the eating routine.

   When we compare two acts of eating, one philosopher will necessarily begin eating before the other. No two philosophers will begin eating at a single instance.

   A consequence of this single event rule is this: *all actions are ordered in time.*

2. If we have a process composed of a set of concurrent processes and the subprocesses share an action/event name. Then the named action occurs in all subprocesses that share the action name simultaneously – this is so because this is one single event involving (perhaps) multiple agents.

   For example, `phil.1.right.get` is an event in a FORK subprocess as well as in a PHIL subprocess. The action/event/transition bearing this name on both subprocesses occurs as a single common action. After the transition `phil.1` has a fork in `right` hand. The fork on the `right` of the philosopher is in its picked-up state. It is no more available to the other philosophers.

Task 02: Deadlock Free Philosophers

This is a good time for the students to look at another example in examples from Chapter 6. This example is named *DeadlockFreePhilosophers*. As the name suggests, it models a deadlock free system.

In what way this process is different from the process in Task 01? Look at the details carefully. Carefully review the examples against the four conditions for deadlocks discussed in the class: *Mutual exclusion, Hold and wait, No pre-emption, and Circular wait.*

Task 03: Understanding LTSA Modelling Language

Let us begin with the dining philosopher example. It defines two core process lifecycles: FORK and PHIL.

```
FORK = (get -> put -> FORK).
```

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

See slides for Chapter 2 if you need more detailed discussion. Processes (lifecycles) are named using upper case letters. Actions, events, and transitions (the three words are synonyms) are named using lower cases. Actions occur in an instant and no two actions ever occur simultaneously.

The dining philosopher problem has multiple philosophers and multiple forks. We need a way to add context to the actions listed in the basic definitions of process lifecycles. We wish to separate `phil.1`'s actions from `phil.2`'s actions. In fact, we need to create five separate phil. This is done here (chapter 6):

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

DINERS is marked as a concurrent composite and carries a parameter N with default value 5. forall should make sense to all programmers 😊 `phil[i]:PHIL` is a way to add context to actions in process PHIL. The prefixes added are "phil." and "[i].". It is convenient to show "[i]" as just "i.". You have already seen the consequence of this rule during the use of LTS Analyser in Task 01.

The 3rd line in the programmed model is a bit intriguing. The context listed in set "{phil[i].left, phil[((i-1)+N)%N].right}" are synonym. They can be used interchangeably. The context will be prefixed to the actions listed in process FORK. The operator used for this rule involving synonyms is "::". This is just a syntax rule for this modelling language.

There are a few other rules related to *renaming* and *name hiding* that we will skip. See Chapter 3 slides if you are interested.

Let us move to Chapter 7 slides where we test the models for useful properties. There are two special processes defined. Processes **STOP** and **ERROR** are predefined bad processes. See slide 3 of Chapter 7 for details. These can be specified in the model by including transitions to bad processes as appropriate.

Progress property is defined on slide 25 of chapter 7. And it is expressed by listing one or more sets of actions. Each set defines a liveness requirement. A violation of progress property is reported by the analyser if the process has no future course matching the liveness requirement.

For example, we may seek to ensure that every philosopher will be able to always eat again in the future. This is somewhat stronger condition than a deadlock. Deadlock is noted if no philosopher can eat.

Deadlock is avoided if one philosopher can eat even if the others starve. The stronger progress property may prompt that every philosopher will always have next opportunity to eat. No one is left to starve forever.

At the bottom of the Dining Philosophers model is a menu statement. This is to create an animator with selected set of actions. In this example, the animator is called RUN. You can access it through Check >> run >> RUN. The path we used at Task01 start was through Check>>run>>DEFAULT.

Task 04: Readers Writers Problem – Readers First Model

Slide 43 from Chapter 7 describes and provides models for a few versions of the readers-writers problem. Read the description and work through the LTSA tool using the model accessible through File >>Examples >> Chapter7. There are three separate versions of the problem discussed in these examples.

a. Readers-Writers
b. Readers-Writers-Priority
c. Readers-Writers-Fair

These models have progress and safety properties mentioned explicitly in the models. Note that progress properties is not about the bound on the waits. No progress violation is reported if after a very long wait the process can acquire a permission to access resource.

Slides in Chapter 7 explain the examples that you can run on the analyser. Slide 46 explains the behaviour of a simple controlling lock for the readers-writers problem. There are Nread readers and Nwrite writers. Initially, RW[0][False] is set to say that 0 readers are reading (examining) and no writer is writing (modifying) the resource.

Actions of process RW_LOCK is defined in the lower half of the slide. A reader can acquire read lock when no writer is writing. After each Reader Acquire action, reader count goes up. Writer can get lock if no reader is reading and no writer is writing.

Release actions are unrestricted and can be done any time, but they have been assigned low priority. This has a consequence. For example, if a reader and a writer are seeking permission to access as the last reader is releasing its permission, the waiting reader will get access before the reader can release its permission.

Slide 47 sets the safety property. SAFE_RW. It defines the correct behaviour decorum. You can run the composite process on LTSA to view results discussed on slide 48. Slide 49 sets the progress requirement. Neither readers nor writers should be starved for ever. A finite wait, however long it may be, is not considered starvation!

Slide 50 explains that the current version of lock controller may indefinitely delay a waiting writer if the readers are very active. The LTSA, however, reports no violation for this example! This is so because wait is for finite number of epochs.

When working with ReaderWriter and other related examples, you may remove \{examine} and \{modify} from the descriptions of READER and WRITER. Do not remove dot (.) at the end! This will display these actions on your animator window.

Task 05: Writers First Model

Slides 55 onwards discusses a different version of the controller. This version prefers a waiting writer's request over the readers' requests.

Once again you can run the analyser to see that in this scenario readers can be go hungry for a long time. Discussions in Slide 56 of Chapter 7 explain the situation. This time, very active writers delay the waiting readers.

Task 06: Fair Solution

The last line on slide 58 suggests a solution. The model is listed as Reader-Writers-Fair in the examples with LTSA. The solution captures the state of system through several variables: a count of readers with permission to read, a Boolean variable to indicate if a writer has the permission to modify, a count of writer waiting for permission to modify, and a Boolean that tracks who had accessed the resource most recently. This turn is flipped between the readers and writers to ensure fairness among the two communities.

The transition diagrams that you notice in the slides are from the tool and you cans see them under draw tab.