We will wait 10 minutes until 10:40 AM for all students to join into the meeting.

We will start the tutorial at **10:40 AM**.

This tutorial will be recorded.

CS 3SD3 - Concurrent Systems Tutorial 4

Mahdee Jodayree

October 05, 2021

Outline

- Announcements / Reminders
- Reminder: Name of characters.
- Reminder: Composition in LTSA
- Mutual exclusion
- How to Draw Shared Object and Mutual Exclusion in LTSA.
- LTSA Labelling is not mutual exclusion DO NOT mix them up.
- LTSA labeling vs mutual exclusion.
- Revision on Designing a Concurrent System
- Arrays in LTSA tool, from lecture notes.
- Run codes in LTSA tool.
- Questions.

Announcements

- Assignment 1 deadline has been extended to October 6th Wednesday 23:59.
- No tutorial or class next week , October 11-17, **Fall Break**.

Reminder: Name of Characters.

{ }	Braces or curly braces
[ ]	Brackets
( )	Parenthesis
:	Colon
::	Double Colon
|	Bar or Pipe
||	Double Bar or Double pipe

More information can be found in this webpage.

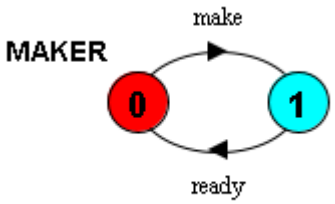
Reminder: Composition in LTSA

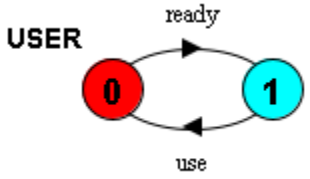
Maker-User example:

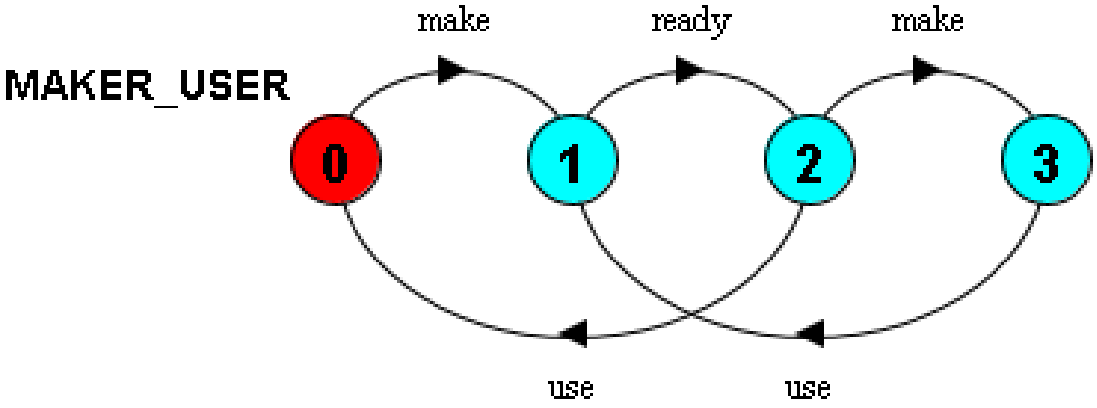
MAKER = (make -> ready -> MAKER).

USER = (ready -> use -> USER).

|| MAKER_USER = (MAKER || USER).





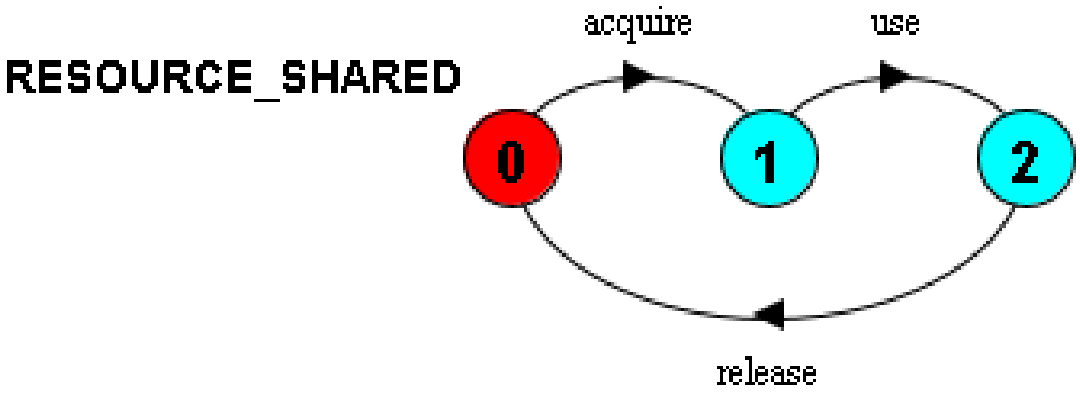


Another Composition example.

RESOURCE = (acquire->release->RESOURCE).

USER = (acquire->use->release->USER).

||RESOURCE_SHARED = (USER || RESOURCE).



What is a mutual exclusion?

- The simple translation for mutual exclusion is **taking turns, when accessing a shared resource (one at a time).**
- If we have a shared variable or resource, by mutual exclusion we can ensure that variable is accessed one at a time.
- We need this, when there are multiple threads or users.
- It prevents race conditions.
- A race condition occurs when two threads access a shared variable at the same time. The first thread reads the variable, and the second thread reads the same value from the variable.

Remember the example from lecture 1?

What will happen if we run this code with multithreading code?

```
class Counter {
        private int c;
                public void incrementone() {
                        c++;
                        }
            public void incrementBytwo() {
                    c=c+2;          }
            public int value() {
                    return c;   }
        }
```

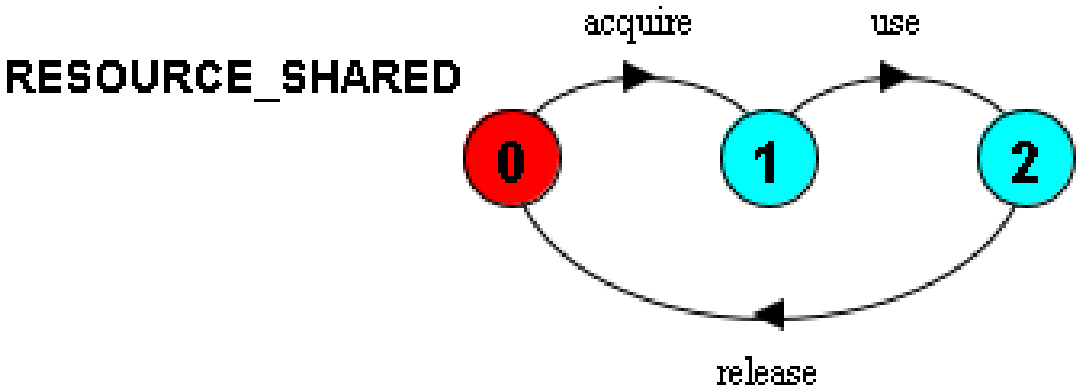
How can we Draw Shared Object and Mutual Exclusion in LTSA?

Another Composition example.

RESOURCE = (acquire->release->RESOURCE).

USER = (acquire->use->release->USER).

||RESOURCE_SHARED = (USER || RESOURCE).

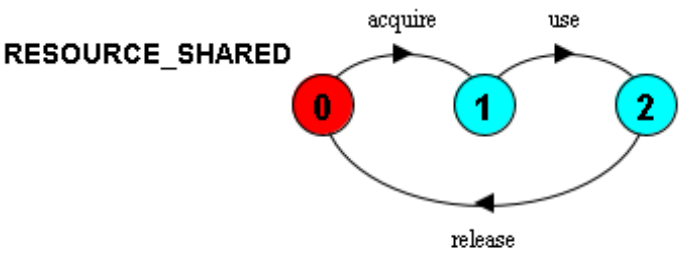


Shared Object and Mutual Exclusion in LTSA

RESOURCE = (acquire->release->RESOURCE).

USER = (acquire->use->release->USER).

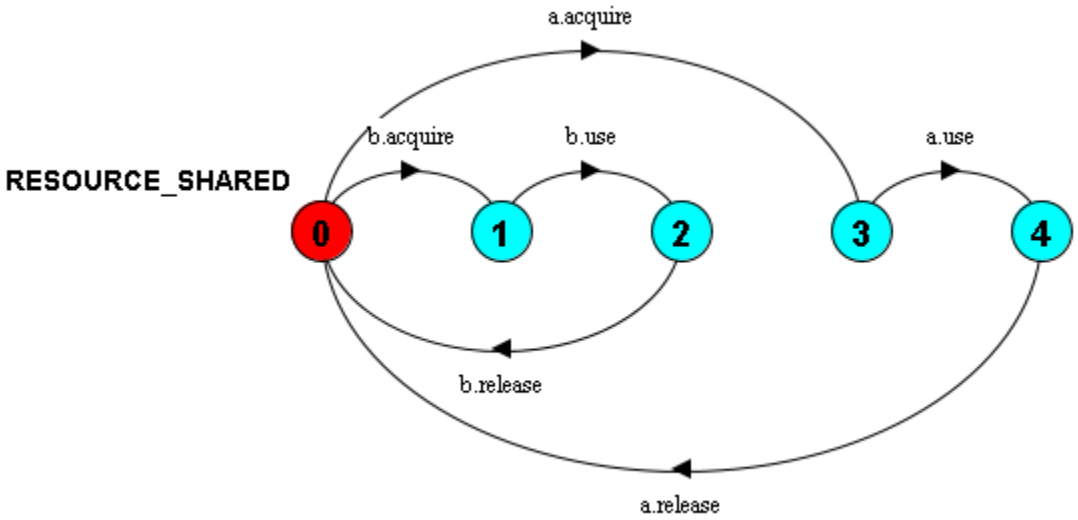
||RESOURCE_SHARED = (USER || RESOURCE).



RESOURCE = (acquire->release->RESOURCE).

USER = (acquire->use->release->USER).

||RESOURCE_SHARED = ({a, b}:USER || {a,b}:: RESOURCE).

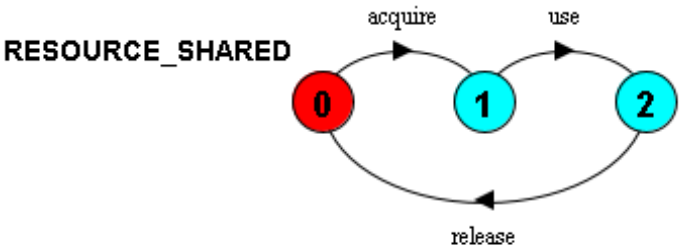


Shared Object and Mutual Exclusion in LTSA

RESOURCE = (acquire->release->RESOURCE).

USER = (acquire->use->release->USER).

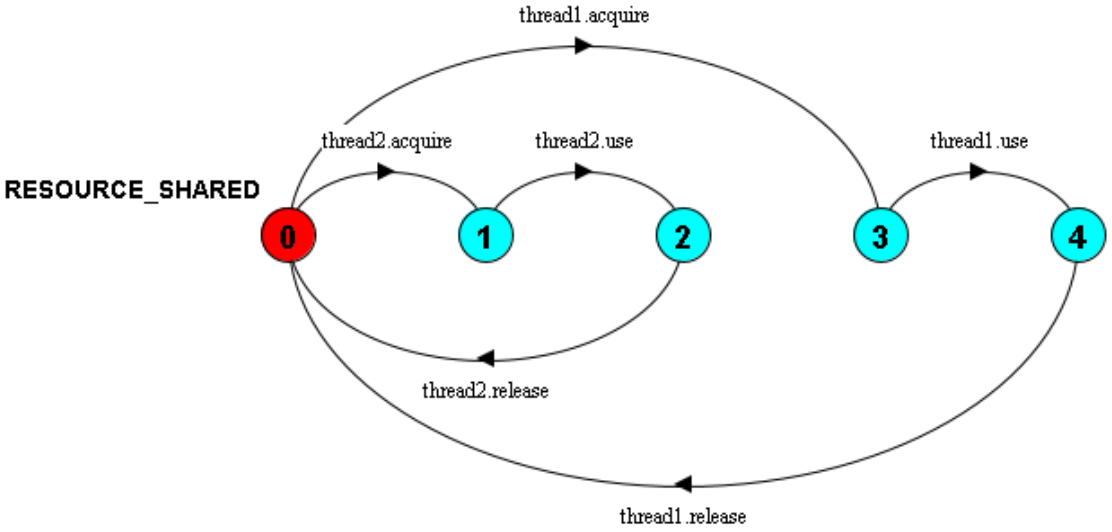
||RESOURCE_SHARED = (USER || RESOURCE).



RESOURCE = (acquire->release->RESOURCE).

USER = (acquire->use->release->USER).

||RESOURCE_SHARED = (**{thread1, thread2}:**USER || **{thread1,thread2}::** RESOURCE).



LTSA Labelling is not mutual exclusion DO NOT mix them up.

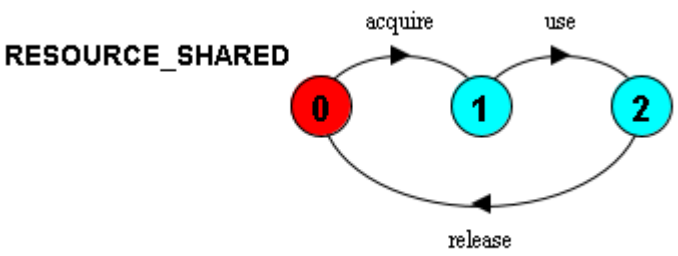
LTSA allows you to change the label of actions as the following

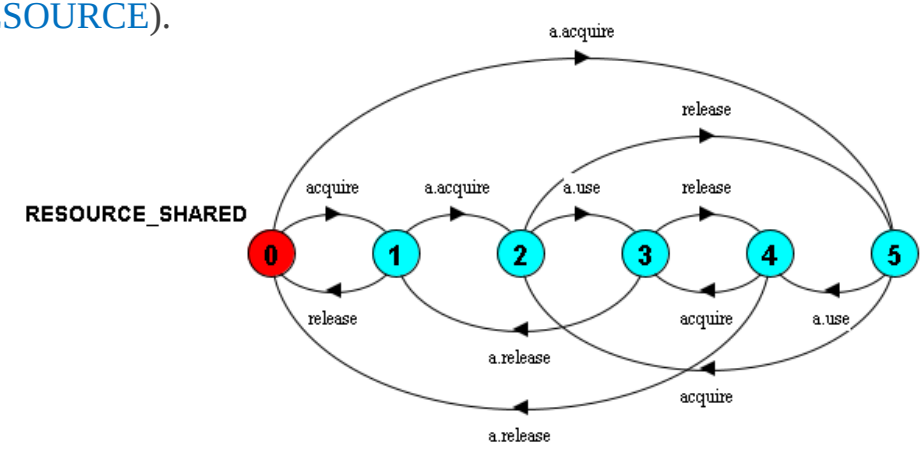
This is different from mutual exclusion.

RESOURCE = (acquire->release->RESOURCE).

USER = (acquire->use->release->USER).

||RESOURCE_SHARED = (**a:**USER || RESOURCE).





LTSA Labelling is not mutual exclusion DO NOT mix them up.

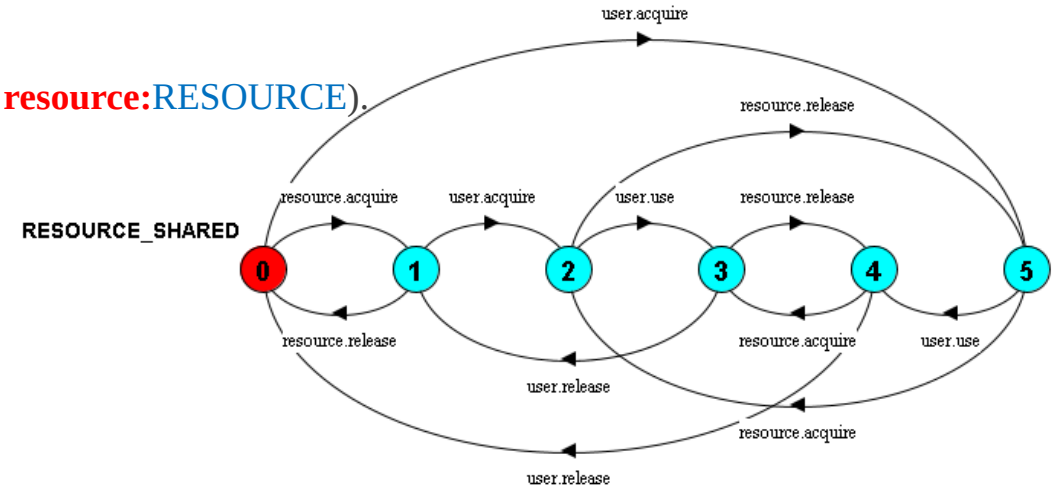
LTSA allows you to change the label of actions as the following

This is different from mutual exclusion.

RESOURCE = (acquire->release->RESOURCE).

USER = (acquire->use->release->USER).

||RESOURCE_SHARED = (**user:**USER || **resource:**RESOURCE).



LTSA labeling vs mutual exclusion.

- In this course we do not need to use labeling with LTSA.
- In this course we will use mutual exclusion a lot.
- Please do not mix labeling with mutual exclusion.
- However if you notice, sometimes you can use the labeling in your assignments to help you to draw a diagram by hand, but it is very confusing.

~~||RESOURCE_SHARED = (**user:**USER || **resource:**RESOURCE).~~

Do not forget.

The most important part of mutual exclusion code in LTSA tool is the double colons **::**

```
RESOURCE = (acquire->release->RESOURCE).

USER = (acquire->use->release->USER).

||RESOURCE_SHARED = ({a, b}:USER || {a,b}::RESOURCE).
```

This has a different meaning.

If you forget double colons in your code, it will mean labeling and does not mean mutual exclusion.

For example the following code will mean

||RESOURCE_SHARED = (**{a, b}:**USER || **{a,b}:** RESOURCE).

||RESOURCE_SHARED = (**a:**USER || **b:**USER || **a:** RESOURCE || **b:** RESOURCE).

This will mean labeling and a composition of 4 different state with a and b labeling and the produced diagram will have completely different meaning.

Revise on Designing a Concurrent System

How to design a good concurrent system?

- Identify primitive processes
- Active processes => Thread
- Passive processes => Monitor (shared object)
  - Identify synchronization points
- If future you will learn that monitors can help you to use more than 2 threads. (This is for future, you do not need to understand this now).

Codes from lecture notes.

- To understand this code from the lecture notes, (lecture 6, slide 5) let quickly review
  - Guarded actions
  - For loop in LTSA.

I fully explained for loops in LTSA, last week.

```
const N = 4
range T = 0..N
set VarAlpha = { value.{read[T],write[T]} }

VAR       = VAR[0],
VAR[u:T]  = (read[u]    ->VAR[u]
            |write[v:T]->VAR[v]).

TURNSTILE = (go     -> RUN),
RUN       = (arrive-> INCREMENT
            |end    -> TURNSTILE),
INCREMENT = (value.read[x:T]
             -> value.write[x+1]->RUN
            )+VarAlpha.

||GARDEN = (east:TURNSTILE || west:TURNSTILE
            || { east,west,display}::value:VAR)
             /{ go /{ east,west} .go,
               end/{ east,west} .end} .
```

The alphabet of shared process **VAR** is declared explicitly as a **set** constant, **VarAlpha**.

The **TURNSTILE** alphabet is extended with **VarAlpha** to ensure no unintended free (autonomous) actions in **VAR** such as **value.write[0]**.

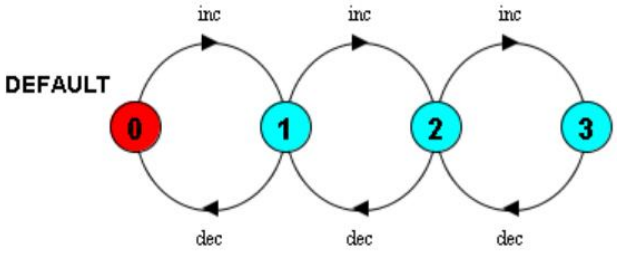
All actions in the shared **VAR** must be controlled (shared) by a **TURNSTILE**.

FSP guarded actions

- You need to know this for Assignment 1, question 10.
- LTSA examples Chapter 2 → Count
- From 0 to 3 (range), do the following, when i<N,…

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

We are defining i in the second line, LTSA is different from JAVA



FSP guarded actions

- There is a another way of writing the code on the right side.
- For the assignment 1, question 10 , use the approach on the left side.

const N=3

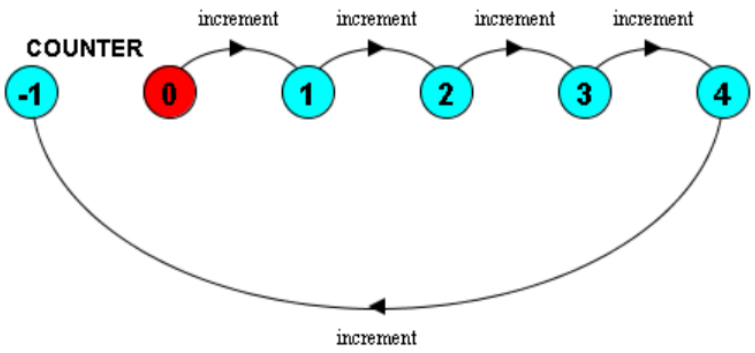
count = count [0],

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

Guarded actions

LTSA examples, Chapter 4 -> counter

```
const N = 4
range T = 0..N

COUNTER = COUNTER[0],
COUNTER[v:T] = (increment ->COUNTER[v+1]).
```



Arrays in LTSA tool

```
const N = 4
range T = 0..N
set VarAlpha = { value.{read[T],write[T]} }

VAR       = VAR[0],
VAR[u:T] = (read[u]    ->VAR[u]
           |write[v:T]->VAR[v]).

TURNSTILE = (go     -> RUN),
RUN       = (arrive-> INCREMENT
           |end    -> TURNSTILE),
INCREMENT = (value.read[x:T]
            -> value.write[x+1]->RUN
           )+VarAlpha.

||GARDEN = (east:TURNSTILE || west:TURNSTILE
           || { east,west,display}::value:VAR)
            /{ go /{ east,west} .go,
              end/{ east,west} .end} .
```

```
const N = 4
range T = 0..N

set VarAlpha = {value.read[T],value.write[T]}

VAR = VAR[0],
VAR[u:T] = (read[u] ->VAR[u]
|write[v:T]->VAR[v]).
TURNSTILE = (go -> RUN),
RUN = (arrive-> INCREMENT
|end -> TURNSTILE),
INCREMENT = (value.read[x:T]
-> value.write[x+1]->RUN
)+VarAlpha.
||GARDEN = (east:TURNSTILE || west:TURNSTILE
|| { east,west,display}::value:VAR)
/{ go /{ east,west} .go,
end/{ east,west} .end} .
```

Reminder: What is for loop?

- In other programing languages, if we wanted to program a for loop, this is how we would done it Java.

```
for (int N = 0; N < 4; N++)
{
    System.out.println("The Value of N is: "+N);


} // at the end of the loop the loop will automatically increment the value of N
```

Result

```
The Value of N is: 0
The Value of N is: 1
The Value of N is: 2
The Value of N is: 3
```

- In Java this translates to the following code where there programing is trying to create array

```
int[] Writearray = new int[5];
int[] Readarray  = new int[5];
```

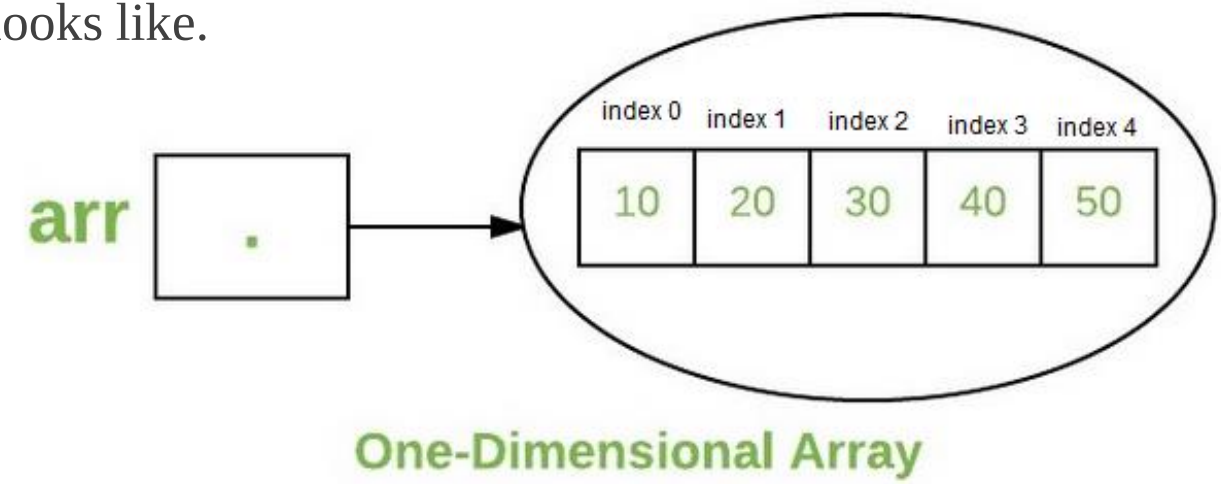
- If you need to set a value in the array in Java you would use

```
Writearray[0]=10;
```

- In **LTSA, an** array is called a **set**.

The alphabet of shared process **VAR** is declared explicitly as a **set** constant, **VarAlpha**.

- Reminder what a array looks like.



```
const N = 4
range T = 0..N
set VarAlpha = { value.{read[T],write[T]} }

VAR      = VAR[0],
VAR[u:T] = (read[u]    ->VAR[u]
           |write[v:T]->VAR[v]).

TURNSTILE = (go     -> RUN),
RUN       = (arrive-> INCREMENT
           |end    -> TURNSTILE),
INCREMENT = (value.read[x:T]
            -> value.write[x+1]->RUN
            )+VarAlpha.

||GARDEN = (east:TURNSTILE || west:TURNSTILE
           || { east,west,display}::value:VAR)
            /{ go /{ east,west} .go,
              end/{ east,west} .end} .
```

```
for (int N = 0; N < 4; N++)
{
```

```
set VarAlpha = {value.read[T],value.write[T]}
```

```
int[] value-Writearray = new int[5];
int[] value-Readarray  = new int[5];
```

This is declaring a variable called X and the value of the x in the first line is equal to the current value of T and in the next line value of the write = value of T plus 1

INCREMENT = (value.read[0] --> value.write[1] --> RUN
INCREMENT = (value.read[1] --> value.write[2] --> RUN
INCREMENT = (value.read[2] --> value.write[3] --> RUN
INCREMENT = (value.read[3] --> value.write[4] --> RUN

Please note that +VarAlpha will increment the value of T every where in the program.

```
const N = 4
range T = 0..N

set VarAlpha = {value.read[T],value.write[T]}

VAR = VAR[0],

VAR[u:T] = (read[u] ->VAR[u] |write[v:T]->VAR[v]).

TURNSTILE = (go -> RUN),

RUN = (arrive-> INCREMENT |end -> TURNSTILE),

INCREMENT = (value.read[x:T] -> value.write[x+1]->RUN)
+VarAlpha.

||GARDEN = (east:TURNSTILE || west:TURNSTILE || { east,west,display}::value:VAR)/{ go /{ east,west}.
go,end/{ east,west} .end} .
```

This declares a Set call VAR and its length is not defined here. Similar to array in Java.

In here it declares a variable v and sets the value of v to the current value of T.

In here it declares a variable u and sets the value of u to the current value of T

This line means
VAR[0]=(read[0]→VAR[0] | write [0] →VAR [0]).
VAR[1]=(read[1]→VAR[1] | write [1] →VAR [1]).
VAR[2]=(read[2]→VAR[2] | write [2] →VAR [2]).
VAR[3]=(read[3]→VAR[3] | write [3] →VAR [3]).

What does this part mean?

```
const N = 4
range T = 0..N
set VarAlpha = { value.{read[T],write[T]} }

VAR       = VAR[0],
VAR[u:T]  = (read[u]    ->VAR[u]
            |write[v:T]->VAR[v]).

TURNSTILE = (go     -> RUN),
RUN       = (arrive-> INCREMENT
            |end    -> TURNSTILE),
INCREMENT = (value.read[x:T]
             -> value.write[x+1]->RUN
            )+VarAlpha.

||GARDEN = (east:TURNSTILE || west:TURNSTILE
            || { east,west,display}::value:VAR)
            /{ go /{ east,west} .go,
              end/{ east,west} .end} .
```

The alphabet of shared process **VAR** is declared explicitly as a **set** constant, **VarAlpha**.

The **TURNSTILE** alphabet is extended with **VarAlpha** to ensure no unintended free (autonomous) actions in **VAR** such as **value.write[0]**.

All actions in the shared **VAR** must be controlled (shared) by a **TURNSTILE**.

```
const N = 4
range T = 0..N

set VarAlpha = {value.read[T],value.write[T]}

VAR = VAR[0],

VAR[u:T] = (read[u] ->VAR[u] |write[v:T]->VAR[v]).
TURNSTILE = (go -> RUN),

RUN = (arrive-> INCREMENT |end -> TURNSTILE),

INCREMENT = (value.read[x:T] -> value.write[x+1]->RUN)
+VarAlpha.

||GARDEN = (east:TURNSTILE || west:TURNSTILE || { east,west,display}::value:VAR)/{ go /{ east,west}.
go,end/{ east,west} .end} .
```

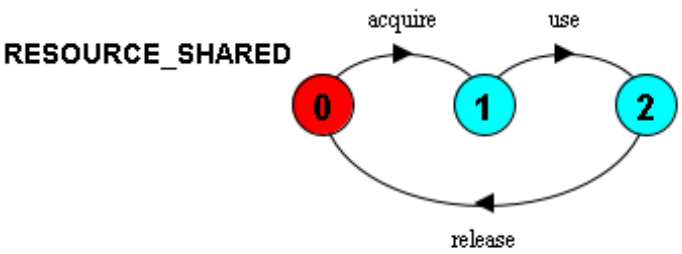
Garden is a composition **Turnstile** and **Var** and they are both mutually excluded to ***east*** and ***west***. It means when east uses the thread west cannot use it, and vise versa.

Shared Object and Mutual Exclusion in LTSA

RESOURCE = (acquire->release->RESOURCE).

USER = (acquire->use->release->USER).

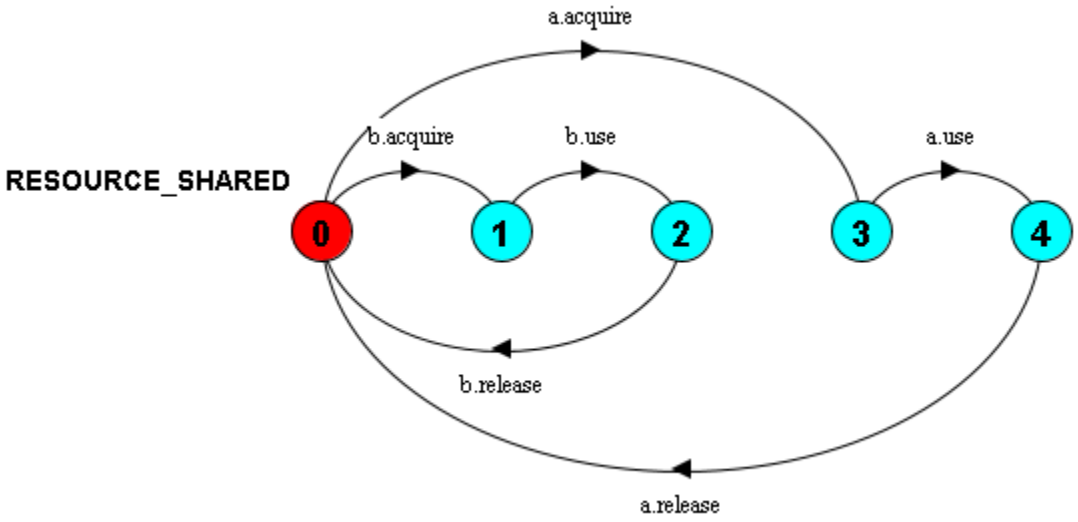
||RESOURCE_SHARED = (USER || RESOURCE).



RESOURCE = (acquire->release->RESOURCE).

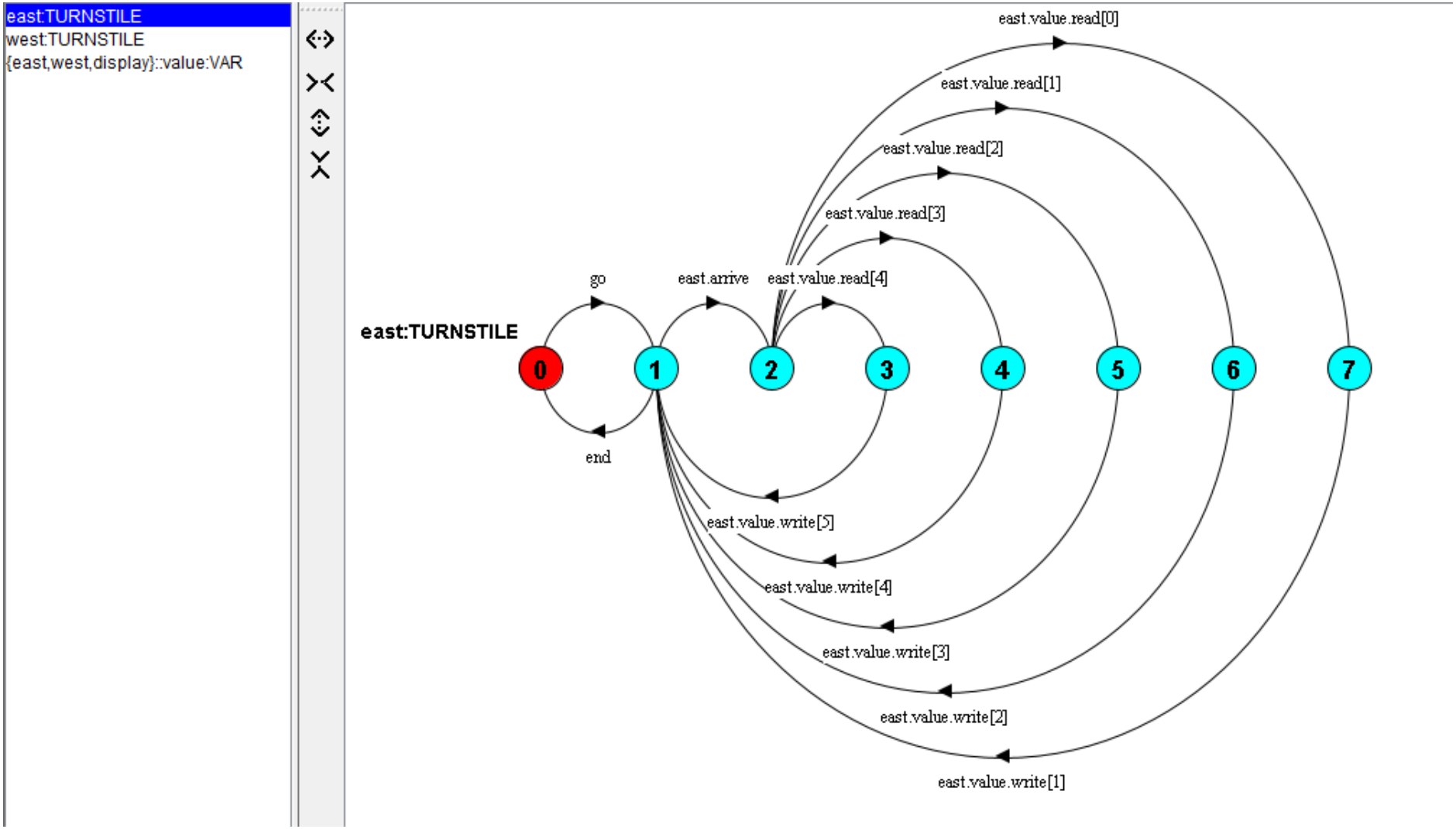
USER = (acquire->use->release->USER).

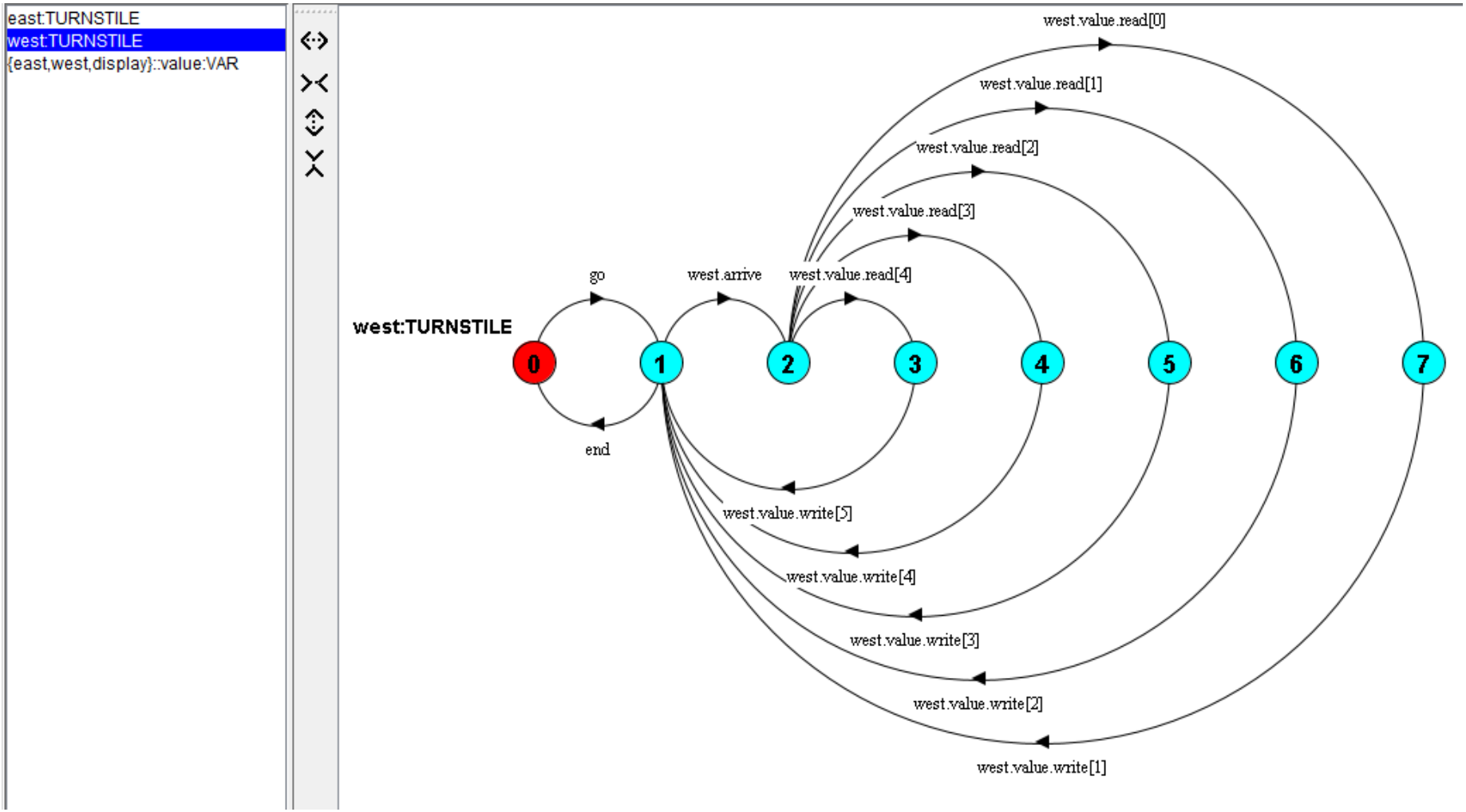
||RESOURCE_SHARED = ({a, b}:USER || {a,b}:: RESOURCE).

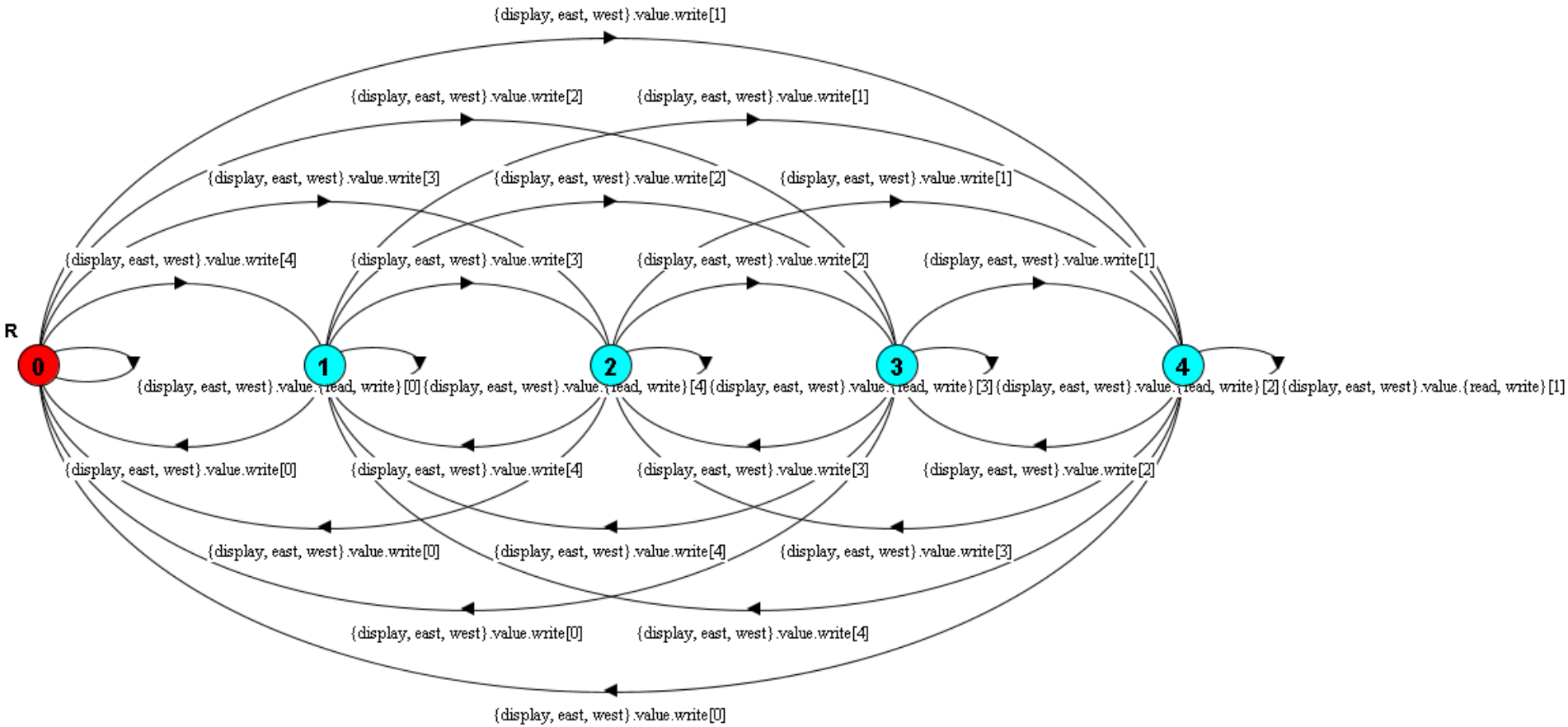


Lets run this code.

```
const N = 4

range T = 0..N

set VarAlpha = {value.read[T],value.write[T]}

VAR = VAR[0],

VAR[u:T] = (read[u] ->VAR[u] |write[v:T]->VAR[v]).

TURNSTILE = (go -> RUN),

RUN = (arrive-> INCREMENT |end -> TURNSTILE),

INCREMENT = (value.read[x:T] -> value.write[x+1]->RUN)

+VarAlpha.


||GARDEN = (east:TURNSTILE || west:TURNSTILE || { east,west,display}::value:VAR)/{ go /{ east,west}.

go,end/{ east,west} .end} .
```

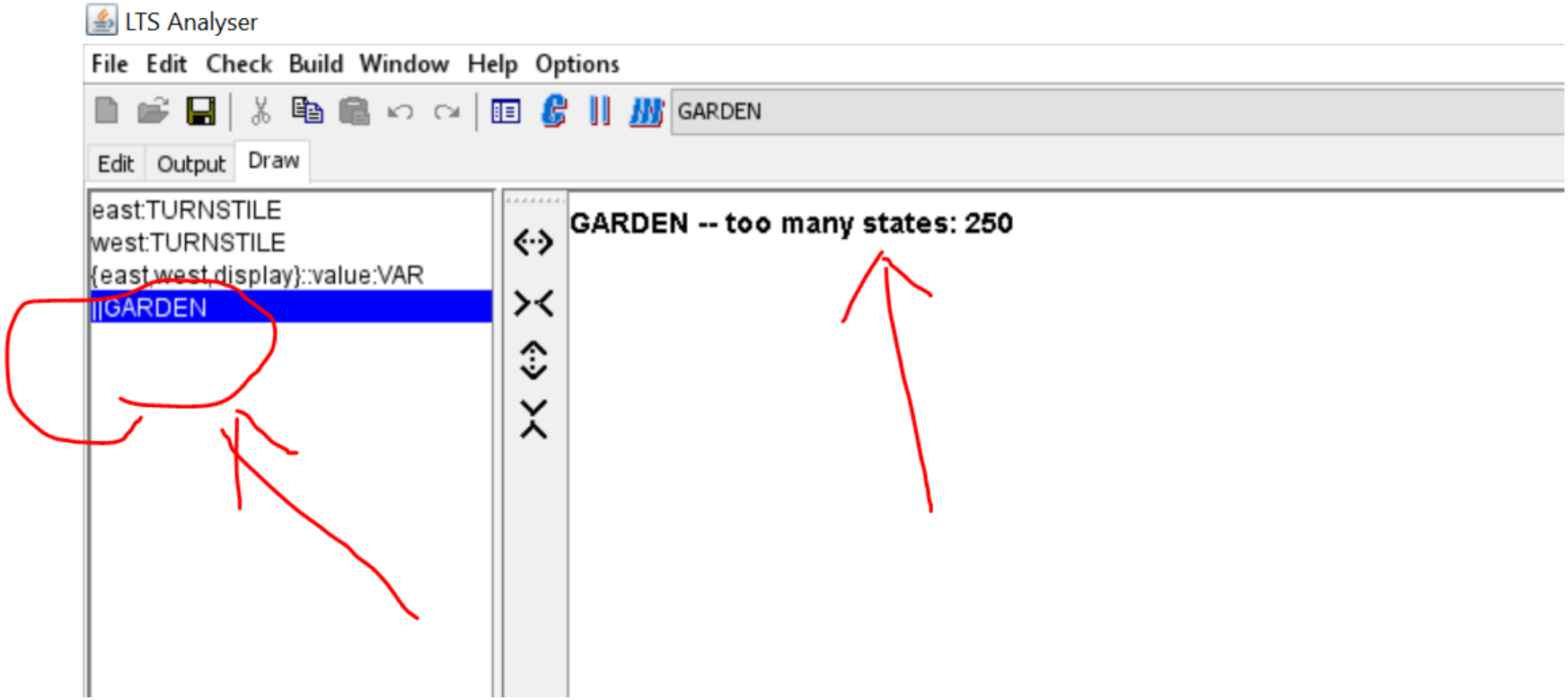

east:TURNSTILE
west:TURNSTILE
{east,west,display}::value:VAR
east:TURNSTILE
0
1
2
3
4
5
6
7
go
end
east.arrive
east.value.read[0]
east.value.read[1]
east.value.read[2]
east.value.read[3]
east.value.read[4]
east.value.write[5]
east.value.write[4]
east.value.write[3]
east.value.write[2]
east.value.write[1]


east:TURNSTILE
west:TURNSTILE
{east,west,display}::value:VAR
west:TURNSTILE
go
end
west.arrive
west.value.read[0]
west.value.read[1]
west.value.read[2]
west.value.read[3]
west.value.read[4]
west.value.write[5]
west.value.write[4]
west.value.write[3]
west.value.write[2]
west.value.write[1]
0
1
2
3
4
5
6
7



But if we run these two processes concurrently, they do not work property.

LTSA Screenshot.



Any Questions?