Good-Enough Design

Kenneth M. Anderson
University of Colorado, Boulder

CSCI 5828 — Lecture 11 — 02/17/2009

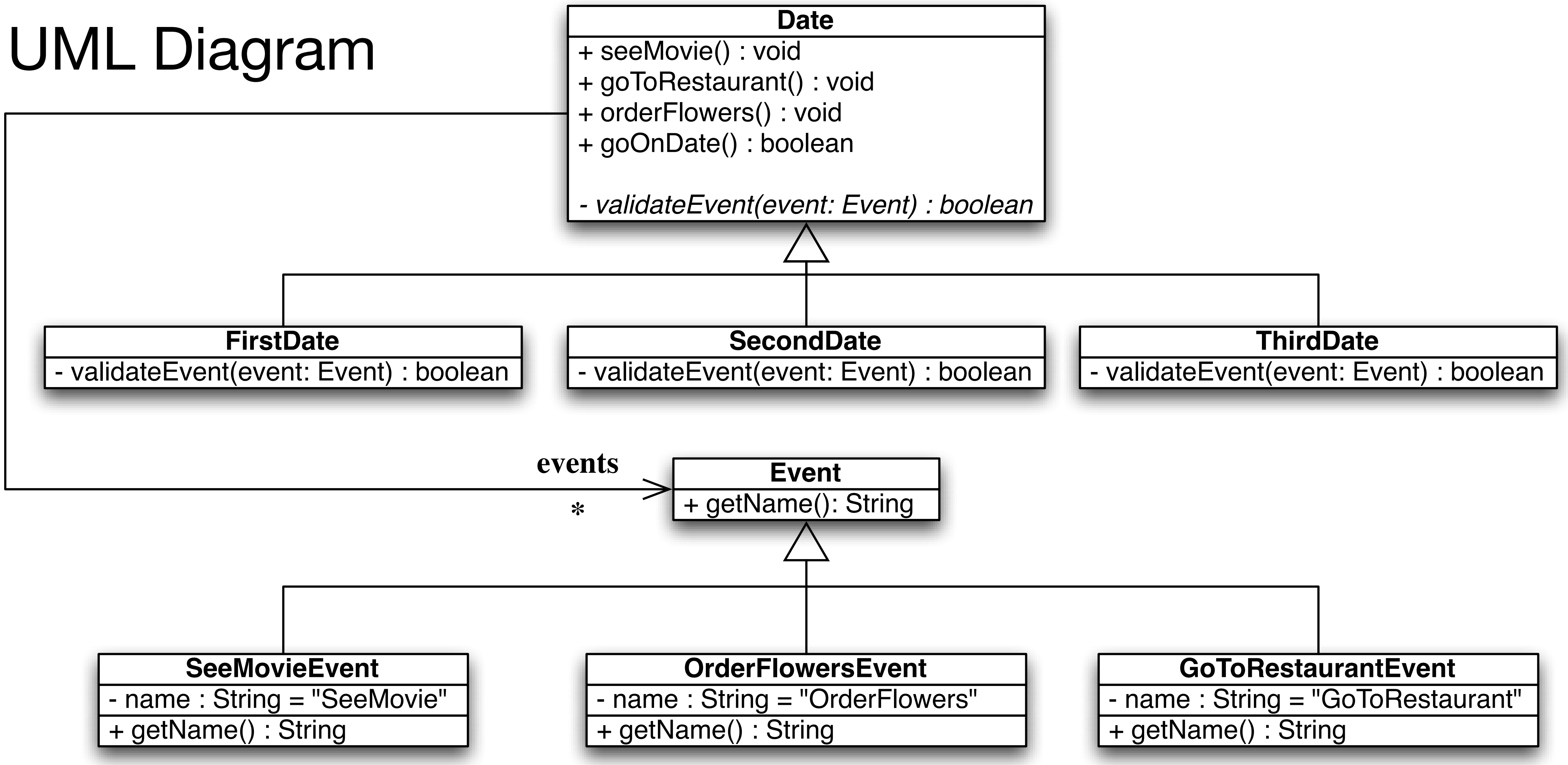
Goals

- Review material from Chapter 5 of Pilone & Miles
  - Software Design
    - Need for Good OO A&D principles
    - SRP: Single Responsibility Principle
    - DRY: Don’t Repeat Yourself Principle

iSwoon in Trouble

- The previous chapter presents a design for associating dates and events that was causing problems
  - Date objects maintain a list of its planned events
    - An Event object is a “dumb data holder” storing only a name
      - It has no logic of its own
  - Date objects provide methods that internally add events to a planned date; The Date object contains information about what events are allowed on a particular date
- The UML diagram is shown on the next slide

UML Diagram

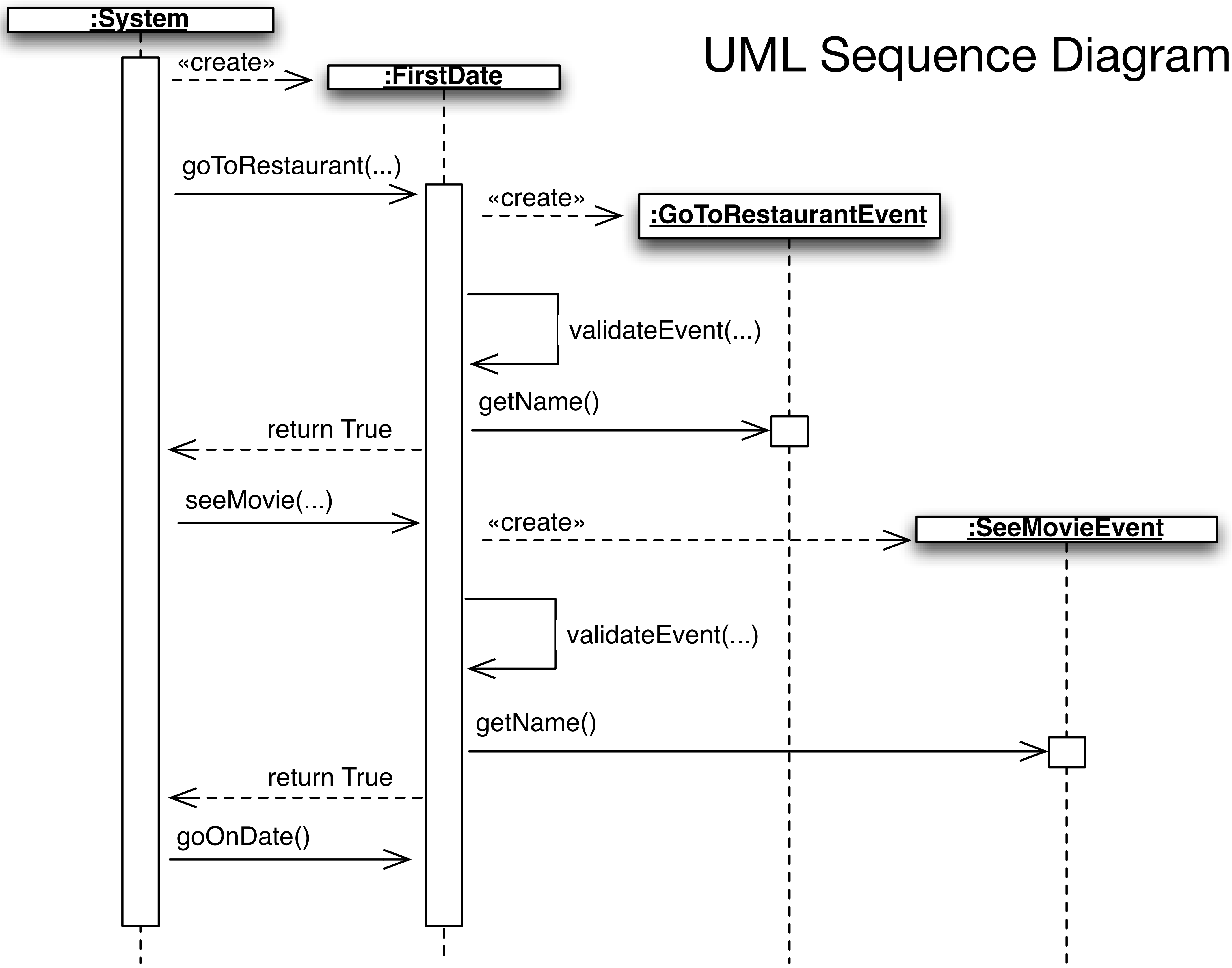


Date
+ seeMovie() : void
+ goToRestaurant() : void
+ orderFlowers() : void
+ goOnDate() : boolean
- validateEvent(event: Event) : boolean
FirstDate
- validateEvent(event: Event) : boolean
SecondDate
- validateEvent(event: Event) : boolean
ThirdDate
- validateEvent(event: Event) : boolean
events
*
Event
+ getName(): String
SeeMovieEvent
- name : String = "SeeMovie"
+ getName() : String
OrderFlowersEvent
- name : String = "OrderFlowers"
+ getName() : String
GoToRestaurantEvent
- name : String = "GoToRestaurant"
+ getName() : String

UML Primer:

Each rectangle represents a class that can have attributes and methods. A “+” symbols refers to “public” visibility; “-” indicates private visibility. The “*” means zero or more. The “large triangle” indicates inheritance. The arrow head indicates “one way navigation”; in the diagram above Dates know about Events while Events are blissfully unaware of Dates

UML Sequence Diagram



Bad Design (I)

- This design has a lot of problems
  - The Event class is completely useless
    - Why not have Date store an array of strings?
  - Date's API is pretty bad
    - Event creation methods are specified for all possible events; that means that some dates have event creation methods for events that are not valid for them!
    - The Date class has a list of allowable events but doesn't show it on the diagram (or it doesn't show the list of planned events; either way it has two lists but only shows one)

Bad Design (II)

- But those are relatively minor issues
  - The main reason why this design is bad is that its **inflexible** with respect to the types of changes that occur regularly for this application domain
    - It can’t easily handle the addition of a new type of Event
    - It can’t easily handle changing the name of an existing Event
    - It can’t easily handle the changing of what events are valid for what dates

Good Design

- A primary goal in OO A&D is producing a design that makes
  - **likely changes, straightforward**
    - typically by adding a new subclass of an existing class
    - or by adding an object that implements a known interface
  - no need to recompile the system or even it bring it down
- You can't anticipate **arbitrary changes** and there is no reason to invest time/$$ into planning for **unlikely** changes
  - So use good OO A&D principles to handle likely changes

Single Responsibility Principle (SRP) (I)

- The Date class has multiple responsibilities
  - tracking the events planned for a date
  - tracking the events allowed for a date
- It has multiple reasons to change
- The single responsibility principle says
  - Every object in your system should have a single responsibility and all the object’s services should be focused on carrying out that single responsibility
    - This is also known as “having high cohesion”

SRP (II)

- Granularity?
  - When we say “responsibility” we are not talking about low level concerns, such as
    - “insert element *e* into array *a* at position *i*”
  - but design level concerns, such as
    - “classify documents by keyword”
    - “store client details”
    - “manage itinerary of Jack and Jill’s second date”

SRP (III)

- The existing iSwoon design is bad because each time we add a new event
  - We need to add a new Event subclass
  - Add a new method to Date
  - Update each of Date’s subclasses (cringe!)
- We need to migrate to a design, in which the addition of a new type of event results in the addition of a new Event subclass and nothing more

Textual Analysis (I)

- One way of identifying high cohesion in a system is to do the following
  - For each class C
    - For each method M
      - Write “The C Ms itself”
  - Examples
    - The Automobile drives itself
    - The Automobile washes itself
    - The Automobile starts itself

Textual Analysis (II)

- Sometimes you need to include parameters in the sentence
  - The CarWash washes the Automobile itself
- If any of these sentences doesn’t make sense then investigate further
  - You may have discovered a service **that belongs to a different responsibility of the system** and should be **moved to a different class**
  - This may require first **creating a new class** before performing the move

Textual Analysis (III)

- Textual analysis is a good heuristic
  - While its useful for spot checking a design, its not perfect
- But the underlying principle is sound
  - Each class in your design should “pull its weight”
    - have a single responsibility with a nice balance of both data AND behavior for handling that responsiblity

Other Problems

- The iSwoon design also has problems with duplication of information (indeed duplication can often lead to classes with “low cohesion” that violate SRP
  - The duplication in iSwoon is related to Event Types
    - The names of event types appear in
      - Event subclass names
      - The name attribute inside of each event subclass
      - The method names in Date
  - In addition, duplication occurs with validateEvent() in each of the Date subclasses

Don’t Repeat Yourself (I)

- The DRY principle
  - Avoid duplicate code by abstracting out things that are common and placing those things in a single location
- Basic Idea
  - Duplication is Bad!
  - At all levels of software engineering: Analysis, Design, Code, and Test

DRY (II)

- We want to avoid duplication in our requirements, use cases, feature lists, etc.
- We want to avoid duplication of responsibilities in our code
- We want to avoid duplication of test coverage in our tests
- Why?

DRY (II)

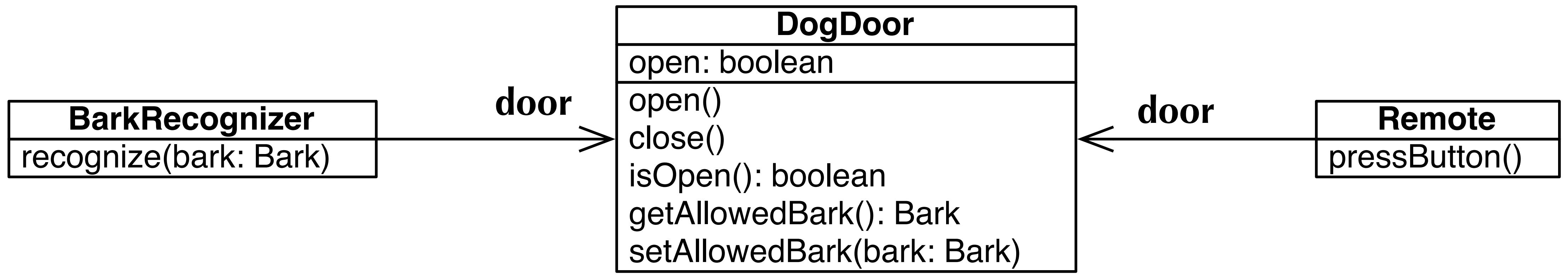
- We want to avoid duplication in our requirements, use cases, feature lists, etc.
- We want to avoid duplication of responsibilities in our code
- We want to avoid duplication of test coverage in our tests
- Why?
  - Incremental errors can creep into a system when one copy is changed but the others are not

DRY (II)

- We want to avoid duplication in our requirements, use cases, feature lists, etc.
- We want to avoid duplication of responsibilities in our code
- We want to avoid duplication of test coverage in our tests
- Why?
  - Incremental errors can creep into a system when one copy is changed but the others are not
  - Isolation of Change Requests: We want to go to ONE place when responding to a change request

Example (I)

- Duplication of Responsibility



- “The dog door should automatically close 30 seconds after it has opened”
- Where should this responsibility live?
  - It would be easy to put this responsibility in the clients
  - But it really should live in DogDoor (which method?)

Example (II)

- DRY is really about ONE requirement in ONE place
  - We want each responsibility of the system to live in a single, sensible place
- This applies at all levels of the project, including requirements
  - Imagine a set of requirements for the dog door…

Example (III)

- The dog door should alert the owner when something inside the house gets too close to the dog door
- The dog door will open only during certain hours of the day
- The dog door will be integrated into the house’s alarm system to make sure it doesn’t activate when the dog door is open
- The dog door should make a noise if the door cannot open because of a blockage outside
- The dog door will track how many times the dog uses the door
- When the door closes, the house alarm will re-arm if it was active before the door opened

Beware of Duplicates!!!

Example (IV)

- The dog door should alert the owner when something inside the house gets too close to the dog door
- The dog door will open only during certain hours of the day
- The dog door will be integrated into the house's alarm system to make sure it doesn't activate when the dog door is open
- The dog door should make a noise if the door cannot open because of a blockage outside
- The dog door will track how many times the dog uses the door
- When the door closes, the house alarm will re-arm if it was active before the door opened

Example (V)

- The dog door should alert the owner when something is too close to the dog door
- The dog door will open only during certain hours of the day
- The dog door will be integrated into the house's alarm system
- The dog door will track how many times the dog uses the door

- Duplicates removed!

Example (VI)

- Ruby on Rails makes use of DRY as a core part of its design
  - focused configuration files; no duplication of information
  - for each request, often single controller, single model update, single view
- But prior to Ruby on Rails 1.2 there was duplication hiding in the URLs used by Rails applications
  - POST /people/create # create a new person
  - GET /people/show/1 # show person with id 1
  - POST /people/update/1 # edit person with id 1
  - POST /people/destroy/1 # delete person with id 1

Example (VII)

- The duplication exists between the HTTP method name and the operation name in the URL
  - POST /people/create
- Recently, there has been a movement to make use of the four major “verbs” of HTTP
  - PUT/POST == create information (create)
  - GET == retrieve information (read)
  - POST == update information (update)
  - DELETE == destroy information (destroy)
- These verbs mirror the CRUD operations found in databases
  - Thus, saying “create” in the URL above is a duplication

Example (VIII)

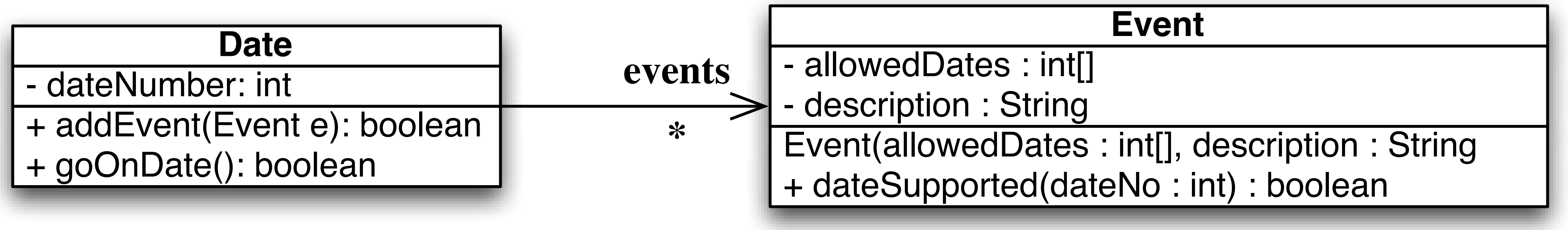
- In version 1.2, Rails eliminates this duplication for something called “resources”
- Now URLs look like this:
  - POST /people
  - GET /people/1
  - PUT /people/1
  - DELETE /people/1
- And the duplication is **logically** eliminated
  - Disclaimer: … but not **actually** eliminated… Web servers do not universally support PUT and DELETE “out of the box”. As a result, Rails uses POST
    - POST /people/1 ; Post-Semantics: DELETE

Other OO Principles

- **Classes are about behavior**
  - Emphasize the behavior of classes over the data
- **Encapsulate what varies**
  - Use classes to achieve information hiding in a design
- **One reason to change**
  - Promotes high cohesion in a design
- **Code to an Interface**
  - Promotes flexible AND extensible code
- **Open-Closed Principle**
  - Classes should be open for extension and closed for modification

Take CSCI 5448 next Fall for more details!

New iSwoon Design


Date
- dateNumber: int
+ addEvent(Event e): boolean
+ goOnDate(): boolean
events
*
Event
- allowedDates : int[]
- description : String
Event(allowedDates : int[], description : String
+ dateSupported(dateNo : int) : boolean

Subclasses eliminated; Events now keep track of what Dates they are allowed on; When you add an event to a Date, Date calls Event.dateSupported() to validate it

You can easily add a new type of Event; just create a new instance of Event with a different description; nothing else changes! To add a new date, just increase the number

Impact on Tasks

With the right design, multiple tasks estimated to take days may take only one (or less than one)

Task: Create Send Flowers Event

Estimate: 2 days

Task: Create a Book Restaurant Event

Estimate: 3 days

Task: Add Order Cab Event

Estimate: 2 days

A great design helps you be more productive!

Discussion

- The underlying message of Chapter 5 is that everyone on your team needs to understand good OO A&D principles
- On a daily basis, you look for ways in which the design can be improved
  - Small changes can occur via refactoring
  - Large changes need to become tasks and tracked like all others
- You welcome such changes since they'll make life easier and more productive down the line

Review

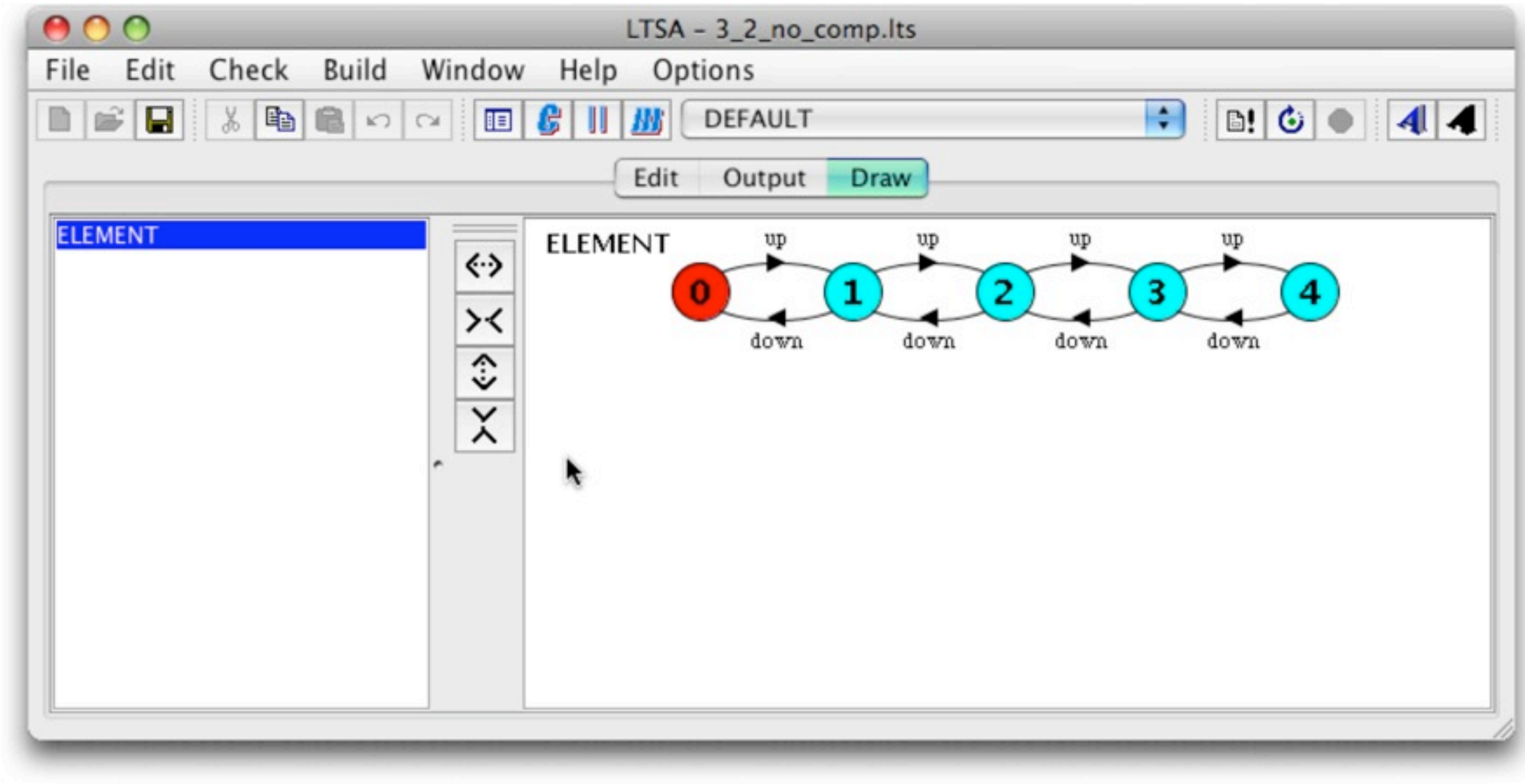
- The remainder of the lecture will be devoted to reviewing
  - Question 3.2
  - Answers to Review 2
  - Answers to Homework 1

Question 3.2

- 3.2. ELEMENT = (up -> down -> ELEMENT) accepts an "up" action and then a "down" action.
    - Using parallel composition and the ELEMENT process describe a model that can accept up to four "up" actions before a "down" action. Draw a structure diagram for your solution.
- Looks like a simple problem. Indeed it is simple if you DON’T use parallel composition (LTS on next slide)

```
ELEMENT = ELEMENT[0],
ELEMENT[i:0..4] = (when (i < 4) up -> ELEMENT[i+1] |
                   when (i > 0) down -> ELEMENT[i-1]).
```

Simple Goal

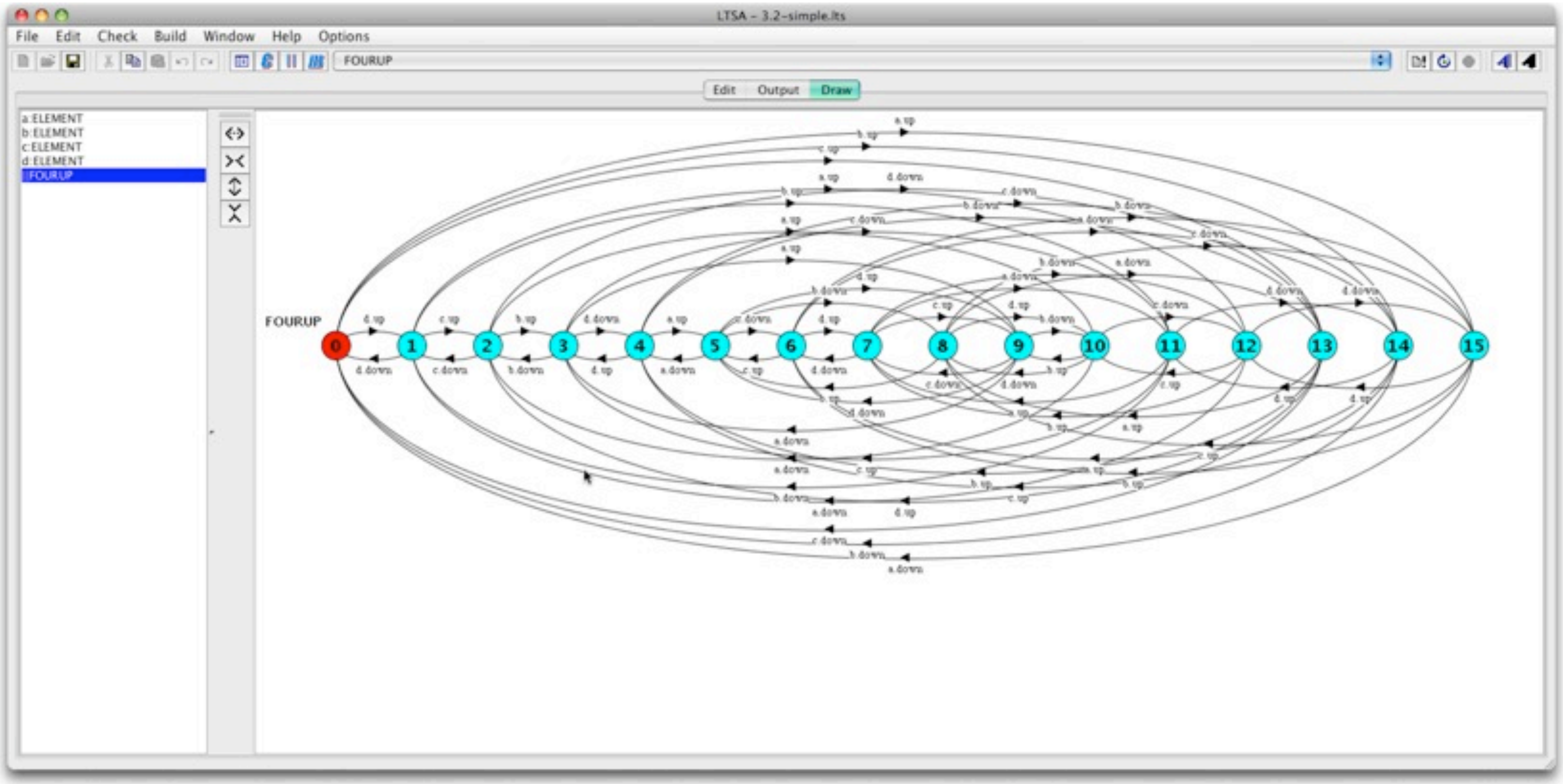

LTSA - 3_2_no_comp.lts
File
Edit
Check
Build
Window
Help
Options
DEFAULT
Edit
Output
Draw
ELEMENT
ELEMENT
0
1
2
3
4
up
up
up
up
down
down
down
down

Question 3.2 (II)

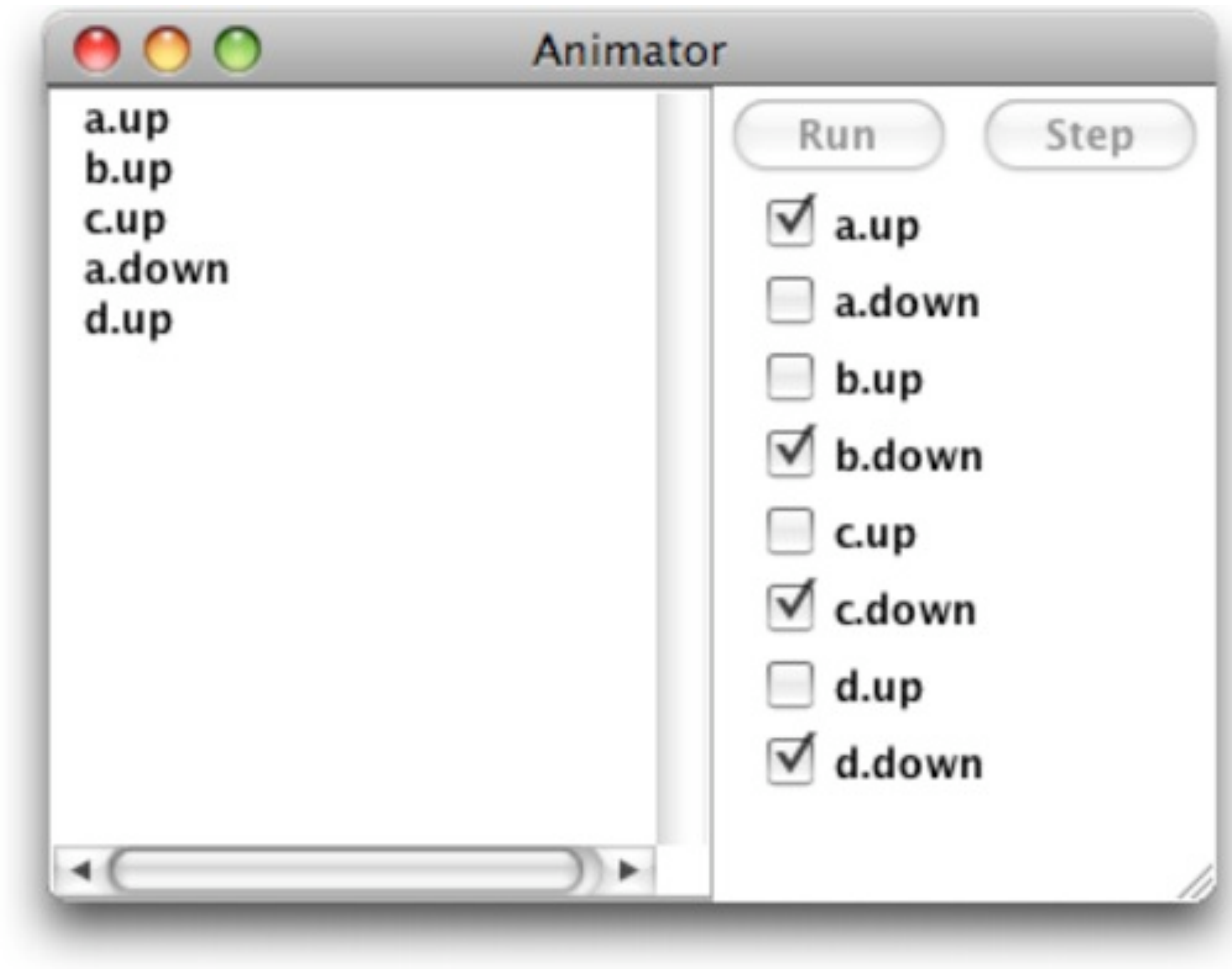
- But, this question asks that we use multiple instances of this process
  - ELEMENT = (up -> down -> ELEMENT).
- to create a parallel composition that can do the same thing that the FSP on the previous page can do.
- In order to do this, we'll need at least four instances of ELEMENT. Lets start with:

```
ELEMENT = (up -> down -> ELEMENT).

||FOURUP = (a: ELEMENT || b: ELEMENT || c: ELEMENT || d: ELEMENT).
```



Here we can see that four processes with two actions each (with no shared actions) produce an LTS with 16 states (2 * 2 * 2 * 2) and on the surface very complex behavior.



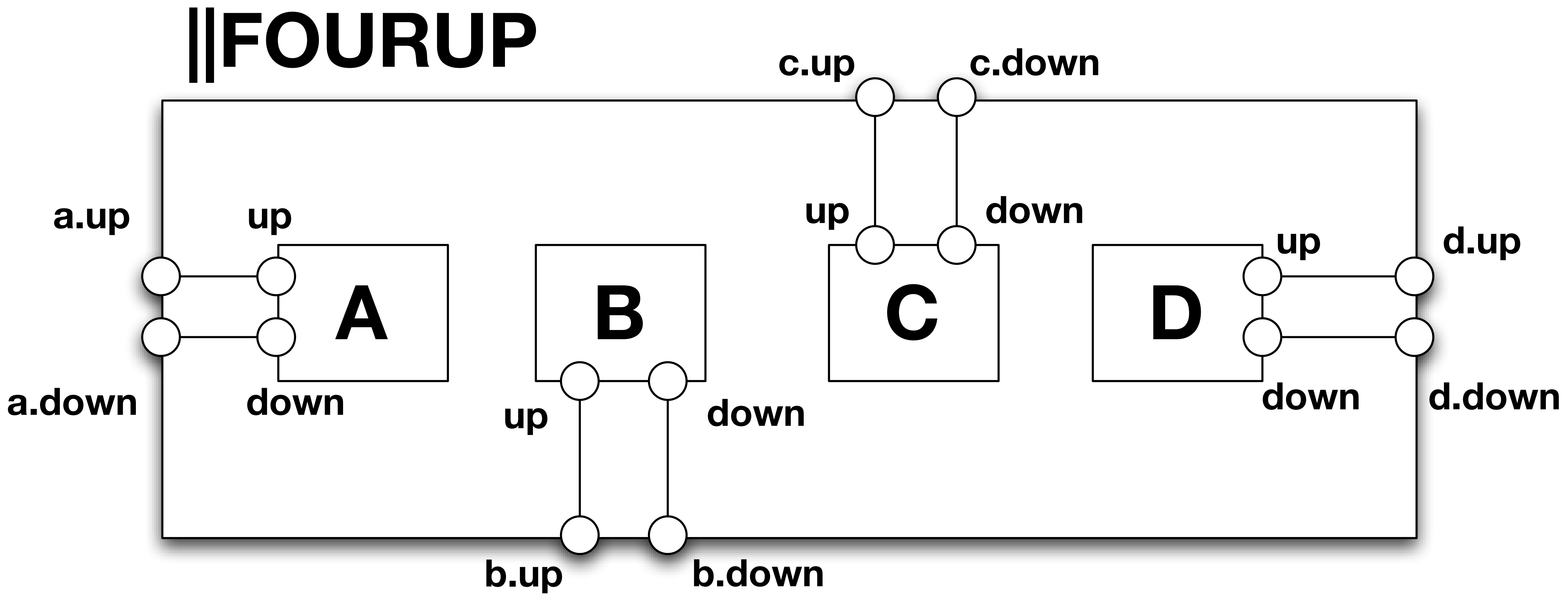
```
a.up
b.up
c.up
a.down
d.up
```

- [✓] a.up
- [ ] a.down
- [ ] b.up
- [✓] b.down
- [ ] c.up
- [✓] c.down
- [ ] d.up
- [✓] d.down

But the complexity is only skin deep. In actuality, each process can only go "up" followed by a "down". The complexity comes from allowing these simple behaviors to arbitrarily interleave.

What we need to do is relabel actions such that sharing occurs between the processes, reducing complexity, and moving us towards the behavior we want to see: 4 "ups" before a "down".

Structure Diagram?



No surprises here. Four instances of ELEMENT, no shared actions, all actions become part of ||FOURUP's alphabet

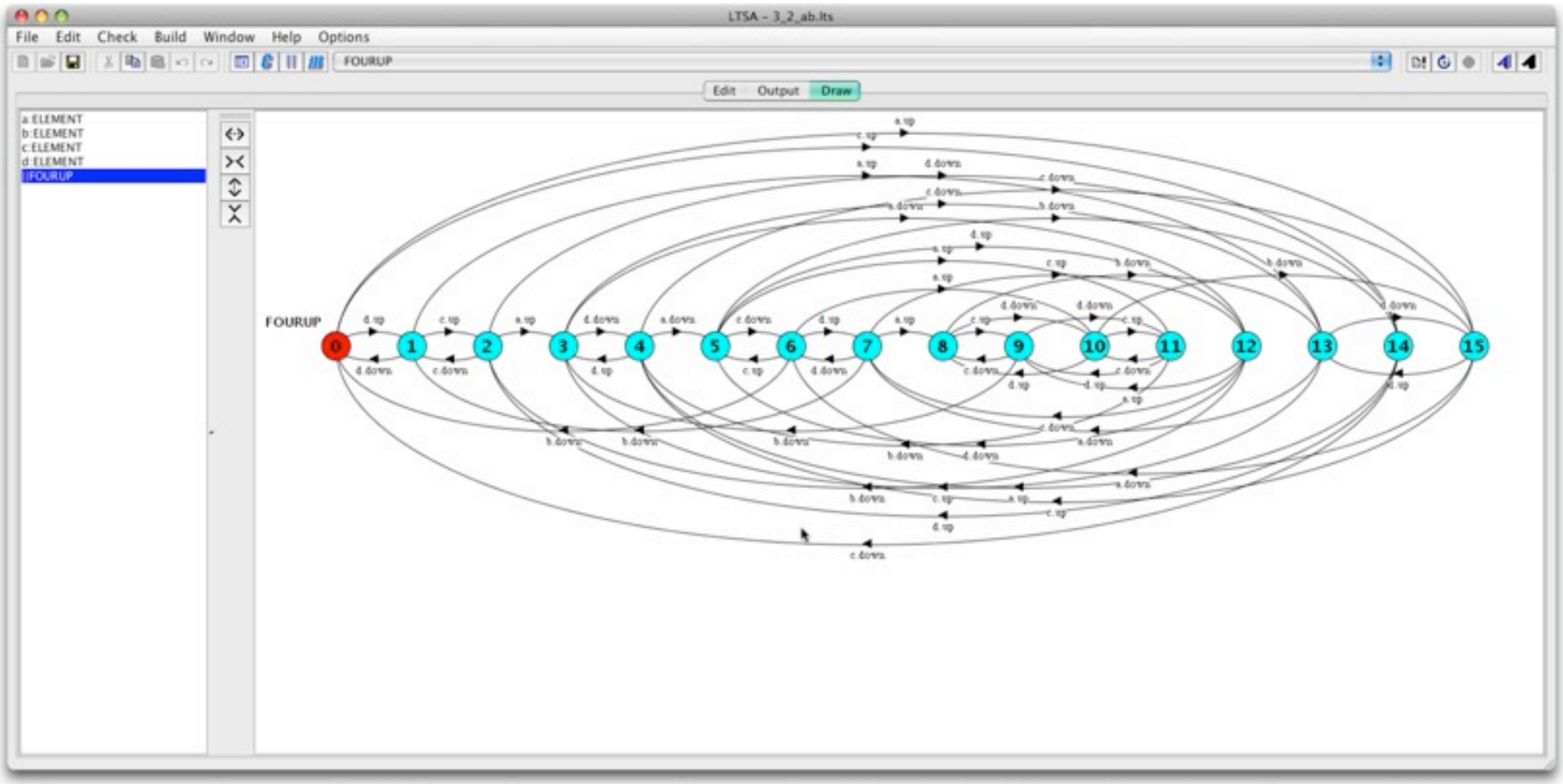
Approach

- The approach we are going to take to solve this problem is to tie the up and down actions of the various ELEMENT instances such that one process cannot "go up" unless another process "goes down"
  - To do this, lets start by tying a's down action with b's up action.

```
ELEMENT = (up -> down -> ELEMENT).

||FOURUP = (a: ELEMENT || b: ELEMENT || c: ELEMENT || d: ELEMENT)
/{a.down/b.up}
```

- The results?



A slightly less complicated LTS since a.down and b.up still have to occur at the same time.

Lets do the same thing with {b, c} and {c, d}. That is have b's down be shared with c's up, etc.

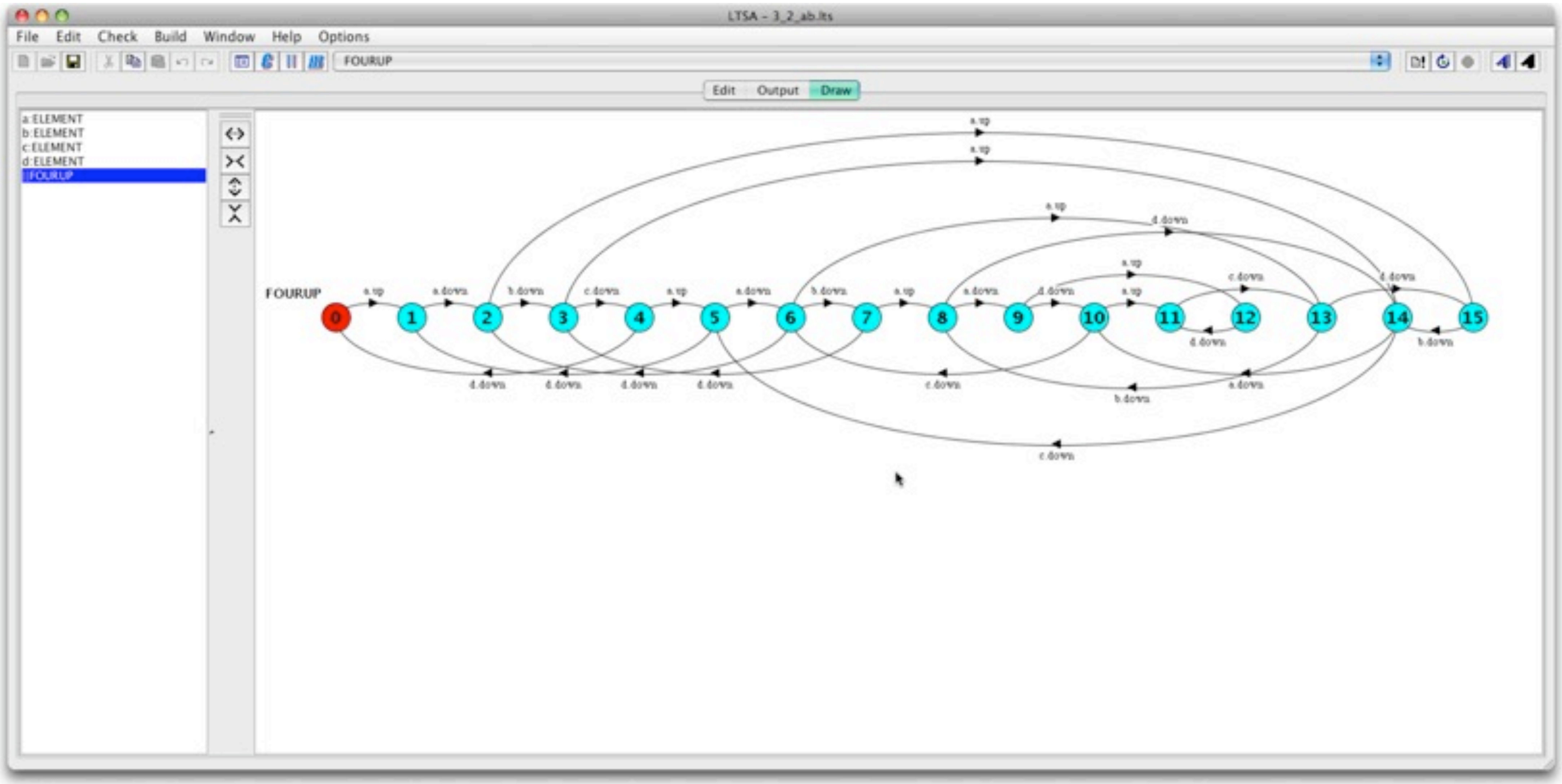
Sharing more actions

- New FSP

```
ELEMENT = (up -> down -> ELEMENT).

||FOURUP = (a: ELEMENT || b: ELEMENT || c: ELEMENT ||
d: ELEMENT) /{a.down/b.up, b.down/c.up, c.down/d.up}
```

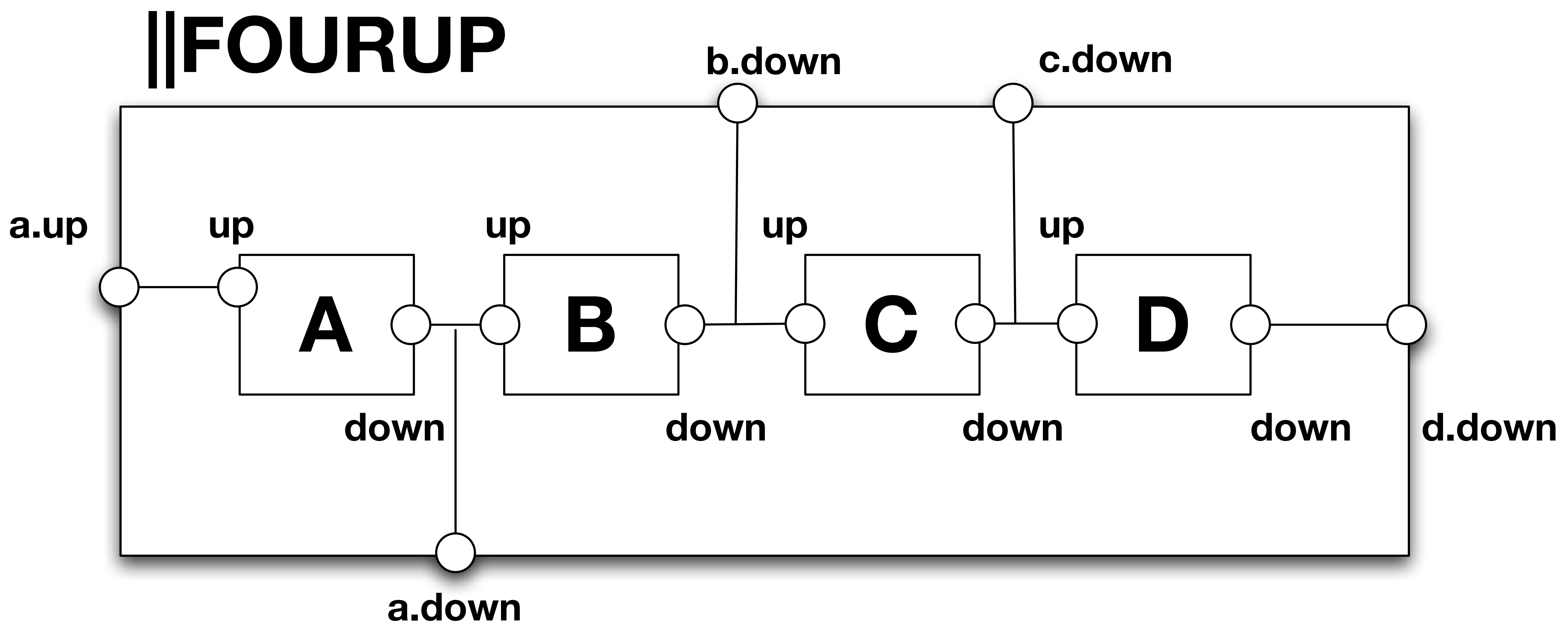
- Leads to a much simpler set of behaviors (next slide)



Now we are seeing a major reduction in complexity. Because b cannot go up until a comes down, c cannot go up until b comes down, etc.

To see, load the spec and run the animation. Recall that underneath this composition, we still have four simple processes.

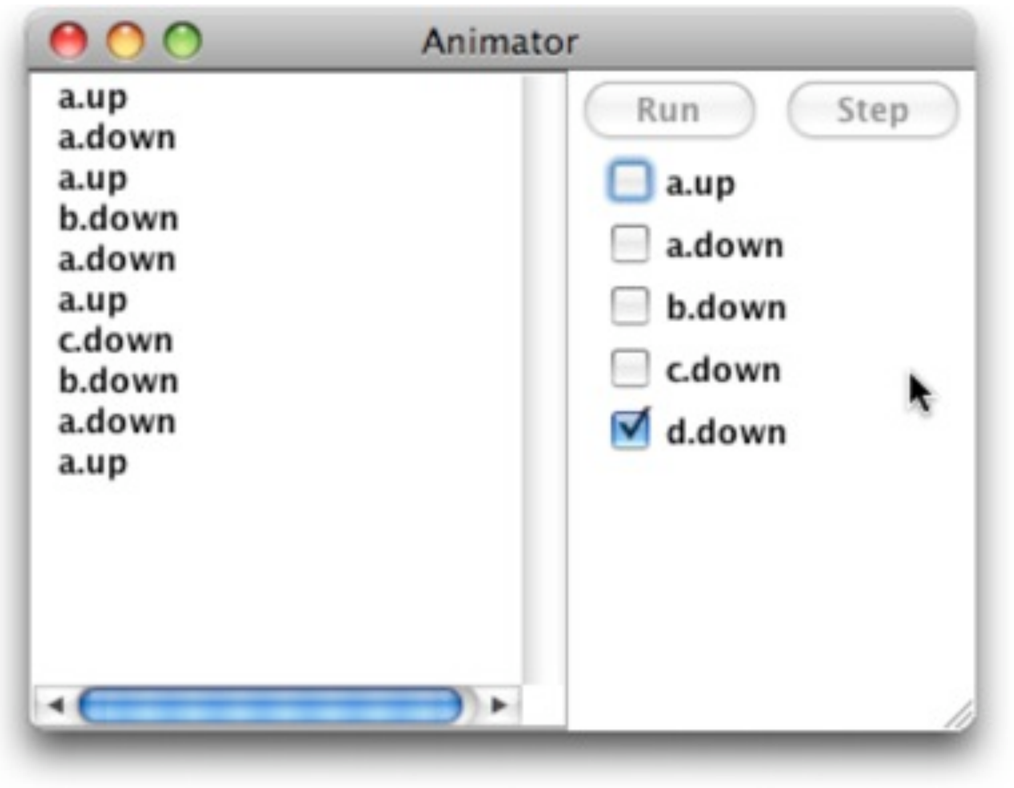
Structure Diagram?



Now we can see why the complexity has gone down. The alphabet of ||FOURUP has been reduced from 8 actions to 5

Now the magic happens…

- Run the animation very carefully and note that a can perform its up operation four times before d is forced to run its down operation



```
a.up
a.down
a.up
b.down
a.down
a.up
c.down
b.down
a.down
a.up
```

☐ a.up
☐ a.down
☐ b.down
☐ c.down
☑ d.down

But that's exactly the behavior we are looking for! Four "up" actions before a "down" action occurs

So...

Up and Down

- We are almost finished; our second to last step is to rename “a.up” to “up” and rename “d.down” to “down”

```
ELEMENT = (up -> down -> ELEMENT).

||FOURUP = (a: ELEMENT || b: ELEMENT || c: ELEMENT
|| d: ELEMENT) /{up/a.up, a.down/b.up, b.down/c.up,
c.down/d.up, down/d.down}.
```

- We get the same LTS as the last time but now in the animation we see the actions “up” and “down”. But, we have all these other actions getting in the way. What to do?

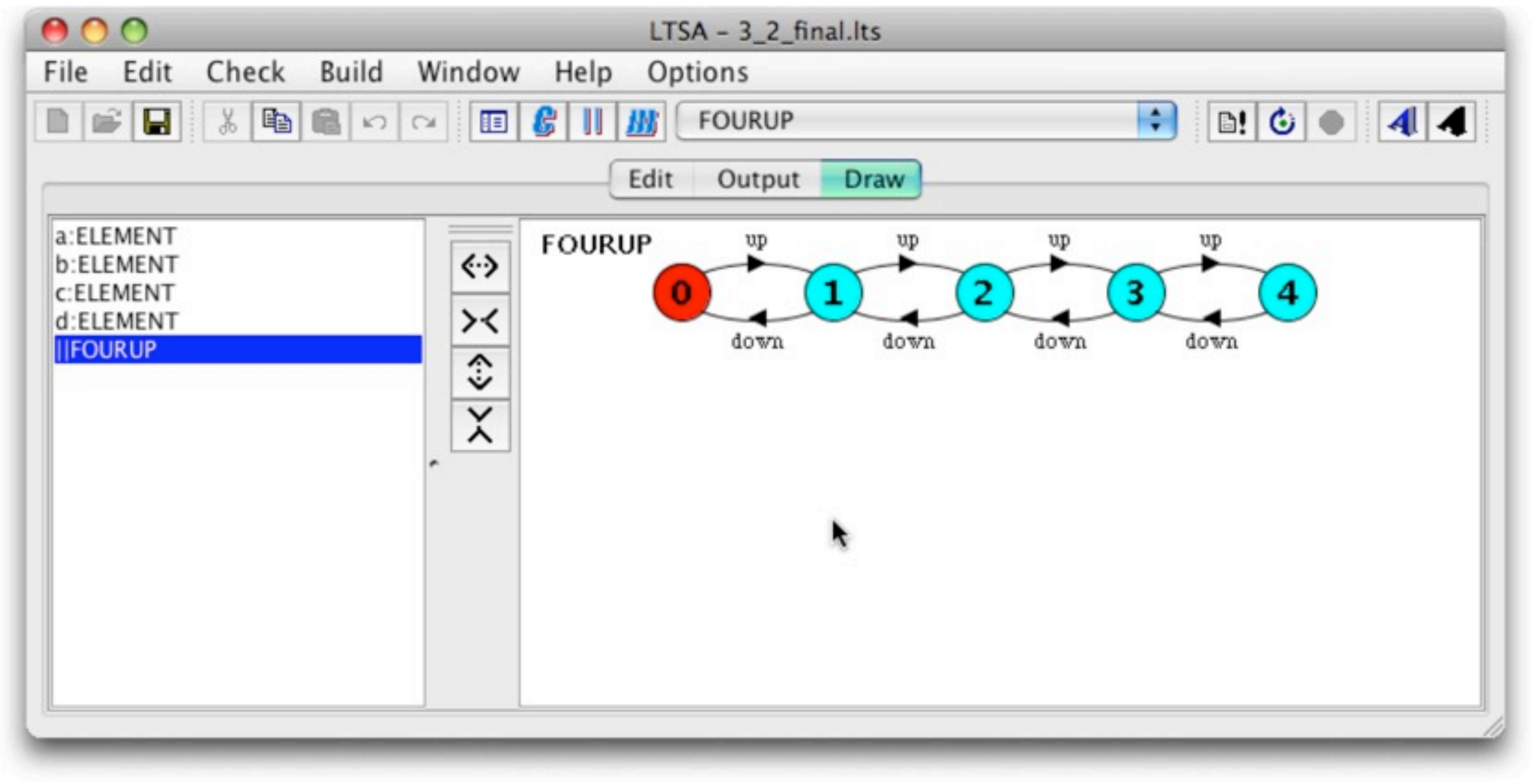
Time to Hide

- The last thing we need to do is hide all of the actions except “up” and “down”; Our composed process will simply have an alphabet of size 2

```
ELEMENT = (up -> down -> ELEMENT).

||FOURUP = (a: ELEMENT || b: ELEMENT || c: ELEMENT
|| d: ELEMENT) /{up/a.up, a.down/b.up, b.down/c.up,
c.down/d.up, down/d.down} @ {up,down}.
```

- This produces the same LTS with a lot of “tau” actions
  - So, minimize that graph and you get…

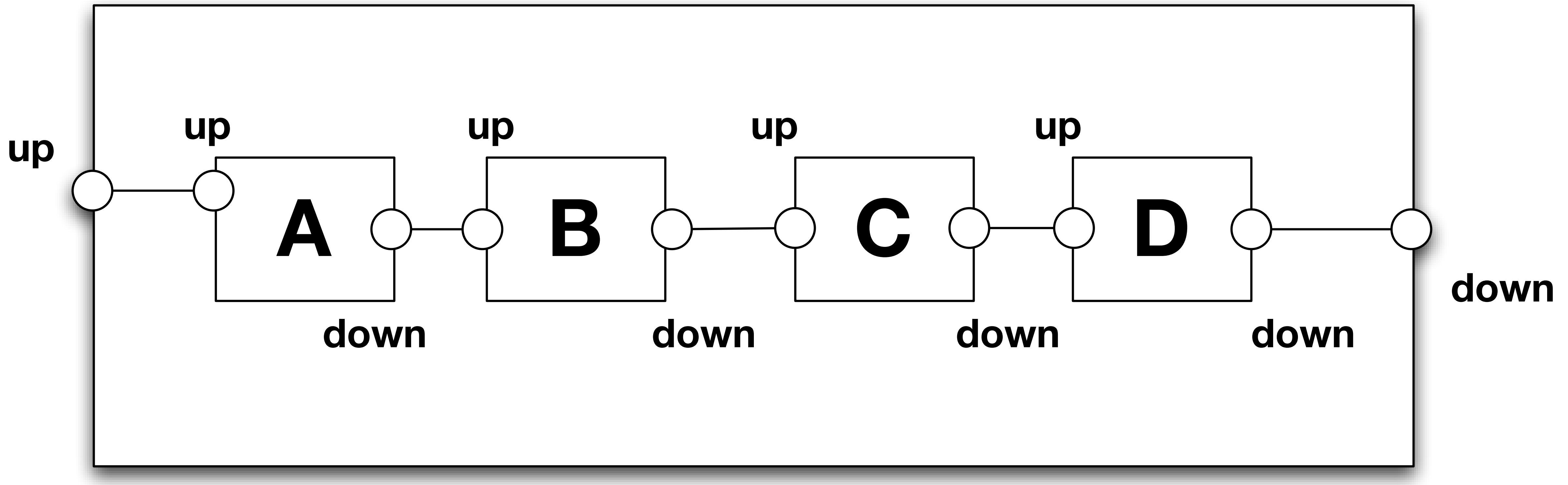


The exact same LTS that we achieved without using parallel composition!

Problem solved.

Final Structure Diagram

||FOURUP



The final process just has "up" and "down" as its alphabet; we depend on the fact that a.up can run four times before "d.down" has to be executed to achieve the desired behavior.

Coming Up

- Lecture 12: Monitors and Condition Synchronization
  - Chapter 5 of Magee and Kramer
- Lecture 13: Version Control
  - Chapter 6 of Pilone & Miles

- Lecture 14 will be a review for the Midterm
  - Chapters 1-6 of Pilone & Miles
  - Chapters 1-5 of Magee and Kramer