

Precise and Verifiable Distributed and Concurrent System Design using TLA+

Jay Parlar
[@parlar](https://twitter.com/parlar)
[#tla-workshop](https://github.com/tla-workshop)

1

Special Thanks

- Laura Santamaria
- Ben Meyer
- Landon Jurgens
- Josh Schairbaum
- Rahman Syed
- Jordan Griege
- Freddy Knuth

2

Workshop, NOT a lecture
PLEASE Ask Questions

3-1

Workshop, NOT a lecture
PLEASE Ask Questions

But please save philosophical pondering until after the workshop

3-2

Useful Resources

4-1

Useful Resources

- TLA Toolbox 1.5.3: <https://rax.io/tlaplus>
 - <https://github.com/tlaplus/tlaplus/releases/tag/v1.5.3>

4-2

Useful Resources

- TLA Toolbox 1.5.3: <https://rax.io/tlaplus>
 - <https://github.com/tlaplus/tlaplus/releases/tag/v1.5.3>
- Workshop Github Repo https://rax.io/tla_workshop
 - https://github.com/parlarjb/tla_workshop

4-3

Useful Resources

- TLA Toolbox 1.5.3: <https://rax.io/tlaplus>
 - <https://github.com/tlaplus/tlaplus/releases/tag/v1.5.3>
- Workshop Github Repo https://rax.io/tla_workshop
 - https://github.com/parlarjb/tla_workshop
- TLA Home Page: https://rax.io/tla_home
 - <http://lamport.azurewebsites.net/tla/tla.html>

4-4

Useful Resources

- TLA Toolbox 1.5.3: <https://rax.io/tlaplus>
 - <https://github.com/tlaplus/tlaplus/releases/tag/v1.5.3>
- Workshop Github Repo https://rax.io/tla_workshop
 - https://github.com/parlarjb/tla_workshop
- TLA Home Page: https://rax.io/tla_home
 - <http://lamport.azurewebsites.net/tla/tla.html>
- TLA Video Series: https://rax.io/tla_videos
 - <http://lamport.azurewebsites.net/video/videos.html>

4-5

Additional Resources

- <https://learntla.com/introduction/>
- <http://lamport.azurewebsites.net/tla/hyperbook.html>
- <https://www.hillelwayne.com/post/modeling-deployments/>
- <http://lamport.azurewebsites.net/tla/summary.pdf>
- <http://lamport.azurewebsites.net/tla/book.html>
- <https://lorinhochstein.wordpress.com/2014/06/04/crossing-the-river-with-tla/>

5

Goals

Goals

- Introduce the concepts of TLA+

6-1

6-2

Goals

- Introduce the concepts of TLA+
- Introduce a good amount of the syntax of TLA+

6-3

Goals

- Introduce the concepts of TLA+
- Introduce a good amount of the syntax of TLA+
- Introduce the model checker, TLC

6-4

Goals

- Introduce the concepts of TLA+
- Introduce a good amount of the syntax of TLA+
- Introduce the model checker, TLC
- Provide a foundation on which you can continue learning

6-5

Goals

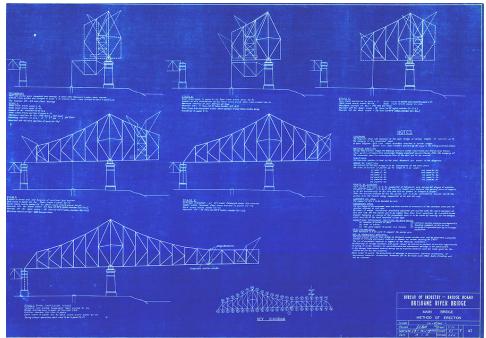
- Introduce the concepts of TLA+
- Introduce a good amount of the syntax of TLA+
- Introduce the model checker, TLC
- Provide a foundation on which you can continue learning
- Convince everyone that thinking **early** is a good thing

6-6

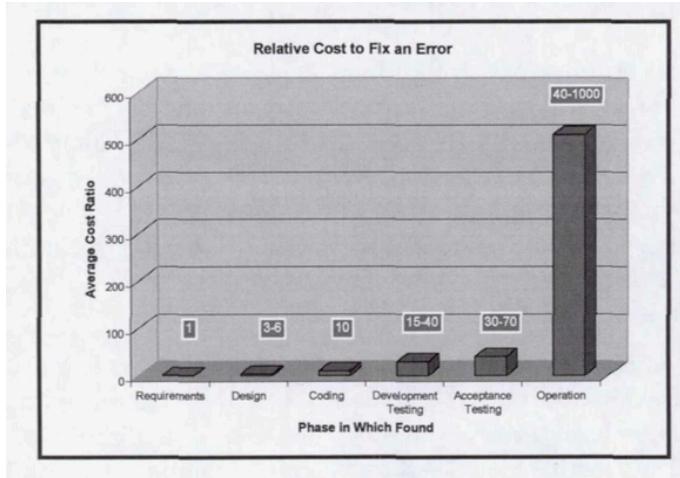
Goals

- Introduce the concepts of TLA+
- Introduce a good amount of the syntax of TLA+
- Introduce the model checker, TLC
- Provide a foundation on which you can continue learning
- Convince everyone that thinking **early** is a good thing
- World domination

6-7



8



"Error Cost Escalation Through the Project Life Cycle"
<https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20100036670.pdf>

7

```
algorithm ford-fulkerson is
    input: Graph  $G$  with flow capacity  $c$ ,
           source node  $s$ ,
           sink node  $t$ 
    output: Flow  $f$  such that  $f$  is maximal from  $s$  to  $t$ 

    (Note that  $f_{(u,v)}$  is the flow from node  $u$  to node  $v$ , and  $c_{(u,v)}$  is the flow capacity from node  $u$  to node  $v$ )

    for each edge  $(u, v)$  in  $G_E$  do
         $f_{(u, v)} \leftarrow 0$ 
         $f_{(v, u)} \leftarrow 0$ 

    while there exists a path  $p$  from  $s$  to  $t$  in the residual network  $G_f$  do
        let  $c_f$  be the flow capacity of the residual network  $G_f$ 
         $c_f(p) \leftarrow \min\{c_f(u, v) \mid (u, v) \text{ in } p\}$ 
        for each edge  $(u, v)$  in  $p$  do
             $f_{(u, v)} \leftarrow f_{(u, v)} + c_f(p)$ 
             $f_{(v, u)} \leftarrow -f_{(u, v)}$ 

    return  $f$ 
```

<https://en.wikipedia.org/wiki/Pseudocode>

9

“Writing is nature’s way of letting you know how sloppy your thinking is.”

- Dick Guindon

10-1

set the default status to "fail"
look up the message based on the error code
if the error code is valid
if doing interactive processing, display the error message
interactively and declare success
if doing command line processing, log the error message to the
command line and declare success
if the error code isn't valid, notify the user that an
internal error has been detected
return status information

<https://blog.codinghorror.com/pseudocode-or-code/>

“Writing is nature’s way of letting you know how sloppy your thinking is.”

- Dick Guindon

**“Writing a specification helps you think clearly.
Thinking clearly is hard; we can use all the help we
can get. Making specification part of the design
process can improve design.”**

- Leslie Lamport

10-2

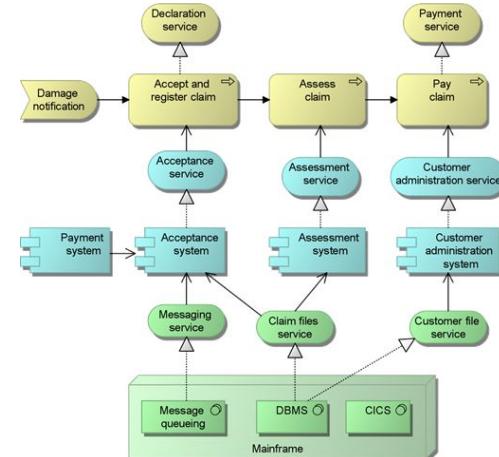
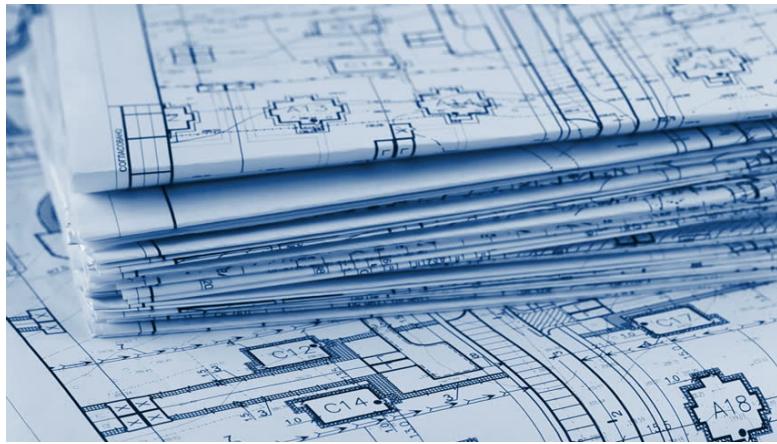


Image via <https://en.wikipedia.org/wiki/Archimate>

11

12



13

Dr. Leslie Lamport

- Turing Award winner
- Winner of first ever “Dijkstra Prize in Distributed Computing” award
- Byzantine Generals’ Problem
- Paxos (which powers Big Table, Google File System, etc)
- Bakery Algorithm
- LaTeX
- PlusCal / TLA+



14



15-1



15-2

Amazon, Microsoft, etc.

System	Components	Benefit
S3	- Fault-tolerant low-level network algorithm - Background redistribution of data	Multiple bugs found in design and optimization phases
DynamoDB	Replication and group-membership systems	Found multiple design bugs, some requiring 35-step traces
EC2	Fault tolerant replication improvement, including zero-downtime deployment	Found design bug
Xbox 360	Memory Interface	Found a bug that would hard-lock the Xbox after four hours

16

"At Amazon, many engineers at all levels of experience have been able to learn TLA+ from scratch and get useful results in 2 to 3 weeks, in some cases just in their personal time on weekends and evenings, and without help or training."

- Chris Newcombe (former AWS Principal Engineer)

17-1

"At Amazon, many engineers at all levels of experience have been able to learn TLA+ from scratch and get useful results in 2 to 3 weeks, in some cases just in their personal time on weekends and evenings, and without help or training."

"Executive management are now proactively encouraging teams to write TLA+ specs for new features and other significant design changes. In annual planning, managers are now allocating engineering time to use TLA+."

- Chris Newcombe (former AWS Principal Engineer)

"At Amazon, many engineers at all levels of experience have been able to learn TLA+ from scratch and get useful results in 2 to 3 weeks, in some cases just in their personal time on weekends and evenings, and without help or training."

"Executive management are now proactively encouraging teams to write TLA+ specs for new features and other significant design changes. In annual planning, managers are now allocating engineering time to use TLA+."

"TLA+ is the most valuable thing that I've learned in my professional career. It has changed how I work by giving me an immensely powerful tool to find subtle flaws in system designs. It has changed how I think..."

- Chris Newcombe (former AWS Principal Engineer)

17-2

17-3

TLA+

18-1

TLA+

- Precise language for modelling systems, abstracting away unimportant details.

18-2

TLA+

- Precise language for modelling systems, abstracting away unimportant details.
- **System:** A single algorithm, a single process, multiple threads, multiple processes, servers communicating over a network, network protocols, etc. Anything digital.

18-3

TLA+

- Precise language for modelling systems, abstracting away unimportant details.
- **System:** A single algorithm, a single process, multiple threads, multiple processes, servers communicating over a network, network protocols, etc. Anything digital.
- Every digital system can be modelled as a set of variables and actions.

18-4

TLA+

- Precise language for modelling systems, abstracting away unimportant details.
- **System:** A single algorithm, a single process, multiple threads, multiple processes, servers communicating over a network, network protocols, etc. Anything digital.
- Every digital system can be modelled as a set of variables and actions.
- “Model checker” discovers **every single possible state** of the described system, looks for “unwanted” states.

18-5

TLA+

- Precise language for modelling systems, abstracting away unimportant details.
- **System:** A single algorithm, a single process, multiple threads, multiple processes, servers communicating over a network, network protocols, etc. Anything digital.
- Every digital system can be modelled as a set of variables and actions.
- “Model checker” discovers **every single possible state** of the described system, looks for “unwanted” states.
- The language is called **TLA+**; the model checker tool is called **TLC**.

18-6

Raft

- A used-heavily-in-industry consensus algorithm designed with TLA+.

19-1

19-2

Raft

- A used-heavily-in-industry consensus algorithm designed with TLA+.
- Used by **etcd** (and thus crucial to Kubernetes) and other systems.

19-3

Raft

- A used-heavily-in-industry consensus algorithm designed with TLA+.
- Used by **etcd** (and thus crucial to Kubernetes) and other systems.
 - **etcd** is a distributed key:value store, where every server must agree on all the key:value pairs.

19-4

Raft

- A used-heavily-in-industry consensus algorithm designed with TLA+.
- Used by **etcd** (and thus crucial to Kubernetes) and other systems.
 - **etcd** is a distributed key:value store, where every server must agree on all the key:value pairs.
- ~500 lines of heavily-commented TLA+.

19-5

Raft

- A used-heavily-in-industry consensus algorithm designed with TLA+.
- Used by **etcd** (and thus crucial to Kubernetes) and other systems.
 - **etcd** is a distributed key:value store, where every server must agree on all the key:value pairs.
- ~500 lines of heavily-commented TLA+.
- <https://raft.github.io>

19-6

Raft

```
Next ==  $\wedge \forall \forall \exists i \in \text{Server} : \text{Restart}(i)$ 
       $\vee \forall \exists i \in \text{Server} : \text{Timeout}(i)$ 
       $\vee \forall \exists i, j \in \text{Server} : \text{RequestVote}(i, j)$ 
       $\vee \forall \exists i \in \text{Server} : \text{BecomeLeader}(i)$ 
       $\vee \forall \exists i \in \text{Server} : \text{ClientRequest}(i)$ 
       $\vee \forall \exists i \in \text{Server} : \text{AdvanceCommitIndex}(i)$ 
       $\vee \forall \exists i, j \in \text{Server} : \text{AppendEntries}(i, j)$ 
       $\vee \forall \exists m \in \text{ValidMessage(messages)} : \text{Receive}(m)$ 
       $\vee \forall \exists m \in \text{SingleMessage(messages)} : \text{DuplicateMessage}(m)$ 
       $\vee \forall \exists m \in \text{ValidMessage(messages)} : \text{DropMessage}(m)$ 
 $\wedge \text{allLogs}' = \text{allLogs} \cup \{ \log[i] : i \in \text{Server} \}$ 
```

20-1

Raft

```
Next ==  $\wedge \forall \forall \exists i \in \text{Server} : \text{Restart}(i)$ 
       $\vee \forall \exists i \in \text{Server} : \text{Timeout}(i)$ 
       $\vee \forall \exists i, j \in \text{Server} : \text{RequestVote}(i, j)$ 
       $\vee \forall \exists i \in \text{Server} : \text{BecomeLeader}(i)$ 
       $\vee \forall \exists i \in \text{Server} : \text{ClientRequest}(i)$ 
       $\vee \forall \exists i \in \text{Server} : \text{AdvanceCommitIndex}(i)$ 
       $\vee \forall \exists i, j \in \text{Server} : \text{AppendEntries}(i, j)$ 
 $\vee \forall \exists m \in \text{ValidMessage(messages)} : \text{Receive}(m)$  (highlighted)
       $\vee \forall \exists m \in \text{SingleMessage(messages)} : \text{DuplicateMessage}(m)$ 
       $\vee \forall \exists m \in \text{ValidMessage(messages)} : \text{DropMessage}(m)$ 
 $\wedge \text{allLogs}' = \text{allLogs} \cup \{ \log[i] : i \in \text{Server} \}$ 
```

20-2

Raft

```
Next ==  $\wedge \forall \forall \exists i \in \text{Server} : \text{Restart}(i)$ 
       $\vee \forall \exists i \in \text{Server} : \text{Timeout}(i)$ 
       $\vee \forall \exists i, j \in \text{Server} : \text{RequestVote}(i, j)$ 
       $\vee \forall \exists i \in \text{Server} : \text{BecomeLeader}(i)$ 
       $\vee \forall \exists i \in \text{Server} : \text{ClientRequest}(i)$ 
       $\vee \forall \exists i \in \text{Server} : \text{AdvanceCommitIndex}(i)$ 
       $\vee \forall \exists i, j \in \text{Server} : \text{AppendEntries}(i, j)$ 
 $\vee \forall \exists m \in \text{ValidMessage(messages)} : \text{Receive}(m)$  (highlighted)
 $\vee \forall \exists m \in \text{SingleMessage(messages)} : \text{DuplicateMessage}(m)$  (highlighted)
 $\vee \forall \exists m \in \text{ValidMessage(messages)} : \text{DropMessage}(m)$  (highlighted)
 $\wedge \text{allLogs}' = \text{allLogs} \cup \{ \log[i] : i \in \text{Server} \}$ 
```

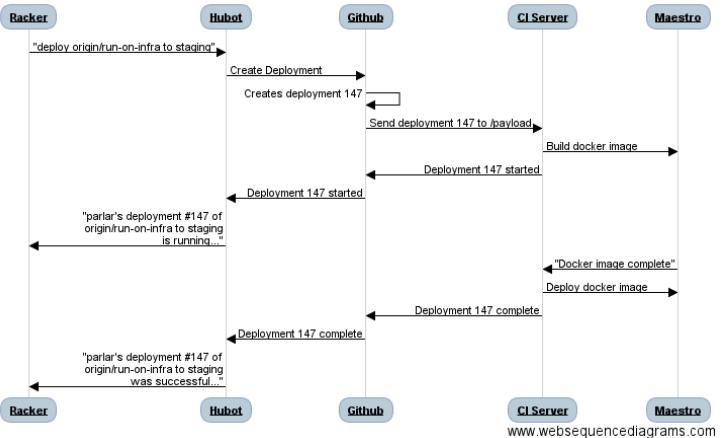
20-3

Raft

```
Next ==  $\wedge \forall \forall \exists i \in \text{Server} : \text{Restart}(i)$ 
       $\vee \forall \exists i \in \text{Server} : \text{Timeout}(i)$ 
       $\vee \forall \exists i, j \in \text{Server} : \text{RequestVote}(i, j)$ 
       $\vee \forall \exists i \in \text{Server} : \text{BecomeLeader}(i)$ 
       $\vee \forall \exists i \in \text{Server} : \text{ClientRequest}(i)$ 
       $\vee \forall \exists i \in \text{Server} : \text{AdvanceCommitIndex}(i)$ 
       $\vee \forall \exists i, j \in \text{Server} : \text{AppendEntries}(i, j)$ 
 $\vee \forall \exists m \in \text{ValidMessage(messages)} : \text{Receive}(m)$  (highlighted)
 $\vee \forall \exists m \in \text{SingleMessage(messages)} : \text{DuplicateMessage}(m)$  (highlighted)
 $\vee \forall \exists m \in \text{ValidMessage(messages)} : \text{DropMessage}(m)$  (highlighted)
 $\wedge \text{allLogs}' = \text{allLogs} \cup \{ \log[i] : i \in \text{Server} \}$ 
```

20-4

Encore Origin Deployments



21

```

182 /* This represents CI receiving a message from Maestro, when Maestro
183 /* has finished trying to build an image
184 HandleImageBuildDone(m) ==
185   \& \& m.type == PRBuildComplete
186   \& CASE m.status == "success" -> UpdatePR(m.sha, "success", m)
187   [] m.status == "error" -> UpdatePR(m.sha, "error", m)
188   [] m.status == "failure" -> UpdatePR(m.sha, "failure", m)
189   \& m.type == DeployBuildComplete
190   \& CASE m.status == "success" -> MaestroDeploy(m.sha, m.id, m )
191   [] m.status \& ("error", "failure") -> UpdateDeployStatus(m.sha, m.id, "failure", m)
192
193 /* UNCHANGED <>next_id, registry, commit_status, maestro_deploys_complete, gh_deploys_complete>>
194
195 /* Represents CI receiving a NewDeployment message from GH
196 HandleNewDeploy(m) ==
197   \& m.type == NewDeployment
198   \& \& m.sha \& registry
199   \& MaestroDeploy(m.sha, m.id, m) /* We already have the image, so tell Maestro to go ahead and
200   \& \& m.sha \&notin registry
201   \& BuildDeployImage(m.sha, m.id, m) /* We don't have the image, so tell Maestro to build it
202   \& UNCHANGED <>registry, next_id, commit_status, maestro_deploys_complete, gh_deploys_complete>>
203
204
205 /* Represents CI receiving a deployment success/failure message from Maestro. Only "success" and "fail
206 HandleDeployComplete(m) ==
207   \& m.type == DeployImageComplete
208   \& \& m.status == "success"
209   \& UpdateDeployStatus(m.sha, m.id, "success", m)
210   \& \& m.status == "failure"
211   \& UpdateDeployStatus(m.sha, m.id, "failure", m)
212
213 /* UNCHANGED <>registry, next_id, commit_status, maestro_deploys_complete, gh_deploys_complete>>
214
215 /* Represents GH receiving a commit_status status update from CI
216 HandlePRStatusUpdate(m) ==
217   \& m.type == PRStatus
218   \& commit_status' == [commit_status EXCEPT !{m.sha} = m.status]
219   \& Discard(m)
220
221 
```

22

Deployment Pipeline

Deployment Pipeline

- “Only deploy if an image exists”

23-1

23-2

Deployment Pipeline

- “Only deploy if an image exists”
- “All deploys must have been initiated by GitHub”

23-3

Deployment Pipeline

- “Only deploy if an image exists”
- “All deploys must have been initiated by GitHub”
- “A single SHA can not have both a successful and failed PR build”

23-4

Deployment Pipeline

- “Only deploy if an image exists”
- “All deploys must have been initiated by GitHub”
- “A single SHA can not have both a successful and failed PR build”
- “Deployment status is consistent through the different systems”

23-5

Deployment Pipeline

- “Only deploy if an image exists”
- “All deploys must have been initiated by GitHub”
- “A single SHA can not have both a successful and failed PR build”
- “Deployment status is consistent through the different systems”
- etc. etc.

23-6

PlusCal

24-1

PlusCal

- A programming language-like layer on top of TLA+.

24-2

PlusCal

- A programming language-like layer on top of TLA+.
- Some people find it easier to read and write than TLA+.

24-3

PlusCal

- A programming language-like layer on top of TLA+.
- Some people find it easier to read and write than TLA+.
- TLC compiles PlusCal code into TLA+.

24-4

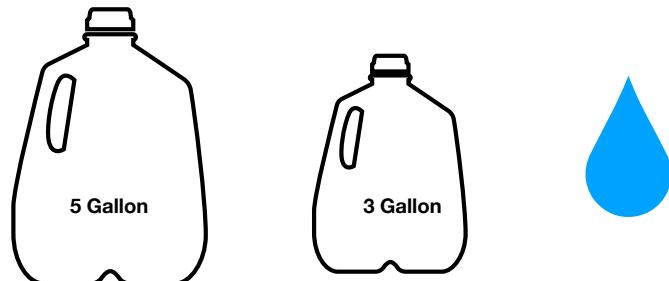
PlusCal

- A programming language-like layer on top of TLA+.
- Some people find it easier to read and write than TLA+.
- TLC compiles PlusCal code into TLA+.
- Hard to learn by starting with PlusCal. Better to gain a foundation in TLA+ then take advantage of PlusCal where appropriate.

24-5

How do we measure out exactly 4 gallons?

i.e. “The Die Hard Problem”



25

One Possible Execution

small: 0
big: 0

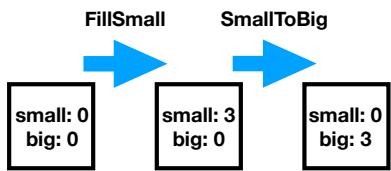
26-1

One Possible Execution

FillSmall
→
small: 0
big: 0 small: 3
 big: 0

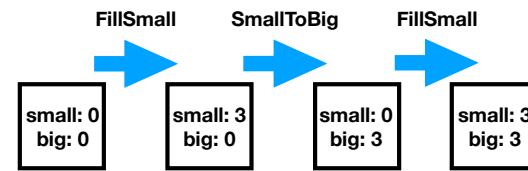
26-2

One Possible Execution



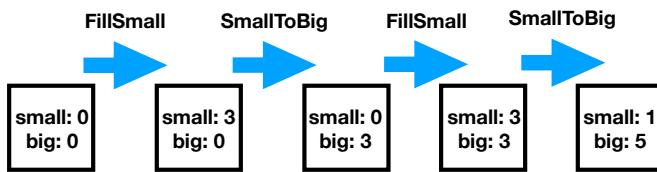
26-3

One Possible Execution



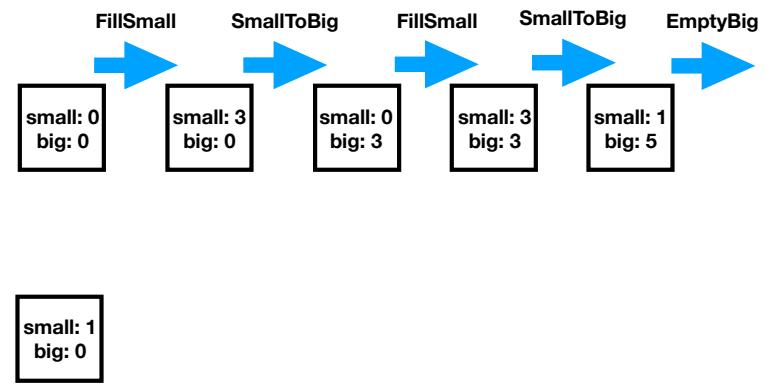
26-4

One Possible Execution



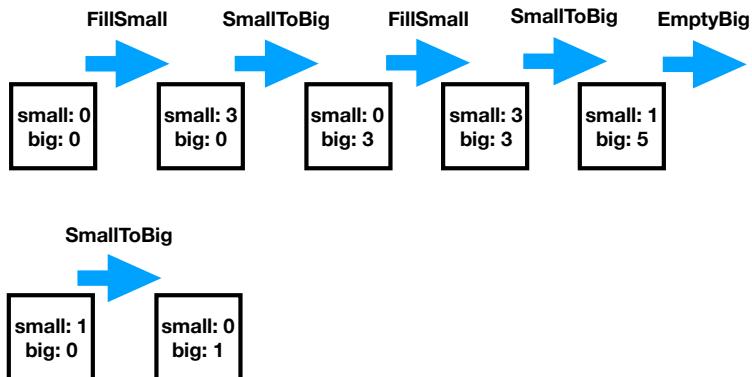
26-5

One Possible Execution



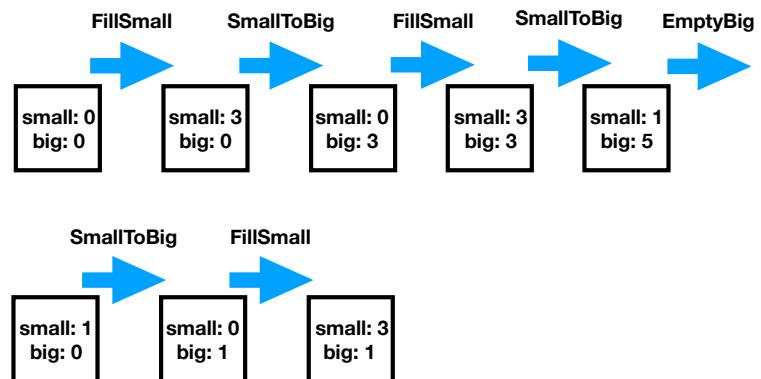
26-6

One Possible Execution



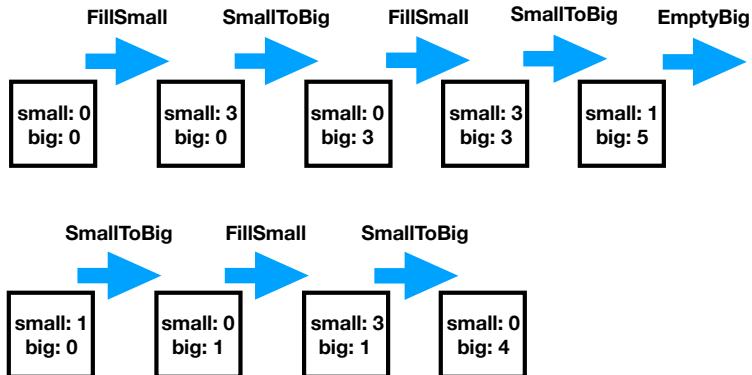
26-7

One Possible Execution



26-8

One Possible Execution



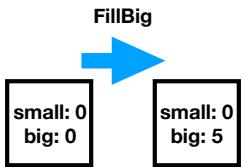
26-9

Another Possible Execution



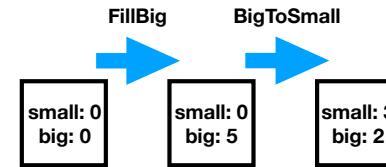
27-1

Another Possible Execution



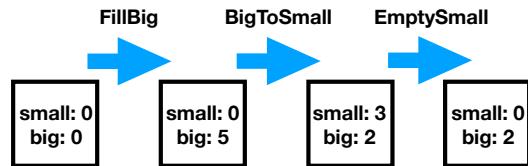
27-2

Another Possible Execution



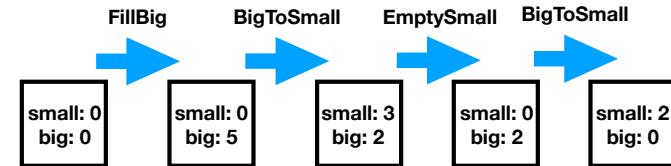
27-3

Another Possible Execution



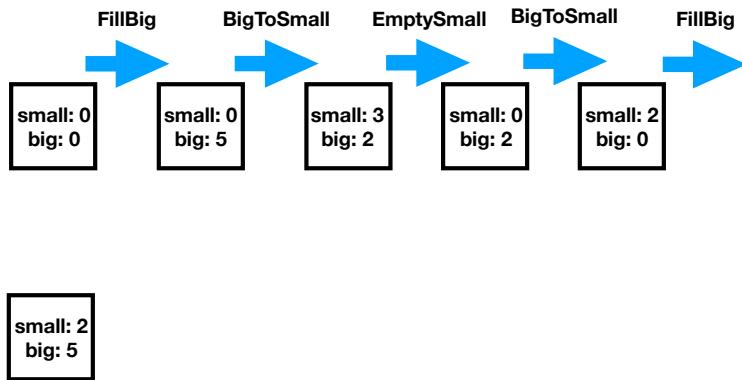
27-4

Another Possible Execution



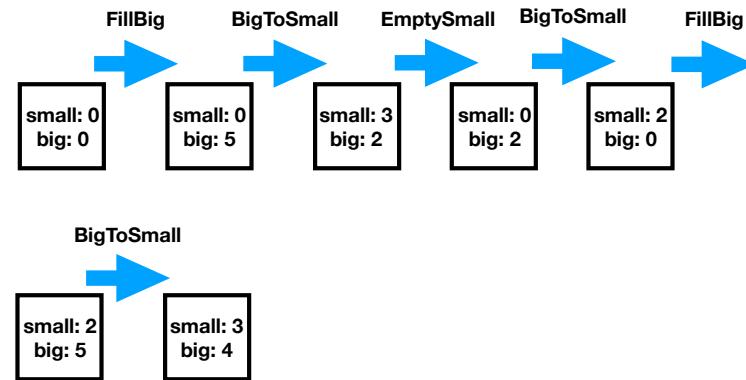
27-5

Another Possible Execution



27-6

Another Possible Execution



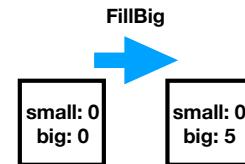
27-7

Yet Another Possible Execution



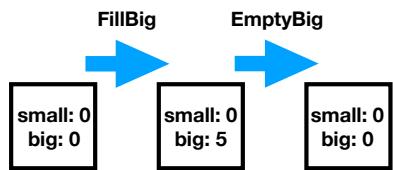
28-1

Yet Another Possible Execution



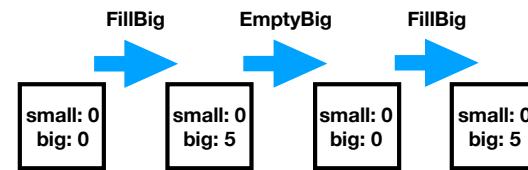
28-2

Yet Another Possible Execution



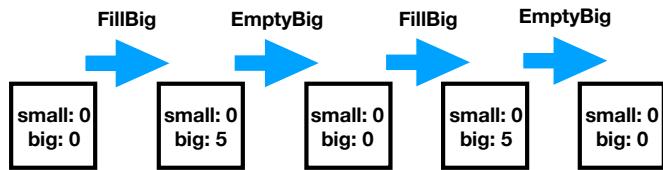
28-3

Yet Another Possible Execution



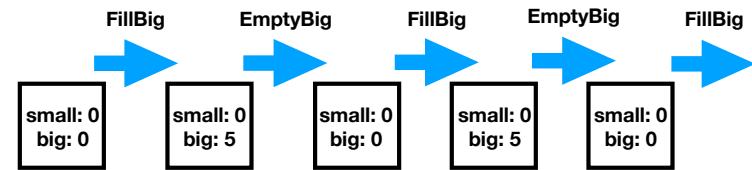
28-4

Yet Another Possible Execution

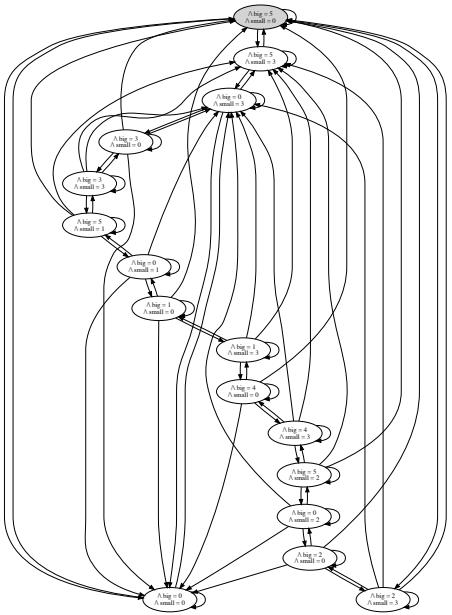


28-5

Yet Another Possible Execution



28-6



29

TLA+ Terms

- A single one of those executions is called a **behaviour**.

TLA+ Terms

30-1

TLA+ Terms

- A single one of those executions is called a **behaviour**.
- A behaviour is a **sequence of states**.

30-2

30-3

TLA+ Terms

- A single one of those executions is called a **behaviour**.
- A behaviour is a **sequence of states**.
- A **step** is the change from one state to the next.

30-4

TLA+ Terms

- A single one of those executions is called a **behaviour**.
- A behaviour is a **sequence of states**.
- A **step** is the change from one state to the next.
- A **state** is an assignment of **values** to **variables**.

30-5

TLA+ Terms

- A single one of those executions is called a **behaviour**.
- A behaviour is a **sequence of states**.
- A **step** is the change from one state to the next.
- A **state** is an assignment of **values** to **variables**.
- TLA+ is all about describing **all possible executions/ behaviours** of a system.

30-6

How could we describe digital systems (behaviours)?

- Programming languages
- Turing machines
- Automata
- Hardware Description Languages

31

We'll instead use State Machines!

A state machine is defined by three things:

32-1

We'll instead use State Machines!

A state machine is defined by three things:

1. A listing of all of our **variables**.
2. All the possible **initial states**.

32-3

We'll instead use State Machines!

A state machine is defined by three things:

1. A listing of all of our **variables**.

32-2

We'll instead use State Machines!

A state machine is defined by three things:

1. A listing of all of our **variables**.
2. All the possible **initial states**.
3. The possible **next states** for any given state (i.e., state transitions), or, the **relationship** between the values of the **variables** in the current state, and their possible values in the **next state**.

32-4

Die Hard State Machine

33-1

Die Hard State Machine

1. Variables: **big** and **small**

33-2

Die Hard State Machine

1. Variables: **big** and **small**

2. Possible initial values: **big=0, small=0**

33-3

Die Hard State Machine

1. Variables: **big** and **small**

2. Possible initial values: **big=0, small=0**

3. Possible next states: ?

33-4

Die Hard Next States

34-1

Die Hard Next States

1.We can always fill **small**, i.e., in the next state **small=3**

34-2

Die Hard Next States

1.We can always fill **small**, i.e., in the next state **small=3**

2.We can always fill **big**, i.e., in the next state **big = 5**

34-3

Die Hard Next States

1.We can always fill **small**, i.e., in the next state **small=3**

2.We can always fill **big**, i.e., in the next state **big = 5**

3.We can always empty **small**, i.e., in the next state **small=0**

34-4

Die Hard Next States

1. We can always fill **small**, i.e., in the next state **small=3**
2. We can always fill **big**, i.e., in the next state **big = 5**
3. We can always empty **small**, i.e., in the next state **small=0**
4. We can always empty **big**, i.e., in the next state **big=0**

34-5

Die Hard Next States

1. We can always fill **small**, i.e., in the next state **small=3**
2. We can always fill **big**, i.e., in the next state **big = 5**
3. We can always empty **small**, i.e., in the next state **small=0**
4. We can always empty **big**, i.e., in the next state **big=0**
5. We can always pour small into big, i.e., in the next state:

34-6

Die Hard Next States

1. We can always fill **small**, i.e., in the next state **small=3**
2. We can always fill **big**, i.e., in the next state **big = 5**
3. We can always empty **small**, i.e., in the next state **small=0**
4. We can always empty **big**, i.e., in the next state **big=0**
5. We can always pour small into big, i.e., in the next state:

if (**big + small**) <= 5 THEN **big=big+small; small=0**

34-7

Die Hard Next States

1. We can always fill **small**, i.e., in the next state **small=3**
2. We can always fill **big**, i.e., in the next state **big = 5**
3. We can always empty **small**, i.e., in the next state **small=0**
4. We can always empty **big**, i.e., in the next state **big=0**
5. We can always pour small into big, i.e., in the next state:

if (**big + small**) <= 5 THEN **big=big+small; small=0**

ELSE **big = 5; small = small - (5 - big)**

34-8

Die Hard Next States

1. We can always fill **small**, i.e., in the next state **small=3**
2. We can always fill **big**, i.e., in the next state **big = 5**
3. We can always empty **small**, i.e., in the next state **small=0**
4. We can always empty **big**, i.e., in the next state **big=0**
5. We can always pour small into big, i.e., in the next state:

```
if (big + small) <= 5 THEN big=big+small; small=0  
ELSE big = 5; small = small - (5 - big)
```

34-9

Die Hard Next States

1. We can always fill **small**, i.e., in the next state **small=3**
2. We can always fill **big**, i.e., in the next state **big = 5**
3. We can always empty **small**, i.e., in the next state **small=0**
4. We can always empty **big**, i.e., in the next state **big=0**
5. We can always pour small into big, i.e., in the next state:

```
if (big + small) <= 5 THEN big=big+small; small=0  
ELSE big = 5; small = small - (5 - big)
```

etc. etc.

34-10

Possible next states

small: 3
big: 2

35-1

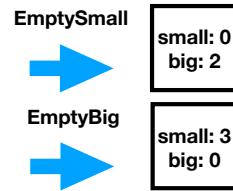
Possible next states

EmptySmall
→ small: 0
big: 2

small: 3
big: 2

35-2

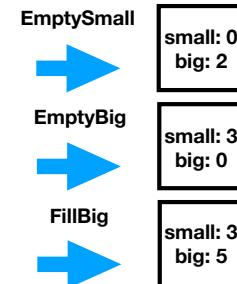
Possible next states



small: 3
big: 2

35-3

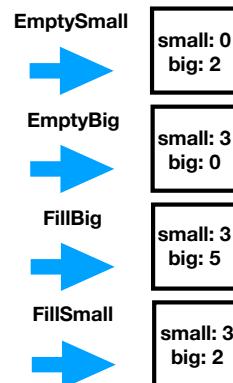
Possible next states



small: 3
big: 2

35-4

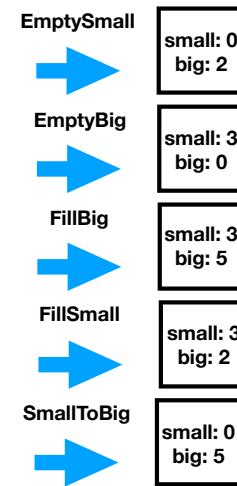
Possible next states



small: 3
big: 2

35-5

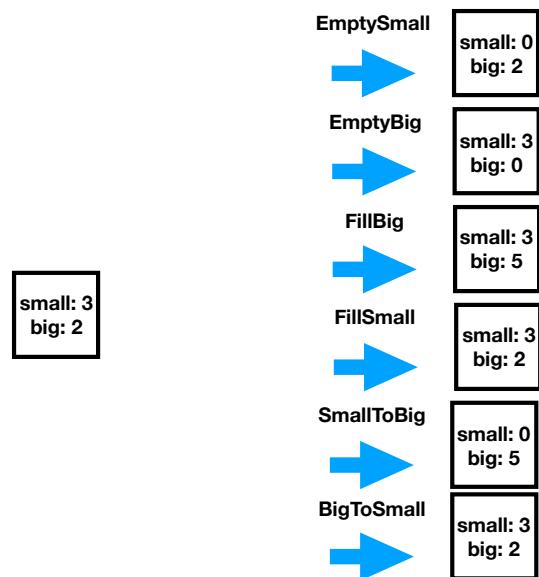
Possible next states



small: 3
big: 2

35-6

Possible next states



35-7

VARIABLES small, big

```
TypeOK == ∧ small ∈ 0..3
    ∧ big   ∈ 0..5

Init == ∧ big  = 5
    ∧ small = 0

FillSmall == ∧ small' = 3
    ∧ big'  = big

FillBig == ∧ big'  = 5
    ∧ small' = small

EmptySmall == ∧ small' = 0
    ∧ big'  = big

EmptyBig == ∧ big'  = 0
    ∧ small' = small

SmallToBig == IF big + small <= 5
    THEN ∧ big'  = big + small
        ∧ small' = 0
    ELSE ∧ big'  = 5
        ∧ small' = small - (5 - big)

BigToSmall == IF big + small = 3
    THEN ∧ big'  = 0
        ∧ small' = big + small
    ELSE ∧ big'  = small - (3 - big)
        ∧ small' = 3

Next == ∨ FillSmall ∨ FillBig
    ∨ EmptySmall ∨ EmptyBig
    ∨ SmallToBig ∨ BigToSmall
```

36

Why use state machines?

Why use state machines?

- State machines are vastly simpler than programming languages. All parts of the state are represented as values of variables

37-1

37-2

Why use state machines?

- State machines are vastly simpler than programming languages. All parts of the state are represented as values of variables
- Programming languages represent different parts of the state differently:

37-3

Why use state machines?

- State machines are vastly simpler than programming languages. All parts of the state are represented as values of variables
- Programming languages represent different parts of the state differently:
 - Values of variables

37-4

Why use state machines?

- State machines are vastly simpler than programming languages. All parts of the state are represented as values of variables
- Programming languages represent different parts of the state differently:
 - Values of variables
 - Program counter

37-5

Why use state machines?

- State machines are vastly simpler than programming languages. All parts of the state are represented as values of variables
- Programming languages represent different parts of the state differently:
 - Values of variables
 - Program counter
 - Call stack

37-6

Why use state machines?

- State machines are vastly simpler than programming languages. All parts of the state are represented as values of variables
- Programming languages represent different parts of the state differently:
 - Values of variables
 - Program counter
 - Call stack
 - Heap

37-7

Why use state machines?

- State machines are vastly simpler than programming languages. All parts of the state are represented as values of variables
- Programming languages represent different parts of the state differently:
 - Values of variables
 - Program counter
 - Call stack
 - Heap
 - etc.

37-8

Why use state machines?

- State machines are vastly simpler than programming languages. All parts of the state are represented as values of variables
- Programming languages represent different parts of the state differently:
 - Values of variables
 - Program counter
 - Call stack
 - Heap
 - etc.
- Programming languages are **terrible** at representing non-determinism.

37-9

Math Time!!!

38

“and” / “or”

39-1

“and” / “or”

Λ Logical “AND”. Called a “conjunct”. Written in ASCII as /\
Written in Python as `and`. Written in C as `&&`

```
TRUE /\ TRUE = TRUE  
TRUE /\ FALSE = FALSE  
FALSE /\ FALSE = FALSE  
FALSE /\ TRUE = FALSE
```

∨ Logical “OR”. Called a “disjunct”. Written in ASCII as \/
Written in Python as `or`. Written in C as `||`

```
TRUE \/ TRUE = TRUE  
TRUE \/ FALSE = TRUE  
FALSE \/ FALSE = FALSE  
FALSE \/ TRUE = TRUE
```

39-3

“and” / “or”

Λ Logical “AND”. Called a “conjunct”. Written in ASCII as /\
Written in Python as `and`. Written in C as `&&`

```
TRUE /\ TRUE = TRUE  
TRUE /\ FALSE = FALSE  
FALSE /\ FALSE = FALSE  
FALSE /\ TRUE = FALSE
```

39-2

= **and** \triangleq

40-1

= and \triangleq

- = means “equality”. It is NOT an assignment operator. It is a boolean operator. It is the = you would have learned in grade 1 mathematics. It’s equivalent to Python’s ==.

40-2

= and \triangleq

- = means “equality”. It is NOT an assignment operator. It is a boolean operator. It is the = you would have learned in grade 1 mathematics. It’s equivalent to Python’s ==.

- \triangleq means “defined to be”. It is written as == in ASCII. You use it to created named definitions of things.

40-3

= and \triangleq

- = means “equality”. It is NOT an assignment operator. It is a boolean operator. It is the = you would have learned in grade 1 mathematics. It’s equivalent to Python’s ==.

- \triangleq means “defined to be”. It is written as == in ASCII. You use it to created named definitions of things.

MyDef == **A** / \backslash **B**

At any time, **MyDef** will be the value of **A** and’ed with **B**

It is a little like using a macro, or an inline.

40-4

Exercise

TRUE / \backslash (**TRUE** / \backslash **FALSE**)

TRUE / \backslash (**FALSE** / \backslash (**TRUE** / \backslash **FALSE**))

FALSE / \backslash (**TRUE** / \backslash ((**FALSE** / \backslash **TRUE**) / \backslash (**TRUE** / \backslash **FALSE**)))

41-1

Exercise

TRUE $\vee\wedge$ (TRUE $\wedge\backslash$ FALSE)

TRUE

TRUE $\wedge\backslash$ (FALSE $\wedge\backslash$ (TRUE $\vee\wedge$ FALSE))

FALSE $\wedge\backslash$ (TRUE $\wedge\backslash$ ((FALSE $\vee\wedge$ TRUE) $\wedge\backslash$ (TRUE $\vee\wedge$ FALSE)))

41-2

Exercise

TRUE $\vee\wedge$ (TRUE $\wedge\backslash$ FALSE)

TRUE

TRUE $\wedge\backslash$ (FALSE $\wedge\backslash$ (TRUE $\vee\wedge$ FALSE))

TRUE

FALSE $\wedge\backslash$ (TRUE $\wedge\backslash$ ((FALSE $\vee\wedge$ TRUE) $\wedge\backslash$ (TRUE $\vee\wedge$ FALSE)))

41-3

Exercise

TRUE $\vee\wedge$ (TRUE $\wedge\backslash$ FALSE)

TRUE

TRUE $\wedge\backslash$ (FALSE $\wedge\backslash$ (TRUE $\vee\wedge$ FALSE))

TRUE

FALSE $\wedge\backslash$ (TRUE $\wedge\backslash$ ((FALSE $\vee\wedge$ TRUE) $\wedge\backslash$ (TRUE $\vee\wedge$ FALSE)))

FALSE

41-4

TLA + Syntax

TLA+ specs make heavy use of formulas like this

A $\wedge\backslash$ (B $\wedge\backslash$ ((C $\vee\wedge$ D) $\wedge\backslash$ (E $\wedge\backslash$ F))) $\wedge\backslash$ (G $\vee\wedge$ H $\wedge\backslash$ I)

42-1

TLA + Syntax

TLA+ specs make heavy use of formulas like this

```
A /\ (B /\ ((C /\ D) \/(E /\ F))) /\ (G /\ H /\ I)
```

This sucks. TLA requires lots of Boolean logic, and these brackets are not fun. TLA+ has a better way to do it.

42-2

TLA + Syntax

TLA+ specs make heavy use of formulas like this

```
A /\ (B /\ ((C /\ D) \/(E /\ F))) /\ (G /\ H /\ I)
```

This sucks. TLA requires lots of Boolean logic, and these brackets are not fun. TLA+ has a better way to do it.

```
/\ A  
/\ (B /\ ((C /\ D) \/(E /\ F)))  
/\ (G /\ H /\ I)
```

Indentation level determines which variables are AND'ed and OR'ed together
It's like using a bulleted-list of \wedge or \vee symbols

42-4

TLA + Syntax

TLA+ specs make heavy use of formulas like this

```
A /\ (B /\ ((C /\ D) \/(E /\ F))) /\ (G /\ H /\ I)
```

This sucks. TLA requires lots of Boolean logic, and these brackets are not fun. TLA+ has a better way to do it.

```
/\ A  
/\ (B /\ ((C /\ D) \/(E /\ F)))  
/\ (G /\ H /\ I)
```

42-3

TLA + Syntax

TLA+ specs make heavy use of formulas like this

```
A /\ (B /\ ((C /\ D) \/(E /\ F))) /\ (G /\ H /\ I)
```

This sucks. TLA requires lots of Boolean logic, and these brackets are not fun. TLA+ has a better way to do it.

```
/\ A  
/\ (B /\ ((C /\ D) \/(E /\ F)))  
/\ (G /\ H /\ I)
```

Indentation level determines which variables are AND'ed and OR'ed together
It's like using a bulleted-list of \wedge or \vee symbols

42-5

TLA + Syntax

TLA+ specs make heavy use of formulas like this
$$A \wedge (B \wedge ((C \vee D) \vee (E \vee F))) \wedge (G \vee H \vee I)$$

This sucks. TLA requires lots of Boolean logic, and these brackets are not fun. TLA+ has a better way to do it.

$$\begin{array}{l} \wedge A \\ \wedge (B \wedge ((C \vee D) \vee (E \vee F))) \\ \wedge (G \vee H \vee I) \end{array}$$

Indentation level determines which variables are AND'ed and OR'ed together
It's like using a bulleted-list of \wedge or \vee symbols

42-6

TLA + Syntax

TLA+ specs make heavy use of formulas like this
$$A \wedge (B \wedge ((C \vee D) \vee (E \vee F))) \wedge (G \vee H \vee I)$$

This sucks. TLA requires lots of Boolean logic, and these brackets are not fun. TLA+ has a better way to do it.

$$\begin{array}{l} \wedge A \\ \wedge (B \wedge ((C \vee D) \vee (E \vee F))) \\ \wedge (G \vee H \vee I) \end{array}$$

Indentation level determines which variables are AND'ed and OR'ed together
It's like using a bulleted-list of \wedge or \vee symbols

42-8

TLA + Syntax

TLA+ specs make heavy use of formulas like this
$$A \wedge (B \wedge ((C \vee D) \vee (E \vee F))) \wedge (G \vee H \vee I)$$

This sucks. TLA requires lots of Boolean logic, and these brackets are not fun. TLA+ has a better way to do it.

$$\begin{array}{l} \wedge A \\ \wedge (B \wedge ((C \vee D) \vee (E \vee F))) \\ \wedge (G \vee H \vee I) \end{array}$$

Indentation level determines which variables are AND'ed and OR'ed together
It's like using a bulleted-list of \wedge or \vee symbols

42-7

TLA + Syntax

TLA+ specs make heavy use of formulas like this
$$A \wedge (B \wedge ((C \vee D) \vee (E \vee F))) \wedge (G \vee H \vee I)$$

This sucks. TLA requires lots of Boolean logic, and these brackets are not fun. TLA+ has a better way to do it.

$$\begin{array}{l} \wedge A \\ \wedge (B \wedge ((C \vee D) \vee (E \vee F))) \\ \wedge (G \vee H \vee I) \end{array}$$

Indentation level determines which variables are AND'ed and OR'ed together
It's like using a bulleted-list of \wedge or \vee symbols

42-9

TLA + Syntax

TLA+ specs make heavy use of formulas like this

```
A /\ (B /\ ((C /\ D) \/(E /\ F))) /\ (G /\ H /\ I)
```

This sucks. TLA requires lots of Boolean logic, and these brackets are not fun. TLA+ has a better way to do it.

```
/\ A  
/\ (B /\ ((C /\ D) \/(E /\ F)))  
/\ (G /\ H /\ I)
```

Indentation level determines which variables are AND'ed and OR'ed together
It's like using a bulleted-list of \wedge or \vee symbols

42-10

TLA + Syntax

TLA+ specs make heavy use of formulas like this

```
A /\ (B /\ ((C /\ D) \/(E /\ F))) /\ (G /\ H /\ I)
```

This sucks. TLA requires lots of Boolean logic, and these brackets are not fun. TLA+ has a better way to do it.

```
/\ A  
/\ (B /\ ((C /\ D) \/(E /\ F)))  
/\ (G /\ H /\ I)
```

Indentation level determines which variables are AND'ed and OR'ed together
It's like using a bulleted-list of \wedge or \vee symbols

42-12

TLA + Syntax

TLA+ specs make heavy use of formulas like this

```
A /\ (B /\ ((C /\ D) \/(E /\ F))) /\ (G /\ H /\ I)
```

This sucks. TLA requires lots of Boolean logic, and these brackets are not fun. TLA+ has a better way to do it.

```
/\ A  
/\ (B /\ ((C /\ D) \/(E /\ F)))  
/\ (G /\ H /\ I)
```

Indentation level determines which variables are AND'ed and OR'ed together
It's like using a bulleted-list of \wedge or \vee symbols

42-11

TLA+ Syntax

In general:

43-1

TLA+ Syntax

In general:

$$\begin{array}{c} /\! \ A \\ /\! \ B \\ /\! \ C \\ \dots \\ /\! \ N \end{array}$$
$$A /\! \ B /\! \ C /\! \ \dots /\! \ N =$$

43-2

TLA+ Syntax

In general:

$$\begin{array}{c} /\! \ A \\ /\! \ B \\ /\! \ C \\ \dots \\ /\! \ N \end{array}$$
$$A /\! \ B /\! \ C /\! \ \dots /\! \ N =$$

$$\begin{array}{c} \backslash/ \ A \\ \backslash/ \ B \\ \backslash/ \ C \\ \dots \\ \backslash/ \ N \end{array}$$
$$A \backslash/ \ B \backslash/ \ C \backslash/ \ \dots \backslash/ \ N =$$

43-3

TLA+ Syntax

In general:

$$\begin{array}{c} /\! \ A \\ /\! \ B \\ /\! \ C \\ \dots \\ /\! \ N \end{array}$$
$$A /\! \ B /\! \ C /\! \ \dots /\! \ N =$$

$$\begin{array}{c} \backslash/ \ A \\ \backslash/ \ B \\ \backslash/ \ C \\ \dots \\ \backslash/ \ N \end{array}$$
$$A \backslash/ \ B \backslash/ \ C \backslash/ \ \dots \backslash/ \ N =$$

43-4

TLA+ Syntax

In general:

$$\begin{array}{c} /\! \ A \\ /\! \ B \\ /\! \ C \\ \dots \\ /\! \ N \end{array}$$
$$A /\! \ B /\! \ C /\! \ \dots /\! \ N =$$

$$\begin{array}{c} \backslash/ \ A \\ \backslash/ \ B \\ \backslash/ \ C \\ \dots \\ \backslash/ \ N \end{array}$$
$$A \backslash/ \ B \backslash/ \ C \backslash/ \ \dots \backslash/ \ N =$$

43-5

TLA+ Syntax

In general:

$$\boxed{A \wedge B \wedge C \wedge \dots \wedge N} = \begin{array}{l} \wedge A \\ \wedge B \\ \wedge C \\ \dots \\ \wedge N \end{array}$$

$$A \vee B \vee C \vee \dots \vee N = \begin{array}{l} \vee A \\ \vee B \\ \vee C \\ \dots \\ \vee N \end{array}$$

43-6

TLA+ Syntax

In general:

$$\boxed{A \wedge B \wedge C \wedge \dots \wedge N} = \begin{array}{l} \wedge A \\ \wedge B \\ \wedge C \\ \dots \\ \wedge N \end{array}$$

$$\boxed{A \vee B \vee C \vee \dots \vee N} = \begin{array}{l} \vee A \\ \vee B \\ \vee C \\ \dots \\ \vee N \end{array}$$

43-7

TLA+ Syntax

TLA+ does not allow “ambiguous” formulas:

$A \wedge B \vee C \wedge D$

44-1

TLA+ Syntax

TLA+ does not allow “ambiguous” formulas:

$A \wedge B \vee C \wedge D$

Mathematically, there is a correct answer, but people have a hard time remembering.
TLA+ forces you to be explicit.

44-2

TLA+ Syntax

TLA+ does not allow “ambiguous” formulas:

A /\ B \/ C /\ D

Mathematically, there is a correct answer, but people have a hard time remembering.
TLA+ forces you to be explicit.

(A /\ B) \/ (C /\ D)

44-3

TLA+ Syntax

TLA+ does not allow “ambiguous” formulas:

A /\ B \/ C /\ D

Mathematically, there is a correct answer, but people have a hard time remembering.
TLA+ forces you to be explicit.

(A /\ B) \/ (C /\ D)

(We'll come back to this.)

44-4

TLA+ Syntax

- The indentation level is meaningful.
- Expressions on the same indent level are “and”ed and “or”ed together.

A /\ (B \/ C \/ D) /\ E /\ (F /\ G)

45-1

TLA+ Syntax

- The indentation level is meaningful.
- Expressions on the same indent level are “and”ed and “or”ed together.

A /\ (B \/ C \/ D) /\ E /\ (F /\ G)

- This formula has four items at the top level, being AND’ed together. Let’s rewrite using TLA+ syntax.

45-2

TLA+ Syntax

- The indentation level is meaningful.
- Expressions on the same indent level are “and”ed and “or”ed together.

```
A /\ (B \/ C \/ D) /\ E /\ (F /\ G)
```

- This formula has four items at the top level, being AND’ed together. Let’s rewrite using TLA+ syntax.

45-3

TLA+ Syntax

- The indentation level is meaningful.
- Expressions on the same indent level are “and”ed and “or”ed together.

```
A /\ (B \/ C \/ D) /\ E /\ (F /\ G)
```

- This formula has four items at the top level, being AND’ed together. Let’s rewrite using TLA+ syntax.

45-4

TLA+ Syntax

- The indentation level is meaningful.
- Expressions on the same indent level are “and”ed and “or”ed together.

```
A /\ (B \/ C \/ D) /\ E /\ (F /\ G)
```

- This formula has four items at the top level, being AND’ed together. Let’s rewrite using TLA+ syntax.

45-5

TLA+ Syntax

- The indentation level is meaningful.
- Expressions on the same indent level are “and”ed and “or”ed together.

```
A /\ (B \/ C \/ D) /\ E /\ (F /\ G)
```

- This formula has four items at the top level, being AND’ed together. Let’s rewrite using TLA+ syntax.

45-6

TLA+ Syntax

- The indentation level is meaningful.
- Expressions on the same indent level are “and”ed and “or”ed together.

`A /\ (B \ / C \ / D) /\ E /\ (F \ / G)`

- This formula has four items at the top level, being AND’ed together. Let’s rewrite using TLA+ syntax.
- We’ll put \wedge at the outermost indent level.

45-7

TLA+ Syntax

`A /\ (B \ / C \ / D) /\ E /\ (F \ / G)`

46-1

TLA+ Syntax

`A /\ (B \ / C \ / D) /\ E /\ (F \ / G)`

$/\ A$

46-2

TLA+ Syntax

`A /\ (B \ / C \ / D) /\ E /\ (F \ / G)`

$/\ A$
 $/\ (B \ / C \ / D)$

46-3

TLA+ Syntax

A /\ B \/\ C \/\ D) /\ E /\ (F \/\ G)

/\ A
/\ (B \/\ C \/\ D)
/\ E

46-4

TLA+ Syntax

A /\ B \/\ C \/\ D) /\ E /\ (F \/\ G)

/\ A
/\ (B \/\ C \/\ D)
/\ E
/\ (F \/\ G)

46-5

TLA+ Syntax

A /\ B \/\ C \/\ D) /\ E /\ (F \/\ G)

/\ A
/\ (B \/\ C \/\ D)
/\ E
/\ (F \/\ G)

Now we can easily see all the “outermost” or “top-level” items being AND’ed together

46-6

TLA+ Syntax

A /\ B \/\ C \/\ D) /\ E /\ (F \/\ G)

/\ A
/\ (B \/\ C \/\ D)
/\ E
/\ (F \/\ G)

Now we can easily see all the “outermost” or “top-level” items being AND’ed together

46-7

TLA+ Syntax

`A /\ (B \ / C \ / D) /\ E /\ (F \ / G)`

$\begin{array}{c} /\ A \\ /\ (B \ / C \ / D) \\ /\ E \\ /\ (F \ / G) \end{array}$

Now we can easily see all the “outermost” or “top-level” items being AND’ed together

46-8

TLA+ Syntax

`A /\ (B \ / C \ / D) /\ E /\ (F \ / G)`

$\begin{array}{c} /\ A \\ /\ (B \ / C \ / D) \\ /\ E \\ /\ (F \ / G) \end{array}$

Now we can easily see all the “outermost” or “top-level” items being AND’ed together

46-10

TLA+ Syntax

`A /\ (B \ / C \ / D) /\ E /\ (F \ / G)`

$\begin{array}{c} /\ A \\ /\ (B \ / C \ / D) \\ /\ E \\ /\ (F \ / G) \end{array}$

Now we can easily see all the “outermost” or “top-level” items being AND’ed together

46-9

TLA+ Syntax

`A /\ (B \ / C \ / D) /\ E /\ (F \ / G)`

$\begin{array}{c} /\ A \\ /\ (B \ / C \ / D) \\ /\ E \\ /\ (F \ / G) \end{array}$

Now we can easily see all the “outermost” or “top-level” items being AND’ed together

We have more parentheses.

46-11

TLA+ Syntax

`A /\ (B \ / C \ / D) /\ E /\ (F / \ G)`

```
/\ A  
/\ (B \ / C \ / D)  
/\ E  
/\ (F / \ G)
```

Now we can easily see all the “outermost” or “top-level” items being AND’ed together

We have more parentheses.

Let’s get rid of those too

46-12

TLA+ Syntax

We’ll create indent levels for each of these

```
/\ A  
/\ (B \ / C \ / D)  
/\ E  
/\ (F / \ G)
```

47-1

TLA+ Syntax

We’ll create indent levels for each of these

```
/\ A  
/\ (B \ / C \ / D)  
/\ E  
/\ (F / \ G)
```

`/\ A`

47-2

TLA+ Syntax

We’ll create indent levels for each of these

```
/\ A  
/\ (B \ / C \ / D)  
/\ E  
/\ (F / \ G)
```

```
/\ A  
/\ \ / B
```

47-3

TLA+ Syntax

We'll create indent levels for each of these

```
/\ A  
/\ (B \/ C \/ D)  
/\ E  
/\ (F /\ G)
```

```
/\ A  
/\ \/ B  
 \/ C
```

47-4

TLA+ Syntax

We'll create indent levels for each of these

```
/\ A  
/\ (B \/ C \/ D)  
/\ E  
/\ (F /\ G)
```

```
/\ A  
/\ \/ B  
 \/ C  
 \/ D
```

47-5

TLA+ Syntax

We'll create indent levels for each of these

```
/\ A  
/\ (B \/ C \/ D)  
/\ E  
/\ (F /\ G)
```

```
/\ A  
/\ \/ B  
 \/ C  
 \/ D  
/\ E
```

47-6

TLA+ Syntax

We'll create indent levels for each of these

```
/\ A  
/\ (B \/ C \/ D)  
/\ E  
/\ (F /\ G)
```

```
/\ A  
/\ \/ B  
 \/ C  
 \/ D  
/\ E  
/\ /\ F
```

47-7

TLA+ Syntax

We'll create indent levels for each of these

```
/\ A  
/\ (B \ / C \ / D)  
/\ E  
/\ (F /\ G)
```

```
/\ A  
/\ /\ B  
/\ C  
/\ D  
/\ E  
/\ /\ F  
/\ G
```

47-8

TLA+ Syntax

We'll create indent levels for each of these

```
/\ A  
/\ (B \ / C \ / D)  
/\ E  
/\ (F /\ G)
```

```
/\ A  
/\ /\ B  
/\ /\ C  
/\ /\ D  
/\ E  
/\ /\ F  
/\ G
```

47-9

TLA+ Syntax

We'll create indent levels for each of these

```
/\ A  
/\ (B \ / C \ / D)  
/\ E  
/\ (F /\ G)
```

```
/\ A  
/\ /\ B  
/\ /\ C  
/\ /\ D  
/\ E  
/\ /\ F  
/\ G
```

47-10

TLA+ Syntax

We'll create indent levels for each of these

```
/\ A  
/\ (B \ / C \ / D)  
/\ E  
/\ (F /\ G)
```

```
/\ A  
/\ /\ B  
/\ /\ C  
/\ /\ D  
/\ E  
/\ /\ F  
/\ G
```

The structure of the formula should be much more clear now.
It creates an explicit graph-like visual structure out of parentheses.
Or think of it like bullet-points, if that helps .

47-11

TLA+ Syntax

What about this? How would this look?

A /\ B \/ C /\ D

48-1

TLA+ Syntax

What about this? How would this look?

A /\ B \/ C /\ D

TLA+ does not allow this. Precedence between \vee and \wedge must be made explicit with parentheses.

48-2

TLA+ Syntax

What about this? How would this look?

A /\ B \/ C /\ D

TLA+ does not allow this. Precedence between \vee and \wedge must be made explicit with parentheses.

(A /\ B) \/ (C /\ D)

48-3

TLA+ Syntax

What about this? How would this look?

A /\ B \/ C /\ D

TLA+ does not allow this. Precedence between \vee and \wedge must be made explicit with parentheses.

(A /\ B) \/ (C /\ D)

Which becomes

48-4

TLA+ Syntax

What about this? How would this look?

$A \wedge B \vee C \wedge D$

TLA+ does not allow this. Precedence between \vee and \wedge must be made explicit with parentheses.

$(A \wedge B) \vee (C \wedge D)$

Which becomes

$\vee \wedge A$
 $\wedge B$
 $\vee \wedge C$
 $\wedge D$

48-5

TLA+ Syntax

What about this? How would this look?

$A \wedge B \vee C \wedge D$

TLA+ does not allow this. Precedence between \vee and \wedge must be made explicit with parentheses.

$(A \wedge B) \vee (C \wedge D)$

Which becomes

$\vee \wedge A$
 $\wedge B$
 $\vee \wedge C$
 $\wedge D$

48-6

TLA+ Syntax

What about this? How would this look?

$A \wedge B \vee C \wedge D$

TLA+ does not allow this. Precedence between \vee and \wedge must be made explicit with parentheses.

$(A \wedge B) \vee (C \wedge D)$

Which becomes

$\vee \wedge A$
 $\wedge B$
 $\vee \wedge C$
 $\wedge D$

48-7

TLA+ Syntax

What about this? How would this look?

$A \wedge B \vee C \wedge D$

TLA+ does not allow this. Precedence between \vee and \wedge must be made explicit with parentheses.

$(A \wedge B) \vee (C \wedge D)$

Which becomes

$\vee \wedge A$
 $\wedge B$
 $\vee \wedge C$
 $\wedge D$

48-8

TLA+ Syntax

What about this? How would this look?

$A \vee B \wedge C \vee D$

TLA+ does not allow this. Precedence between \vee and \wedge must be made explicit with parentheses.

($A \vee B$) \wedge ($C \vee D$)

Which becomes

\vee \wedge \vee
 \vee \wedge \vee
 \vee \wedge \vee

48-9

TLA+ Syntax

What about this? How would this look?

$A \vee B \wedge C \vee D$

TLA+ does not allow this. Precedence between \vee and \wedge must be made explicit with parentheses.

($A \vee B$) \wedge ($C \wedge D$)

Which becomes

\vee \wedge \wedge
 \vee \wedge \wedge
 \vee \wedge \wedge

48-10

TLA+ Syntax

What about this? How would this look?

$A \vee B \wedge C \vee D$

TLA+ does not allow this. Precedence between \vee and \wedge must be made explicit with parentheses.

($A \vee B$) \wedge ($C \wedge D$)

Which becomes

\vee \wedge \wedge
 \vee \wedge \wedge
 \vee \wedge \wedge

48-11

Exercise

1. $A \vee (B \wedge (C \vee D) \vee E)$

2. $(A \vee B \wedge (C \vee D)) \wedge (E \vee F \vee G) \wedge (H \vee (I \vee J))$

3. $(A \vee (B \wedge C) \vee D) \wedge E \wedge F \wedge (G \vee H \vee (I \vee J))$

49

Solutions

1. $A \vee (B \wedge (C \vee D) \wedge E)$

$$\begin{array}{c} \vee A \\ \vee \wedge B \\ \wedge C \vee D \\ \wedge E \end{array}$$

50-1

Solutions

1. $\boxed{A} \vee (B \wedge (C \vee D) \wedge E)$

$$\begin{array}{c} \vee \boxed{A} \\ \vee \wedge B \\ \wedge C \vee D \\ \wedge E \end{array}$$

50-2

Solutions

1. $\boxed{A} \vee (B \wedge (C \vee D) \wedge E)$

$$\begin{array}{c} \vee \boxed{A} \\ \vee \wedge B \\ \wedge C \vee D \\ \wedge E \end{array}$$

50-3

Solutions

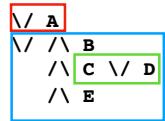
1. $\boxed{A} \vee (B \wedge \boxed{(C \vee D)} \wedge E)$

$$\begin{array}{c} \vee \boxed{A} \\ \vee \wedge \boxed{B} \\ \wedge \boxed{C \vee D} \\ \wedge E \end{array}$$

50-4

Solutions

1. $\boxed{A} \vee (B \wedge \boxed{(C \wedge D)} \wedge E)$



Alternate Solution

$\begin{array}{c} \vee A \\ \vee \vee B \\ \vee \vee C \\ \vee D \\ \wedge E \end{array}$

50-5

Solutions

2. $(A \wedge B \wedge (C \wedge D)) \wedge (E \wedge F \wedge G) \wedge (H \wedge (I \wedge J))$

$\begin{array}{c} \wedge \wedge A \\ \wedge B \\ \wedge C \wedge D \\ \wedge \vee E \\ \wedge F \\ \wedge G \\ \wedge \vee H \\ \wedge I \wedge J \end{array}$

51-1

Solutions

2. $(A \wedge B \wedge (C \wedge D)) \wedge (E \wedge F \wedge G) \wedge (H \wedge (I \wedge J))$

$\begin{array}{c} \wedge \wedge A \\ \wedge B \\ \wedge C \wedge D \\ \wedge \vee E \\ \wedge F \\ \wedge G \\ \wedge \vee H \\ \wedge I \wedge J \end{array}$

51-2

Solutions

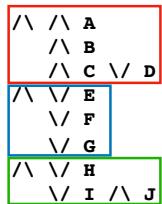
2. $(A \wedge B \wedge (C \wedge D)) \wedge \boxed{(E \wedge F \wedge G)} \wedge (H \wedge (I \wedge J))$

$\begin{array}{c} \wedge \wedge A \\ \wedge B \\ \wedge C \wedge D \\ \wedge \vee E \\ \wedge F \\ \wedge G \\ \wedge \vee H \\ \wedge I \wedge J \end{array}$

51-3

Solutions

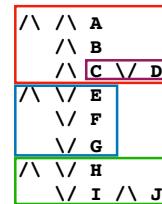
$$2. (\text{A} \wedge \text{B} \wedge (\text{C} \vee \text{D})) \wedge (\text{E} \vee \text{F} \vee \text{G}) \wedge (\text{H} \vee (\text{I} \wedge \text{J}))$$



51-4

Solutions

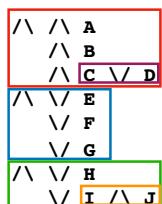
$$2. (\text{A} \wedge \text{B} \wedge (\text{C} \vee \text{D})) \wedge (\text{E} \vee \text{F} \vee \text{G}) \wedge (\text{H} \vee (\text{I} \wedge \text{J}))$$



51-5

Solutions

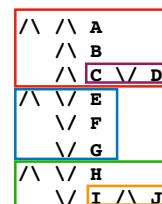
$$2. (\text{A} \wedge \text{B} \wedge (\text{C} \vee \text{D})) \wedge (\text{E} \vee \text{F} \vee \text{G}) \wedge (\text{H} \vee (\text{I} \wedge \text{J}))$$



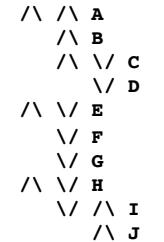
51-6

Solutions

$$2. (\text{A} \wedge \text{B} \wedge (\text{C} \vee \text{D})) \wedge (\text{E} \vee \text{F} \vee \text{G}) \wedge (\text{H} \vee (\text{I} \wedge \text{J}))$$



Alternate Solution



51-7

Solutions

$$3. (\text{A} \vee (\text{B} \wedge \text{C}) \vee \text{D}) \wedge \text{E} \wedge \text{F} \wedge (\text{G} \vee \text{H} \vee (\text{I} \wedge \text{J}))$$

$\wedge \vee \text{A}$
 $\vee \text{B} \wedge \text{C}$
 $\vee \text{D}$
 $\wedge \text{E}$
 $\wedge \text{F}$
 $\wedge \vee \text{G}$
 $\vee \text{H}$
 $\vee \text{I} \wedge \text{J}$

52-1

Solutions

$$3. (\text{A} \vee (\text{B} \wedge \text{C}) \vee \text{D}) \wedge \text{E} \wedge \text{F} \wedge (\text{G} \vee \text{H} \vee (\text{I} \wedge \text{J}))$$

$\wedge \vee \text{A}$
 $\vee \text{B} \wedge \text{C}$
 $\vee \text{D}$
 $\wedge \text{E}$
 $\wedge \text{F}$
 $\wedge \vee \text{G}$
 $\vee \text{H}$
 $\vee \text{I} \wedge \text{J}$

52-2

Solutions

$$3. (\text{A} \vee (\text{B} \wedge \text{C}) \vee \text{D}) \wedge \text{E} \wedge \text{F} \wedge (\text{G} \vee \text{H} \vee (\text{I} \wedge \text{J}))$$

$\wedge \vee \text{A}$
 $\vee \text{B} \wedge \text{C}$
 $\vee \text{D}$
 $\wedge \text{E}$
 $\wedge \text{F}$
 $\wedge \vee \text{G}$
 $\vee \text{H}$
 $\vee \text{I} \wedge \text{J}$

52-3

Solutions

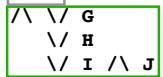
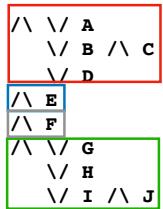
$$3. (\text{A} \vee (\text{B} \wedge \text{C}) \vee \text{D}) \wedge \text{E} \wedge \text{F} \wedge (\text{G} \vee \text{H} \vee (\text{I} \wedge \text{J}))$$

$\wedge \vee \text{A}$
 $\vee \text{B} \wedge \text{C}$
 $\vee \text{D}$
 $\wedge \text{E}$
 $\wedge \text{F}$
 $\wedge \vee \text{G}$
 $\vee \text{H}$
 $\vee \text{I} \wedge \text{J}$

52-4

Solutions

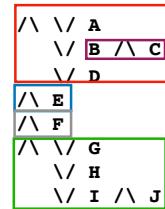
$$3. (\text{A} \vee \neg (\text{B} \wedge \text{C}) \vee \neg \text{D}) \wedge \text{E} \wedge \neg \text{F} \wedge (\text{G} \vee \neg \text{H} \vee \neg (\text{I} \wedge \text{J}))$$



52-5

Solutions

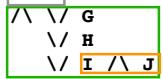
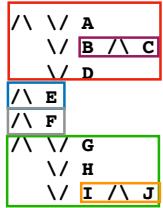
$$3. (\text{A} \vee \neg (\text{B} \wedge \text{C}) \vee \neg \text{D}) \wedge \text{E} \wedge \neg \text{F} \wedge (\text{G} \vee \neg \text{H} \vee \neg (\text{I} \wedge \text{J}))$$



52-6

Solutions

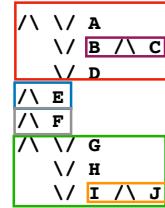
$$3. (\text{A} \vee \neg (\text{B} \wedge \text{C}) \vee \neg \text{D}) \wedge \text{E} \wedge \neg \text{F} \wedge (\text{G} \vee \neg \text{H} \vee \neg (\text{I} \wedge \text{J}))$$



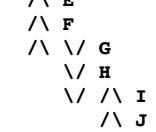
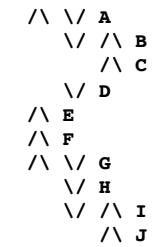
52-7

Solutions

$$3. (\text{A} \vee \neg (\text{B} \wedge \text{C}) \vee \neg \text{D}) \wedge \text{E} \wedge \neg \text{F} \wedge (\text{G} \vee \neg \text{H} \vee \neg (\text{I} \wedge \text{J}))$$



Alternate Solution



52-8

Sets

53-1

Sets

- An unordered collection of unique things.

53-2

Sets

- An unordered collection of unique things.
- No item is ever duplicated in a set.

53-3

Sets

- An unordered collection of unique things.
- No item is ever duplicated in a set.
- $\{\}$ is the “empty set”.

53-4

Sets

- An unordered collection of unique things.
- No item is ever duplicated in a set.
- {} is the “empty set”.
- {"a", "b", 123, {"a"}} is the set containing “a”, “b”, the number 123, and the set {"a"}.

53-5

Sets

- An unordered collection of unique things.
- No item is ever duplicated in a set.
- {} is the “empty set”.
- {"a", "b", 123, {"a"}} is the set containing “a”, “b”, the number 123, and the set {"a"}.
- {"a", "b"} = {"b", "a"} (i.e. order does not matter).

53-6

Sets

- An unordered collection of unique things.
- No item is ever duplicated in a set.
- {} is the “empty set”.
- {"a", "b", 123, {"a"}} is the set containing “a”, “b”, the number 123, and the set {"a"}.
- {"a", "b"} = {"b", "a"} (i.e. order does not matter).
- The concept maps to Python’s `set()`.

53-7

Sets

54-1

Sets

- \in is the “in set” operator, written as `\in` in ASCII

54-2

Sets

- \in is the “in set” operator, written as `\in` in ASCII
 - “a” $\in \{“a”, “b”\}$ is TRUE

54-3

Sets

- \in is the “in set” operator, written as `\in` in ASCII
 - “a” $\in \{“a”, “b”\}$ is TRUE
 - Python: `“a” in set([“a”, “b”])`

54-4

Sets

- \in is the “in set” operator, written as `\in` in ASCII
 - “a” $\in \{“a”, “b”\}$ is TRUE
 - Python: `“a” in set([“a”, “b”])`
 - “a” $\in \{“b”, “c”\}$ is FALSE

54-5

Sets

- \in is the “in set” operator, written as `\in` in ASCII
 - “a” $\in \{“a”, “b”\}$ is TRUE
 - Python: `“a” in set([“a”, “b”])`
 - “a” $\in \{“b”, “c”\}$ is FALSE
 - Python: `“a” in set([“b”, “c”])`

54-6

Sets

- \subseteq is the “subset” operator, written as `\subseteq` in ASCII

55-1

Sets

- \subseteq is the “subset” operator, written as `\subseteq` in ASCII
 - $\{“a”\} \subseteq \{“a”, “b”, “c”\}$ is TRUE

55-2

55-3

Sets

- \subseteq is the “subset” operator, written as `\subseteq` in ASCII
 - $\{a\} \subseteq \{a, b, c\}$ is TRUE
 - Python:
`set(["a"]).issubset(set(["a", "b", "c"]))`

55-4

Sets

- \subseteq is the “subset” operator, written as `\subseteq` in ASCII
 - $\{a\} \subseteq \{a, b, c\}$ is TRUE
 - Python:
`set(["a"]).issubset(set(["a", "b", "c"]))`
- $\{a, b\} \subseteq \{b, c\}$ is FALSE

55-5

Sets

- \subseteq is the “subset” operator, written as `\subseteq` in ASCII
 - $\{a\} \subseteq \{a, b, c\}$ is TRUE
 - Python:
`set(["a"]).issubset(set(["a", "b", "c"]))`
- $\{a, b\} \subseteq \{b, c\}$ is FALSE
- Python:
`set(["a", "b"]).issubset(set(["b", "c"]))`

55-6

Sets

56-1

Sets

- \cup is the “union” operator, written as `\union` in ASCII

56-2

Sets

- \cup is the “union” operator, written as `\union` in ASCII
- It “squishes together” two sets to create a new set

56-3

Sets

- \cup is the “union” operator, written as `\union` in ASCII
- It “squishes together” two sets to create a new set
 - $\{\text{“a”, “b”}\} \cup \{\} = \{\text{“a”, “b”}\}$

56-4

Sets

- \cup is the “union” operator, written as `\union` in ASCII
- It “squishes together” two sets to create a new set
 - $\{\text{“a”, “b”}\} \cup \{\} = \{\text{“a”, “b”}\}$
 - $\{\text{“a”, “b”}\} \cup \{\text{“c”, “d”}\} = \{\text{“a”, “b”, “c”, “d”}\}$

56-5

Sets

- \cup is the “union” operator, written as `\union` in ASCII
- It “squishes together” two sets to create a new set
 - $\{\text{“a”, “b”}\} \cup \{\} = \{\text{“a”, “b”}\}$
 - $\{\text{“a”, “b”}\} \cup \{\text{“c”, “d”}\} = \{\text{“a”, “b”, “c”, “d”}\}$
 - $\{\text{“a”, “b”}\} \cup \{\text{“b”, “c”}\} = \{\text{“a”, “b”, “c”}\}$

56-6

Sets

- \cup is the “union” operator, written as `\union` in ASCII
- It “squishes together” two sets to create a new set
 - $\{\text{“a”, “b”}\} \cup \{\} = \{\text{“a”, “b”}\}$
 - $\{\text{“a”, “b”}\} \cup \{\text{“c”, “d”}\} = \{\text{“a”, “b”, “c”, “d”}\}$
 - $\{\text{“a”, “b”}\} \cup \{\text{“b”, “c”}\} = \{\text{“a”, “b”, “c”}\}$
- Python:
`set([“a”, “b”]).union(set([“b”, “c”]))`

56-7

Sets

- \cap is the “intersect” operator, written as `\intersect` in ASCII

57-1

Sets

57-2

Sets

- \cap is the “intersect” operator, written as `\intersect` in ASCII
- It returns a new set containing the elements common to both sets

57-3

Sets

- \cap is the “intersect” operator, written as `\intersect` in ASCII
- It returns a new set containing the elements common to both sets
 - $\{\text{“a”, “b”}\} \cap \{\} = \{\}$

57-4

Sets

- \cap is the “intersect” operator, written as `\intersect` in ASCII
- It returns a new set containing the elements common to both sets
 - $\{\text{“a”, “b”}\} \cap \{\} = \{\}$
 - $\{\text{“a”, “b”}\} \cap \{\text{“c”, “d”}\} = \{\}$

57-5

Sets

- \cap is the “intersect” operator, written as `\intersect` in ASCII
- It returns a new set containing the elements common to both sets
 - $\{\text{“a”, “b”}\} \cap \{\} = \{\}$
 - $\{\text{“a”, “b”}\} \cap \{\text{“c”, “d”}\} = \{\}$
 - $\{\text{“a”, “b”}\} \cap \{\text{“b”, “c”}\} = \{\text{“b”}\}$

57-6

Sets

- \cap is the “intersect” operator, written as `\intersect` in ASCII
- It returns a new set containing the elements common to both sets
 - $\{\text{“a”, “b”}\} \cap \{\} = \{\}$
 - $\{\text{“a”, “b”}\} \cap \{\text{“c”, “d”}\} = \{\}$
 - $\{\text{“a”, “b”}\} \cap \{\text{“b”, “c”}\} = \{\text{“b”}\}$
 - Python:

57-7

Sets

- \cap is the “intersect” operator, written as `\intersect` in ASCII
- It returns a new set containing the elements common to both sets
 - $\{\text{“a”, “b”}\} \cap \{\} = \{\}$
 - $\{\text{“a”, “b”}\} \cap \{\text{“c”, “d”}\} = \{\}$
 - $\{\text{“a”, “b”}\} \cap \{\text{“b”, “c”}\} = \{\text{“b”}\}$
 - Python:
 - `set([“a”, “b”]).intersection(set([“b”, “c”]))`

57-8

Sets

- `..` is roughly equivalent to Python’s `range()` function

58-1

58-2

Sets

- .. is roughly equivalent to Python's `range()` function
- $1..10 = \{1,2,3,4,5,6,7,8,9,10\}$

58-3

Sets

- .. is roughly equivalent to Python's `range()` function
- $1..10 = \{1,2,3,4,5,6,7,8,9,10\}$
- In Python: `set(range(1,11))`

58-4

Exercises

(TRUE or FALSE for each)

$$\{\text{"a"}, \text{"b"}\} \subseteq \{\text{"b"}, \text{"a"}, \text{"c"}\}$$

$$\{1,2,3\} \subseteq (\{1,2,3\} \cap \{2,3\})$$

$$\text{"a"} \in ((\{\text{"b"}, \text{"c"}, \text{"d"}\} \cap \{\text{"e"}, \text{"f"}\}) \cup \{\text{"a"}\})$$

$$(\text{"a"} \in \{\text{"a"}, \text{"b"}\}) \vee (\text{"b"} \in \{\text{"c"}, \text{"d"}\})$$

59-1

Exercises

(TRUE or FALSE for each)

$$\{\text{"a"}, \text{"b"}\} \subseteq \{\text{"b"}, \text{"a"}, \text{"c"}\}$$

TRUE

$$\{1,2,3\} \subseteq (\{1,2,3\} \cap \{2,3\})$$

$$\text{"a"} \in ((\{\text{"b"}, \text{"c"}, \text{"d"}\} \cap \{\text{"e"}, \text{"f"}\}) \cup \{\text{"a"}\})$$

$$(\text{"a"} \in \{\text{"a"}, \text{"b"}\}) \vee (\text{"b"} \in \{\text{"c"}, \text{"d"}\})$$

59-2

Exercises

(TRUE or FALSE for each)

$$\{\text{``a''}, \text{``b''}\} \subseteq \{\text{``b''}, \text{``a''}, \text{``c''}\}$$

TRUE

$$\{1,2,3\} \subseteq (\{1,2,3\} \cap \{2,3\})$$

FALSE

$$\text{``a''} \in ((\{\text{``b''}, \text{``c''}, \text{``d''}\} \cap \{\text{``e''}, \text{``f''}\}) \cup \{\text{``a''}\})$$

$$(\text{``a''} \in \{\text{``a''}, \text{``b''}\}) \vee (\text{``b''} \in \{\text{``c''}, \text{``d''}\})$$

59-3

Exercises

(TRUE or FALSE for each)

$$\{\text{``a''}, \text{``b''}\} \subseteq \{\text{``b''}, \text{``a''}, \text{``c''}\}$$

TRUE

$$\{1,2,3\} \subseteq (\{1,2,3\} \cap \{2,3\})$$

FALSE

$$\text{``a''} \in ((\{\text{``b''}, \text{``c''}, \text{``d''}\} \cap \{\text{``e''}, \text{``f''}\}) \cup \{\text{``a''}\})$$

TRUE

$$(\text{``a''} \in \{\text{``a''}, \text{``b''}\}) \vee (\text{``b''} \in \{\text{``c''}, \text{``d''}\})$$

59-4

Exercises

(TRUE or FALSE for each)

$$\{\text{``a''}, \text{``b''}\} \subseteq \{\text{``b''}, \text{``a''}, \text{``c''}\}$$

TRUE

$$\{1,2,3\} \subseteq (\{1,2,3\} \cap \{2,3\})$$

FALSE

$$\text{``a''} \in ((\{\text{``b''}, \text{``c''}, \text{``d''}\} \cap \{\text{``e''}, \text{``f''}\}) \cup \{\text{``a''}\})$$

TRUE

$$(\text{``a''} \in \{\text{``a''}, \text{``b''}\}) \vee (\text{``b''} \in \{\text{``c''}, \text{``d''}\})$$

TRUE

59-5

“There exists”

60-1

“There exists”

- \exists means “there exists”, written as \exists in ASCII

60-2

“There exists”

- \exists means “there exists”, written as \exists in ASCII
- The usual form is $\exists x \in S : P(x)$

60-3

“There exists”

- \exists means “there exists”, written as \exists in ASCII
- The usual form is $\exists x \in S : P(x)$
 - This means “there exists some x in the set S such that $P(x)$ is TRUE”

60-4

“There exists”

- \exists means “there exists”, written as \exists in ASCII
- The usual form is $\exists x \in S : P(x)$
 - This means “there exists some x in the set S such that $P(x)$ is TRUE”

60-5

“There exists”

- \exists means “there exists”, written as \exists in ASCII
- The usual form is $\exists x \in S : P(x)$
 - This means “there exists some x in the set S such that $P(x)$ is TRUE”

60-6

“There exists”

- \exists means “there exists”, written as \exists in ASCII
- The usual form is $\exists x \in S : P(x)$
 - This means “there exists some x in the set S such that $P(x)$ is TRUE”

60-7

“There exists”

- \exists means “there exists”, written as \exists in ASCII
- The usual form is $\exists x \in S : P(x)$
 - This means “there exists some x in the set S such that $P(x)$ is TRUE”

60-8

“There exists”

- \exists means “there exists”, written as \exists in ASCII
- The usual form is $\exists x \in S : P(x)$
 - This means “there exists some x in the set S such that $P(x)$ is TRUE”
 - The entire expression evaluates to TRUE or FALSE

60-9

“There exists”

- \exists means “there exists”, written as \E in ASCII
- The usual form is $\exists x \in S : P(x)$
 - This means “there exists some x in the set S such that $P(x)$ is TRUE”
 - The entire expression evaluates to TRUE or FALSE
 - $\exists x \in \{1,2,3,4\} : x > 3$ is TRUE

60-10

“There exists”

- \exists means “there exists”, written as \E in ASCII
- The usual form is $\exists x \in S : P(x)$
 - This means “there exists some x in the set S such that $P(x)$ is TRUE”
 - The entire expression evaluates to TRUE or FALSE
 - $\exists x \in \{1,2,3,4\} : x > 3$ is TRUE
 - $\exists x \in \{1,2,3,4\} : x > 5$ is FALSE

60-11

“There exists”

“There exists”

$$\exists x \in \{1,2,3,4,5\} : x > 5$$

is roughly equivalent to the following Python expressions

61-1

61-2

“There exists”

$$\exists x \in \{1,2,3,4,5\} : x > 5$$

is roughly equivalent to the following Python expressions

```
def exists(S):
    for x in S:
        if x > 5:
            return True
    return False
```

61-3

“There exists”

$$\exists x \in \{1,2,3,4,5\} : x > 5$$

is roughly equivalent to the following Python expressions

```
def exists(S):
    for x in S:
        if x > 5:
            return True
    return False

exists(set([1,2,3,4,5])) # False
```

61-4

“There exists”

$$\exists x \in \{1,2,3,4,5\} : x > 5$$

is roughly equivalent to the following Python expressions

```
def exists(S):
    for x in S:
        if x > 5:
            return True
    return False

exists(set([1,2,3,4,5])) # False

any(map(lambda x: x > 5, [1,2,3,4,5])) #False
```

61-5

“For all”

62-1

“For all”

- \forall means “for all”, written as \A in ASCII

62-2

“For all”

- \forall means “for all”, written as \A in ASCII
- The usual form is $\forall x \in S : P(x)$

62-3

“For all”

- \forall means “for all”, written as \A in ASCII
- The usual form is $\forall x \in S : P(x)$
 - This means “for all x in the set S , it is the case that $P(x)$ is TRUE”

62-4

“For all”

- \forall means “for all”, written as \A in ASCII
- The usual form is $\forall x \in S : P(x)$
 - This means ‘**for all** x in the set S , it is the case that $P(x)$ is TRUE’

62-5

“For all”

- \forall means “for all”, written as \A in ASCII
- The usual form is $\forall x \in S : P(x)$
 - This means ‘for allx in the set S, it is the case that P(x) is TRUE’

62-6

“For all”

- \forall means “for all”, written as \A in ASCII
- The usual form is $\forall x \in S : P(x)$
 - This means ‘for allx in the set S, it is the case that P(x) is TRUE’

62-7

“For all”

- \forall means “for all”, written as \A in ASCII
- The usual form is $\forall x \in S : P(x)$
 - This means ‘for allx in the set S, it is the case that P(x) is TRUE’

62-8

“For all”

- \forall means “for all”, written as \A in ASCII
- The usual form is $\forall x \in S : P(x)$
 - This means ‘for allx in the set S, it is the case that P(x) is TRUE’
- The entire expression evaluates to TRUE or FALSE

62-9

“For all”

- \forall means “for all”, written as \A in ASCII
- The usual form is $\forall x \in S : P(x)$
 - This means ‘for all x in the set S , it is the case that $P(x)$ is TRUE’
 - The entire expression evaluates to TRUE or FALSE
 - $\forall x \in \{1,2,3,4\} : x > 3$ is FALSE

62-10

“For all”

- \forall means “for all”, written as \A in ASCII
- The usual form is $\forall x \in S : P(x)$
 - This means ‘for all x in the set S , it is the case that $P(x)$ is TRUE’
 - The entire expression evaluates to TRUE or FALSE
 - $\forall x \in \{1,2,3,4\} : x > 3$ is FALSE
 - $\forall x \in \{1,2,3,4\} : x > 0$ is TRUE

62-11

“For all”

“For all”

$$\forall x \in \{1,2,3,4,5\} : x > 5$$

is roughly equivalent to the following Python expressions

63-1

63-2

“For all”

$$\forall x \in \{1,2,3,4,5\} : x > 5$$

is roughly equivalent to the following Python expressions

```
def for_all(S):
    for x in S:
        if not (x > 5):
            return False
    return True
```

63-3

“For all”

$$\forall x \in \{1,2,3,4,5\} : x > 5$$

is roughly equivalent to the following Python expressions

```
def for_all(S):
    for x in S:
        if not (x > 5):
            return False
    return True

for_all(set([1,2,3,4,5])) # False
```

63-4

“For all”

$$\forall x \in \{1,2,3,4,5\} : x > 5$$

is roughly equivalent to the following Python expressions

```
def for_all(S):
    for x in S:
        if not (x > 5):
            return False
    return True

for_all(set([1,2,3,4,5])) # False

all(map(lambda x: x > 5, [1,2,3,4,5])) # False
```

63-5

Exercises

$$\exists x \in \{1,2,3,4\} : (x > 3) \wedge (x < 4)$$

$$\exists x \in \{1,2,3,4\} : \exists y \in \{5, 6, 7\} : x < y$$

$$\forall x \in \{1,2,3\} : \{x\} \subseteq \{1, 2\}$$

$$\exists x \in \{1,2,3\} : \{x\} \subseteq \{1, 2\}$$

$$\forall x \in \{3, 4, 5\} : \wedge x > 1 \\ \wedge x < 6$$

64-1

Exercises

$\exists x \in \{1,2,3,4\} : (x > 3) \wedge (x < 4)$ FALSE

$\exists x \in \{1,2,3,4\} : \exists y \in \{5, 6, 7\} : x < y$

$\forall x \in \{1,2,3\} : \{x\} \subseteq \{1, 2\}$

$\exists x \in \{1,2,3\} : \{x\} \subseteq \{1, 2\}$

$\forall x \in \{3, 4, 5\} : \wedge x > 1$
 $\quad \wedge x < 6$

64-2

Exercises

$\exists x \in \{1,2,3,4\} : (x > 3) \wedge (x < 4)$ FALSE

$\exists x \in \{1,2,3,4\} : \exists y \in \{5, 6, 7\} : x < y$ TRUE

$\forall x \in \{1,2,3\} : \{x\} \subseteq \{1, 2\}$

$\exists x \in \{1,2,3\} : \{x\} \subseteq \{1, 2\}$

$\forall x \in \{3, 4, 5\} : \wedge x > 1$
 $\quad \wedge x < 6$

64-3

Exercises

$\exists x \in \{1,2,3,4\} : (x > 3) \wedge (x < 4)$ FALSE

$\exists x \in \{1,2,3,4\} : \exists y \in \{5, 6, 7\} : x < y$ TRUE

$\forall x \in \{1,2,3\} : \{x\} \subseteq \{1, 2\}$ FALSE

$\exists x \in \{1,2,3\} : \{x\} \subseteq \{1, 2\}$

$\forall x \in \{3, 4, 5\} : \wedge x > 1$
 $\quad \wedge x < 6$

64-4

Exercises

$\exists x \in \{1,2,3,4\} : (x > 3) \wedge (x < 4)$ FALSE

$\exists x \in \{1,2,3,4\} : \exists y \in \{5, 6, 7\} : x < y$ TRUE

$\forall x \in \{1,2,3\} : \{x\} \subseteq \{1, 2\}$ FALSE

$\exists x \in \{1,2,3\} : \{x\} \subseteq \{1, 2\}$ TRUE

$\forall x \in \{3, 4, 5\} : \wedge x > 1$
 $\quad \wedge x < 6$

64-5

Exercises

$\exists x \in \{1,2,3,4\} : (x > 3) \wedge (x < 4)$ FALSE

$\exists x \in \{1,2,3,4\} : \exists y \in \{5, 6, 7\} : x < y$ TRUE

$\forall x \in \{1,2,3\} : \{x\} \subseteq \{1, 2\}$ FALSE

$\exists x \in \{1,2,3\} : \{x\} \subseteq \{1, 2\}$ TRUE

$\forall x \in \{3, 4, 5\} : \wedge x > 1$
 $\quad \quad \quad \wedge x < 6$ TRUE

64-6

Set constructors

65-1

Set constructors

$\{x \in S : P\}$ is like a **filter()** function, creating a new set consisting of the elements of **S** that satisfy **P**.

65-2

Set constructors

$\{x \in S : P\}$ is like a **filter()** function, creating a new set consisting of the elements of **S** that satisfy **P**.

$$\{x \in \{1,2,3,4,5\} : x > 3\} = \{4,5\}$$

65-3

Set constructors

$\{x \in S : P\}$ is like a `filter()` function, creating a new set consisting of the elements of **S** that satisfy **P**.

$$\{x \in \{1,2,3,4,5\} : x > 3\} = \{4,5\}$$

This is equivalent to the following Python expressions:

65-4

Set constructors

$\{x \in S : P\}$ is like a `filter()` function, creating a new set consisting of the elements of **S** that satisfy **P**.

$$\{x \in \{1,2,3,4,5\} : x > 3\} = \{4,5\}$$

This is equivalent to the following Python expressions:

```
set([x for x in [1,2,3,4,5] if x > 3])
```

65-5

Set constructors

$\{x \in S : P\}$ is like a `filter()` function, creating a new set consisting of the elements of **S** that satisfy **P**.

$$\{x \in \{1,2,3,4,5\} : x > 3\} = \{4,5\}$$

This is equivalent to the following Python expressions:

```
set([x for x in [1,2,3,4,5] if x > 3])
```

```
set(filter(lambda x: x > 3, [1,2,3,4,5]))
```

65-6

Set constructors

66-1

Set constructors

$\{ e : x \in S \}$ is like a `map()` function, applying `e` to every element of `S`.

$$\{ x * 2 : x \in \{1,2,3,4,5\} \} = \{2,4,6,8,10\}$$

66-2

Set constructors

$\{ e : x \in S \}$ is like a `map()` function, applying `e` to every element of `S`.

$$\{ x * 2 : x \in \{1,2,3,4,5\} \} = \{2,4,6,8,10\}$$

This is roughly equivalent to the following Python expressions:

66-3

Set constructors

$\{ e : x \in S \}$ is like a `map()` function, applying `e` to every element of `S`.

$$\{ x * 2 : x \in \{1,2,3,4,5\} \} = \{2,4,6,8,10\}$$

This is roughly equivalent to the following Python expressions:

```
set([x*2 for x in [1,2,3,4,5]])
```

66-4

Set constructors

$\{ e : x \in S \}$ is like a `map()` function, applying `e` to every element of `S`.

$$\{ x * 2 : x \in \{1,2,3,4,5\} \} = \{2,4,6,8,10\}$$

This is roughly equivalent to the following Python expressions:

```
set([x*2 for x in [1,2,3,4,5]])
```

```
set(map(lambda x: x*2, [1,2,3,4,5]))
```

66-5

Exercises

1. Write a set constructor that creates a set of elements from $\{1,2,3,4,5\}$, consisting of the elements that are greater than **1** and less than **5** (i.e. we want $\{2, 3, 4\}$).
2. Write a set constructor that creates a set of elements from $\{1,2,3,4,5\}$, consisting of the elements that, when multiplied by **3**, are greater than **10** (i.e. we want $\{4, 5\}$).
3. Write a set constructor that creates a set of elements consisting of each element of $\{1, 2, 3, 4, 5\}$ added to itself and then subtracting **1** (i.e. we want $\{1, 3, 5, 7, 9\}$).

67

Solutions

68-1

Solutions

$$1. \{ x \in \{1,2,3,4,5\} : x > 1 \quad \text{or} \quad \{ x \in \{1,2,3,4,5\} : x > 1 \wedge x < 5 \} \\ \wedge x < 5 \}$$

68-2

Solutions

$$1. \{ x \in \{1,2,3,4,5\} : x > 1 \quad \text{or} \quad \{ x \in \{1,2,3,4,5\} : x > 1 \wedge x < 5 \} \\ \wedge x < 5 \}$$

$$2. \{ x \in \{1,2,3,4,5\} : x^3 > 10 \}$$

68-3

Solutions

1. $\{ x \in \{1,2,3,4,5\} : x > 1 \quad \text{or} \quad \{ x \in \{1,2,3,4,5\} : x > 1 \wedge x < 5 \}$

2. $\{ x \in \{1,2,3,4,5\} : x^3 > 10 \}$

3. $\{ x+x-1 : x \in \{1,2,3,4,5\} \}$

68-4



69

Back to State Machines

70-1

Back to State Machines

Recall that a state machine is defined by three things:

70-2

Back to State Machines

Recall that a state machine is defined by three things:

1. The **variables**.

70-3

Back to State Machines

Recall that a state machine is defined by three things:

1. The **variables**.
2. The possible **initial values** of the variables.

70-4

Back to State Machines

Recall that a state machine is defined by three things:

1. The **variables**.
2. The possible **initial values** of the variables.
3. The possible **next states** from any given state (the relationship between the **values** of the **variables** in the current state and their possible **values** in the **next state**).

70-5

A small C program

```
int i;
void main () {
    i = someNumber();
    i = i + 1;
}
```

71-1

A small C program

```
int i;           Initialized to 0
void main () {
    i = someNumber();
    i = i + 1;
}
```

71-2

A small C program

```
int i;           Initialized to 0
void main () {
    i = someNumber(); Returns some value from 1 to 1000
    i = i + 1;
}
```

Let's exactly represent this C program as a state machine

71-4

A small C program

```
int i;           Initialized to 0
void main () {
    i = someNumber(); Returns some value from 1 to 1000
    i = i + 1;
}
```

71-3

A small C program

```
int i;           Initialized to 0
void main () {
    i = someNumber(); Returns some value from 1 to 1000
    i = i + 1;
}
```

Let's exactly represent this C program as a state machine

- Variables: i

71-5

A small C program

```
int i;           Initialized to 0
void main () {
    i = someNumber(); Returns some value from 1 to 1000
    i = i + 1;
}
```

Let's exactly represent this C program as a state machine

- Variables: i
- Possible initial values: i = 0

71-6

A small C program

```
int i;           Initialized to 0
void main () {
    i = someNumber(); Returns some value from 1 to 1000
    i = i + 1;
}
```

Let's exactly represent this C program as a state machine

- Variables: i
- Possible initial values: i = 0
- Next states for some given state... let's see!

71-7

A small C program - example behaviour

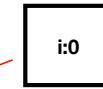
Assume that for this particular *behaviour*, `someNumber()` returns 42.

```
int i;
void main () {
    i = someNumber();
    i = i + 1;
}
```

72-1

A small C program - example behaviour

Assume that for this particular *behaviour*, `someNumber()` returns 42.

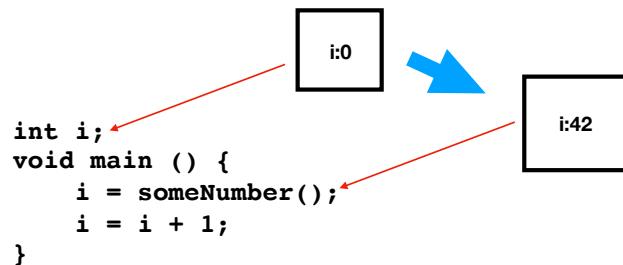


```
int i;           i:0
void main () {
    i = someNumber();
    i = i + 1;
}
```

72-2

A small C program - example behaviour

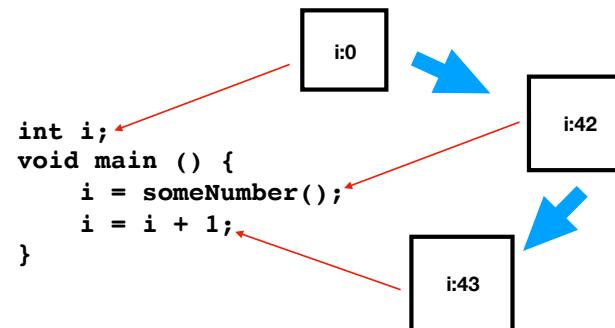
Assume that for this particular behaviour, `someNumber()` returns 42.



72-3

A small C program - example behaviour

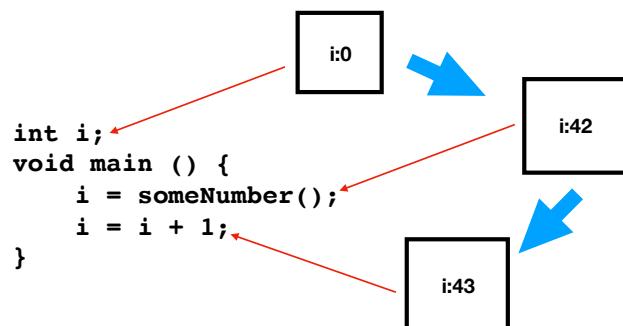
Assume that for this particular behaviour, `someNumber()` returns 42.



72-4

A small C program - example behaviour

Assume that for this particular behaviour, `someNumber()` returns 42.

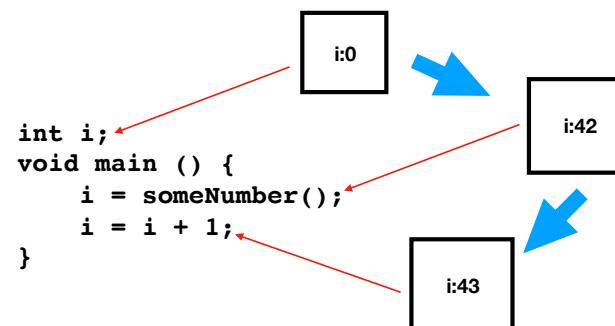


So we know that for the state `[i:42]`, the only next possible state is `[i:43]`.

72-5

A small C program - example behaviour

Assume that for this particular behaviour, `someNumber()` returns 42.



So we know that for the state `[i:42]`, the only next possible state is `[i:43]`. And for the state `[i:43]`, the only next state is “program termination”.

72-6

A small C program - example behaviour

Now let's look at a behaviour where `someNumber()` returned 43.

```
int i;
void main () {
    i = someNumber();
    i = i + 1;
}
```

73-1

A small C program - example behaviour

Now let's look at a behaviour where `someNumber()` returned 43.

```
int i;           i:0
void main () {   ↓
    i = someNumber(); i:43
    i = i + 1;
}
```

73-3

A small C program - example behaviour

Now let's look at a behaviour where `someNumber()` returned 43.

```
int i;           i:0
void main () {   ↓
    i = someNumber(); i:43
    i = i + 1;
}
```

73-2

A small C program - example behaviour

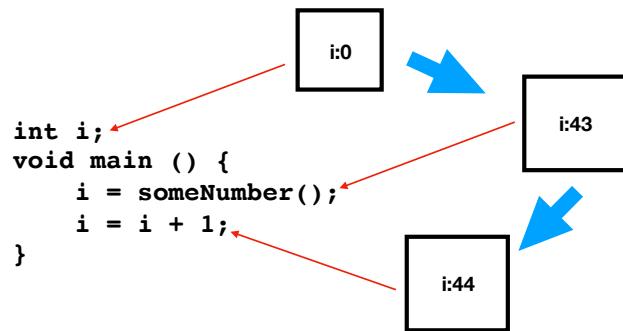
Now let's look at a behaviour where `someNumber()` returned 43.

```
int i;           i:0
void main () {   ↓
    i = someNumber(); i:43
    i = i + 1;      ↓
                    i:44
}
```

73-4

A small C program - example behaviour

Now let's look at a behaviour where `someNumber()` returned 43.

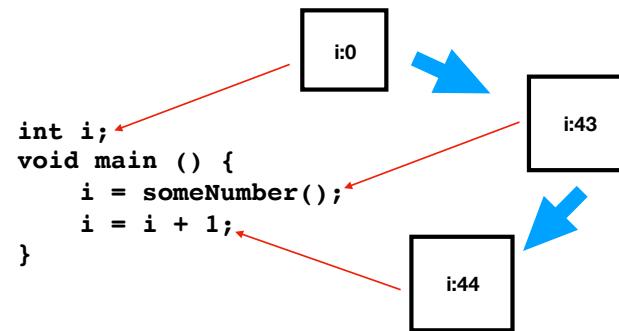


In this case, a state of `[i:43]` has a next state of `[i:44]`.

73-5

A small C program - example behaviour

Now let's look at a behaviour where `someNumber()` returned 43.

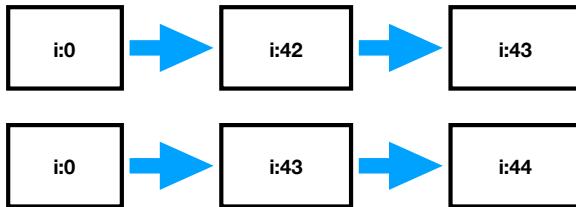


In this case, a state of `[i:43]` has a next state of `[i:44]`.

But that's different than the next state for `[i:43]` in the last behaviour!

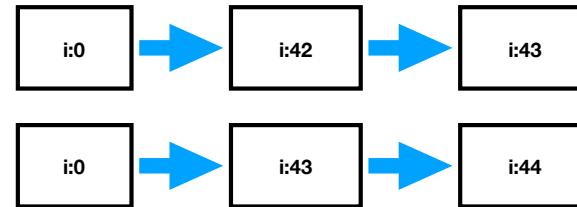
73-6

43



74-1

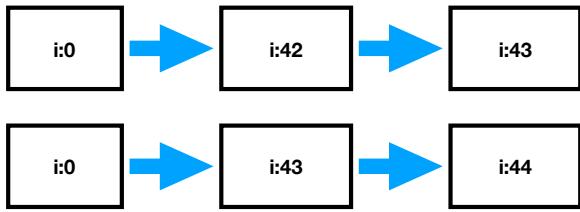
43



Thus representing the state of the program with just the variable `i` is not sufficient.

74-2

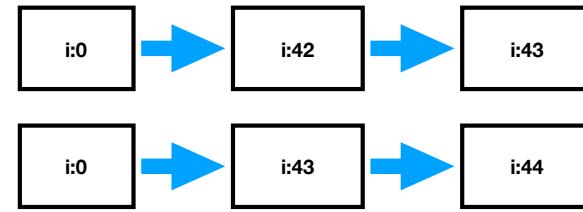
43



Thus representing the state of the program with just the variable **i** is not sufficient.
The two behaviours require different **next values** for **i** for the same **current value** of 43

74-3

43



Thus representing the state of the program with just the variable **i** is not sufficient.
The two behaviours require different **next values** for **i** for the same **current value** of 43
We must be able to exactly represent the program using the variables of our state machine.

74-4

A small C program - example behaviour

We can't represent this simple program using just the variable **i**

We need a second variable, **pc**, to represent "control state", i.e. where in the program we are

```
int i;
void main () {
    i = someNumber();
    i = i + 1;
}
```

75-1

A small C program - example behaviour

We can't represent this simple program using just the variable **i**

We need a second variable, **pc**, to represent "control state", i.e. where in the program we are

The diagram shows a C program with a red arrow pointing from the variable 'i' in the declaration 'int i;' to a box labeled 'i:0 pc:"start"'. The code is as follows:

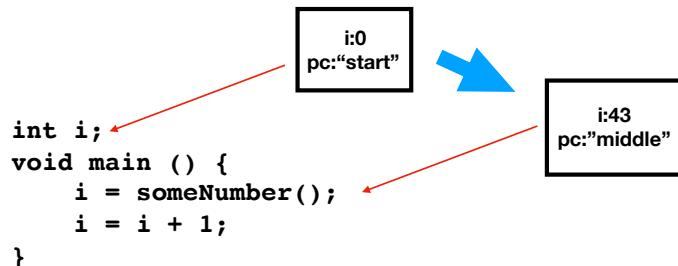
```
int i;
void main () {
    i = someNumber();
    i = i + 1;
}
```

75-2

A small C program - example behaviour

We can't represent this simple program using just the variable **i**

We need a second variable, **pc**, to represent "control state", i.e. where in the program we are

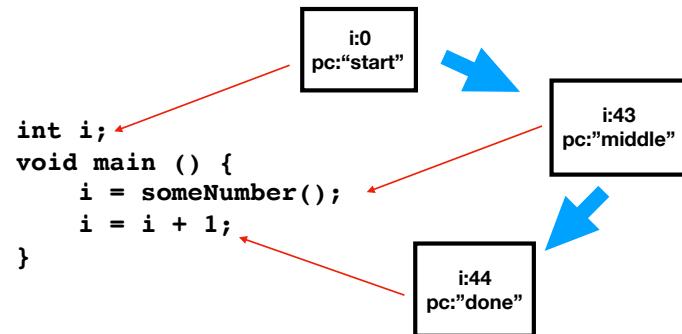


75-3

A small C program - example behaviour

We can't represent this simple program using just the variable **i**

We need a second variable, **pc**, to represent "control state", i.e. where in the program we are



75-4

A small C program

Let's make this a bit more formal

```
int i;
void main () {
    i = someNumber();
    i = i + 1;
}
```

76-1

A small C program

Let's make this a bit more formal

1. Variables: **i, pc**

```
int i;
void main () {
    i = someNumber();
    i = i + 1;
}
```

76-2

A small C program

Let's make this a bit more formal

1. Variables: i, pc
2. Initial values: i=0, pc="start"

```
int i;
void main () {
    i = someNumber();
    i = i + 1;
}
```

76-3

A small C program

Let's make this a bit more formal

1. Variables: i, pc
2. Initial values: i=0, pc="start"
3. Next state for a given state:

```
int i;
void main () {
    i = someNumber();
    i = i + 1;
}
```

```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```

76-5

A small C program

Let's make this a bit more formal

1. Variables: i, pc
2. Initial values: i=0, pc="start"
3. Next state for a given state:

```
int i;
void main () {
    i = someNumber();
    i = i + 1;
}
```

76-4

A small C program

Let's make this a bit more formal

1. Variables: i, pc
2. Initial values: i=0, pc="start"
3. Next state for a given state:

```
int i;
void main () {
    i = someNumber();
    i = i + 1;
}
```

```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```

76-6

A small C program

Let's make this a bit more formal

1. Variables: i, pc
2. Initial values: i=0, pc="start"
3. Next state for a given state:

```
int i;
void main () {
    i = someNumber();
    i = i + 1;
}
```

```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
    then
        next value of i = current value of i + 1
        next value of pc = "done"
    else
        no next values
```

76-7

A small C program

Let's make this a bit more formal

1. Variables: i, pc
2. Initial values: i=0, pc="start"
3. Next state for a given state:

```
int i;
void main () {
    i = someNumber();
    i = i + 1;
}
```

```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
    then
        next value of i = current value of i + 1
        next value of pc = "done"
    else
        no next values
```

76-8

A small C program

Let's make this a bit more formal

1. Variables: i, pc
2. Initial values: i=0, pc="start"
3. Next state for a given state:

```
int i;
void main () {
    i = someNumber();
    i = i + 1;
}
```

```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
    then
        next value of i = current value of i + 1
        next value of pc = "done"
    else
        no next values
```

76-9

A small C program

Let's make this a bit more formal

1. Variables: i, pc
2. Initial values: i=0, pc="start"
3. Next state for a given state:

```
int i;
void main () {
    i = someNumber();
    i = i + 1;
}
```

```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
    then
        next value of i = current value of i + 1
        next value of pc = "done"
    else
        no next values
```

76-10

A small C program

```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```

77-1

A small C program

```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```

i:0
pc:"start"

77-2

A small C program

```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```

i:0
pc:"start"



77-3

A small C program

```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```

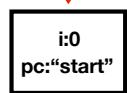
i:0
pc:"start"



77-4

A small C program

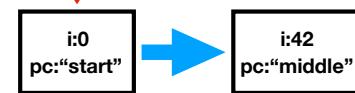
```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
    then
        next value of i = current value of i + 1
        next value of pc = "done"
    else
        no next values
```



77-5

A small C program

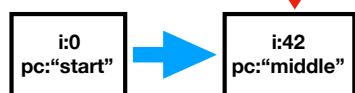
```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
    then
        next value of i = current value of i + 1
        next value of pc = "done"
    else
        no next values
```



77-6

A small C program

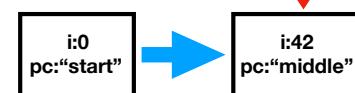
```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
    then
        next value of i = current value of i + 1
        next value of pc = "done"
    else
        no next values
```



77-7

A small C program

```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
    then
        next value of i = current value of i + 1
        next value of pc = "done"
    else
        no next values
```



77-8

A small C program

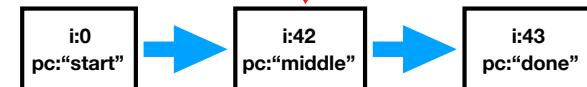
```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```



77-9

A small C program

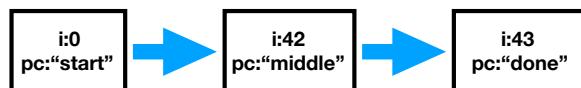
```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```



77-10

A small C program

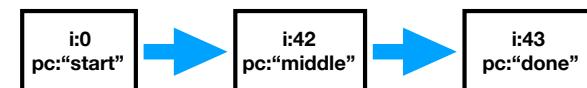
```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```



78-1

A small C program

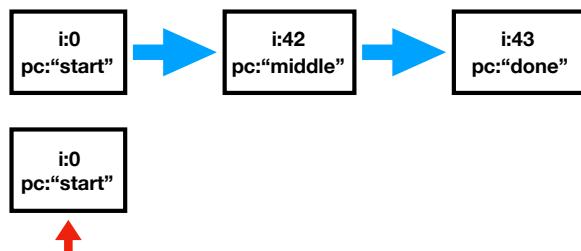
```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```



78-2

A small C program

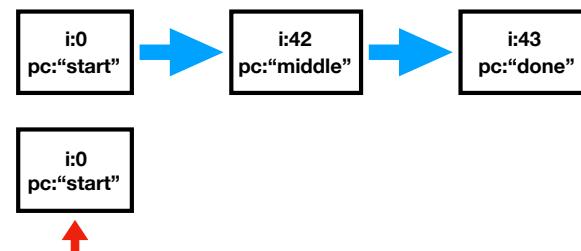
```
if current value of pc = "start"  
then  
    next value of i ∈ 1..1000  
    next value of pc = "middle"  
else if current value of pc = "middle"  
then  
    next value of i = current value of i + 1  
    next value of pc = "done"  
else  
    no next values
```



78-3

A small C program

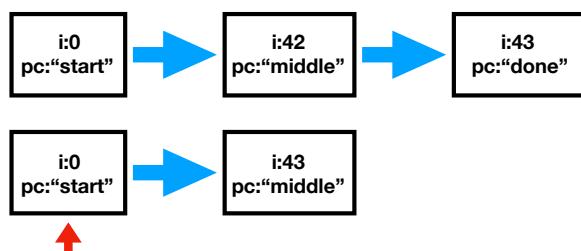
```
if current value of pc = "start"  
then  
    next value of i ∈ 1..1000  
    next value of pc = "middle"  
else if current value of pc = "middle"  
then  
    next value of i = current value of i + 1  
    next value of pc = "done"  
else  
    no next values
```



78-4

A small C program

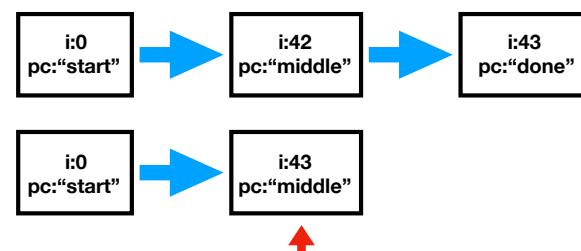
```
if current value of pc = "start"  
then  
    next value of i ∈ 1..1000  
    next value of pc = "middle"  
else if current value of pc = "middle"  
then  
    next value of i = current value of i + 1  
    next value of pc = "done"  
else  
    no next values
```



78-5

A small C program

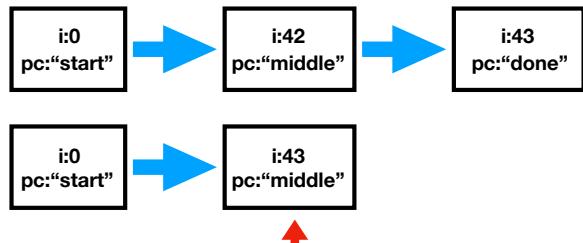
```
if current value of pc = "start"  
then  
    next value of i ∈ 1..1000  
    next value of pc = "middle"  
else if current value of pc = "middle"  
then  
    next value of i = current value of i + 1  
    next value of pc = "done"  
else  
    no next values
```



78-6

A small C program

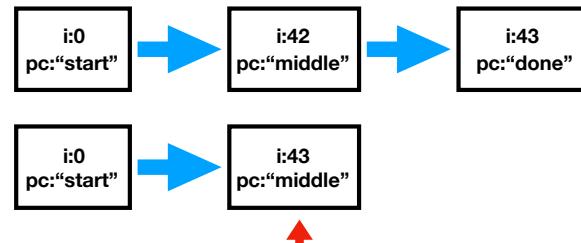
```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```



78-7

A small C program

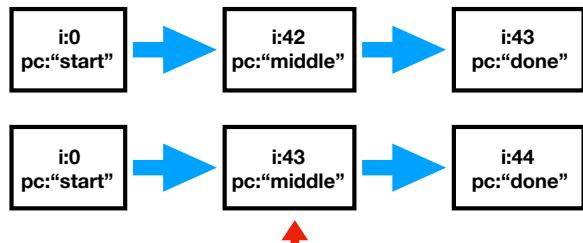
```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```



78-8

A small C program

```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```



78-9

Improving the next state formula

79-1

Improving the next state formula

```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```

79-2

Improving the next state formula

```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```

80-1

Improving the next state formula

```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```

pc replaces current value of pc

80-2

Improving the next state formula

```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```

pc replaces current value of pc

i replaces current value of i

80-3

Improving the next state formula

```
if pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if pc = "middle"
then
    next value of i = i + 1
    next value of pc = "done"
else
    no next values
```

81

Improving the next state formula

```
if pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if pc = "middle"
then
    next value of i = i + 1
    next value of pc = "done"
else
    no next values
```

82-1

Improving the next state formula

```
if pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if pc = "middle"
then
    next value of i = i + 1
    next value of pc = "done"
else
    no next values
```

i' replaces next value of i

82-2

Improving the next state formula

```
if pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if pc = "middle"
then
    next value of i = i + 1
    next value of pc = "done"
else
    no next values
```

i' replaces next value of i

pc' replaces next value of pc

82-3

Improving the next state formula

```
if pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if pc = "middle"
then
    next value of i = i + 1
    next value of pc = "done"
else
    no next values
```

i' replaces next value of i
pc' replaces next value of pc

These are called “primed” variables, and represent the value in the next state

82-4

= doesn't mean assignment!!!

84-1

Improving the next state formula

```
if pc = "start"
then
    i' ∈ 1..1000
    pc' = "middle"
else if pc = "middle"
then
    i' = i + 1
    pc' = "done"
else
    no next values
```

83

= doesn't mean assignment!!!

This means:
“If in the current state pc is equal to “start”, then a valid next state is any state in which i is any value between 1 and 1000, and pc is the string “middle”

84-2

= doesn't mean assignment!!!

This means:

"If in the current state pc is equal to "start", then a valid next state is any state in which i is any value between 1 and 1000, and pc is the string "middle"

```
if pc = "start"
then
  i' ∈ 1..1000
  pc' = "middle"
else if pc = "middle"
then
  i' = i + 1
  pc' = "done"
else
  no next values
```

84-3

= doesn't mean assignment!!!

This means:

"If in the current state pc is equal to "start", then a valid next state is any state in which i is any value between 1 and 1000, and pc is the string "middle"

```
if pc = "start"
then
  i' ∈ 1..1000
  pc' = "middle"
else if pc = "middle"
then
  i' = i + 1
  pc' = "done"
else
  no next values
```

84-4

= doesn't mean assignment!!!

This means:

"If in the current state pc is equal to "start", then a valid next state is any state in which i is any value between 1 and 1000, and pc is the string "middle"

```
if pc = "start"
then
  i' ∈ 1..1000
  pc' = "middle"
else if pc = "middle"
then
  i' = i + 1
  pc' = "done"
else
  no next values
```

84-5

= doesn't mean assignment!!!

This means:

"If in the current state pc is equal to "start", then a valid next state is any state in which i is any value between 1 and 1000, and pc is the string "middle"

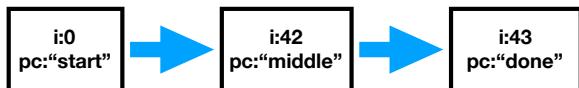
```
if pc = "start"
then
  i' ∈ 1..1000
  pc' = "middle"
else if pc = "middle"
then
  i' = i + 1
  pc' = "done"
else
  no next values
```

Again, NOT assignment, we are specifying the next possible states in a state machine.

84-6

State pairs

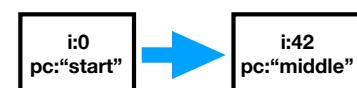
```
if pc = "start"
then
  i' ∈ 1..1000
  pc' = "middle"
else if pc = "middle"
then
  i' = i + 1
  pc' = "done"
else
  no next values
```



85

State pairs

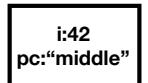
```
if pc = "start"
then
  i' ∈ 1..1000
  pc' = "middle"
else if pc = "middle"
then
  i' = i + 1
  pc' = "done"
else
  no next values
```



86

State pairs

```
if pc = "start"
then
  i' ∈ 1..1000
  pc' = "middle"
else if pc = "middle"
then
  i' = i + 1
  pc' = "done"
else
  no next values
```



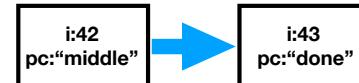
Current state

Next state

87

State pairs

```
if pc = "start"
then
  i' ∈ 1..1000
  pc' = "middle"
else if pc = "middle"
then
  i' = i + 1
  pc' = "done"
else
  no next values
```



Current state

Next state

88

State pairs

89-1

State pairs

- TLA+ is all about creating a formula that determines all possible pairs of states.

89-2

State pairs

- TLA+ is all about creating a formula that determines all possible pairs of states.
- We've said multiple times that a state machine is defined by three things:

89-3

State pairs

- TLA+ is all about creating a formula that determines all possible pairs of states.
- We've said multiple times that a state machine is defined by three things:
 1. **Variables.**

89-4

State pairs

- TLA+ is all about creating a formula that determines all possible pairs of states.
- We've said multiple times that a state machine is defined by three things:
 1. **Variables**.
 2. **Initial values** of the variables.

89-5

State pairs

- TLA+ is all about creating a formula that determines all possible pairs of states.
- We've said multiple times that a state machine is defined by three things:
 1. **Variables**.
 2. **Initial values** of the variables.
 3. Possible **next state** for a given state, i.e., relationship between **values of variables** in the current state and their **values** in the **next state**.

89-6

State pairs

- TLA+ is all about creating a formula that determines all possible pairs of states.
- We've said multiple times that a state machine is defined by three things:
 1. **Variables**.
 2. **Initial values** of the variables.
 3. Possible **next state** for a given state, i.e., relationship between **values of variables** in the current state and their **values** in the **next state**.
- That third point is really a **formula** that defines all possible current<->next state pairs (See your cheat sheet for a reminder of what a **formula** is).

89-7

State pairs

- TLA+ is all about creating a formula that determines all possible pairs of states.
- We've said multiple times that a state machine is defined by three things:
 1. **Variables**.
 2. **Initial values** of the variables.
 3. Possible **next state** for a given state, i.e., relationship between **values of variables** in the current state and their **values** in the **next state**.
- That third point is really a **formula** that defines all possible current<->next state pairs (See your cheat sheet for a reminder of what a **formula** is).
- This is SUPER important. Please stop me now with questions if you don't understand!

89-8

A little bit more formal

```
if pc = "start"
then
  i' ∈ 1..1000
  pc' = "middle"
else if pc = "middle"
then
  i' = i + 1
  pc' = "done"
else
  no next values
```

90-1

A little bit more formal

```
if pc = "start"
then
  i' ∈ 1..1000
  pc' = "middle"
else if pc = "middle"
then
  i' = i + 1
  pc' = "done"
else
  no next values
```

This is pretty formal, but to turn it into real TLA+, we need to go a bit further. This is a formula, it must always return a TRUE or FALSE

90-2

A little bit more formal

```
if pc = "start"
then
  i' ∈ 1..1000
  pc' = "middle"
else if pc = "middle"
then
  i' = i + 1
  pc' = "done"
else
  no next values
```

This is pretty formal, but to turn it into real TLA+, we need to go a bit further. This is a formula, it must always return a TRUE or FALSE

The “**then**” and “**else**” clauses must return TRUE or FALSE

90-3

A little bit more formal

```
if pc = "start"
then
  i' ∈ 1..1000
  pc' = "middle"
else if pc = "middle"
then
  i' = i + 1
  pc' = "done"
else
  no next values
```

($i' \in 1..1000 \wedge pc' = "middle"$) replaces $i' \in 1..1000$
 $pc' = "middle"$

91

A little bit more formal

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  i' = i + 1
  pc' = "done"
else
  no next values
```

92

A little bit more formal

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  i' = i + 1
  pc' = "done"
else
  no next values
```

(i' = i + 1) /\ (pc' = "done") replaces i' = i + 1
 pc' = "done"

93

A little bit more formal

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  (i' = i + 1) /\ (pc' = "done")
else
  no next values
```

94

A little bit more formal

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  (i' = i + 1) /\ (pc' = "done")
else
  no next values
```

FALSE replaces no next values

95

A little bit more formal

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  (i' = i + 1) /\ (pc' = "done")
else
  FALSE
```

96

Looking pretty good...

But we're not there just yet. Our formula is good, but we can make it better.

97

A little bit more formal

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  (i' = i + 1) /\ (pc' = "done")
else
  FALSE
```

98-1

A little bit more formal

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  (i' = i + 1) /\ (pc' = "done")
else
  FALSE
```

When does this formula return TRUE?

98-2

A little bit more formal

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  (i' = i + 1) /\ (pc' = "done")
else
  FALSE
```

When does this formula return TRUE?

There are two cases

98-3

Case 1

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  (i' = i + 1) /\ (pc' = "done")
else
  FALSE
```

99-1

Case 1

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  (i' = i + 1) /\ (pc' = "done")
else
  FALSE
```

When **pc="start"** in the current state

99-2

Case 1

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  (i' = i + 1) /\ (pc' = "done")
else
  FALSE
```

When **pc="start"** in the current state

99-3

Case 1

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  (i' = i + 1) /\ (pc' = "done")
else
  FALSE
```

When **pc="start"** in the current state

And $(i' \in 1..1000) \wedge (pc' = "middle")$ in the next state

99-4

Case 1

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  (i' = i + 1) /\ (pc' = "done")
else
  FALSE
```

When **pc="start"** in the current state

And $(i' \in 1..1000) \wedge (pc' = "middle")$ in the next state

99-5

Case 1

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  (i' = i + 1) /\ (pc' = "done")
else
  FALSE
```

When **pc="start"** in the current state

And $(i' \in 1..1000) \wedge (pc' = "middle")$ in the next state

Or in other words:

99-6

Case 1

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  (i' = i + 1) /\ (pc' = "done")
else
  FALSE
```

When **pc="start"** in the current state

And $(i' \in 1..1000) \wedge (pc' = "middle")$ in the next state

Or in other words:

$(pc = "start") \wedge ((i' \in 1..1000) \wedge (pc' = "middle"))$

99-7

Case 1

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
  then
    (i' = i + 1) /\ (pc' = "done")
  else
    FALSE
```

When **pc="start"** in the current state

And $(i' \in 1..1000) \wedge (pc' = "middle")$ in the next state

Or in other words:

(pc="start") /\ (($i' \in 1..1000$) /\ (pc' = "middle"))

99-8

Case 1

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
  then
    (i' = i + 1) /\ (pc' = "done")
  else
    FALSE
```

When **pc="start"** in the current state

And $(i' \in 1..1000) \wedge (pc' = "middle")$ in the next state

Or in other words:

(pc="start") /\ (($i' \in 1..1000$) /\ (pc' = "middle"))

99-9

Case 1

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
  then
    (i' = i + 1) /\ (pc' = "done")
  else
    FALSE
```

When **pc="start"** in the current state

And $(i' \in 1..1000) \wedge (pc' = "middle")$ in the next state

Or in other words:

(pc="start") /\ (($i' \in 1..1000$) /\ (pc' = "middle"))

99-10

Case 2

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
  then
    (i' = i + 1) /\ (pc' = "done")
  else
    FALSE
```

100-1

Case 2

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  (i' = i + 1) /\ (pc' = "done")
else
  FALSE
```

When **pc="middle"** in the current state

100-2

Case 2

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  (i' = i + 1) /\ (pc' = "done")
else
  FALSE
```

When **pc="middle"** in the current state

100-3

Case 2

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  (i' = i + 1) /\ (pc' = "done")
else
  FALSE
```

When **pc="middle"** in the current state

And $(i' = i + 1) /\ (pc' = "done")$ in the next state

100-4

Case 2

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  (i' = i + 1) /\ (pc' = "done")
else
  FALSE
```

When **pc="middle"** in the current state

And $(i' = i + 1) /\ (pc' = "done")$ in the next state

100-5

Case 2

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  (i' = i + 1) /\ (pc' = "done")
else
  FALSE
```

When **pc="middle"** in the current state

And $(i' = i + 1) \wedge (pc' = \text{"done"})$ in the next state

Or in other words:

100-6

Case 2

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  (i' = i + 1) /\ (pc' = "done")
else
  FALSE
```

When **pc="middle"** in the current state

And $(i' = i + 1) \wedge (pc' = \text{"done"})$ in the next state

Or in other words:

$(pc = \text{"middle"}) \wedge ((i' = i + 1) \wedge (pc' = \text{"done"}))$

100-7

Rewrite!

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  (i' = i + 1) /\ (pc' = "done")
else
  FALSE
```

101-1

Rewrite!

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  (i' = i + 1) /\ (pc' = "done")
else
  FALSE
```

becomes

101-2

Rewrite!

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  (i' = i + 1) /\ (pc' = "done")
else
  FALSE
```

becomes

```
((pc="start") /\ ((i' ∈ 1..1000) /\ (pc' = "middle")))
```

101-3

Rewrite!

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  (i' = i + 1) /\ (pc' = "done")
else
  FALSE
```

becomes

```
((pc="start") /\ ((i' ∈ 1..1000) /\ (pc' = "middle")))
\ / ((pc="middle") /\ ((i' = i + 1) /\ (pc' = "done")))
```

101-4

Rewrite!

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  (i' = i + 1) /\ (pc' = "done")
else
  FALSE
```

becomes

```
((pc="start") /\ ((i' ∈ 1..1000) /\ (pc' = "middle")))
\ / ((pc="middle") /\ ((i' = i + 1) /\ (pc' = "done")))
```

101-5

Rewrite!

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  (i' = i + 1) /\ (pc' = "done")
else
  FALSE
```

becomes

```
((pc="start") /\ ((i' ∈ 1..1000) /\ (pc' = "middle")))
\ / ((pc="middle") /\ ((i' = i + 1) /\ (pc' = "done")))
```

101-6

Rewrite!

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  (i' = i + 1) /\ (pc' = "done")
else
  FALSE
```

becomes

```
((pc="start") /\ ((i' ∈ 1..1000) /\ (pc' = "middle")))
\| ((pc="middle") /\ ((i' = i + 1) /\ (pc' = "done")))
```

101-7

Rewrite!

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  (i' = i + 1) /\ (pc' = "done")
else
  FALSE
```

becomes

```
((pc="start") /\ ((i' ∈ 1..1000) /\ (pc' = "middle")))
\| ((pc="middle") /\ ((i' = i + 1) /\ (pc' = "done")))
```

101-8

Rewrite!

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  (i' = i + 1) /\ (pc' = "done")
else
  FALSE
```

becomes

```
((pc="start") /\ ((i' ∈ 1..1000) /\ (pc' = "middle")))
\| ((pc="middle") /\ ((i' = i + 1) /\ (pc' = "done")))
```

101-9

Rewrite!

```
if pc = "start"
then
  (i' ∈ 1..1000) /\ (pc' = "middle")
else if pc = "middle"
then
  (i' = i + 1) /\ (pc' = "done")
else
  FALSE
```

becomes

```
((pc="start") /\ ((i' ∈ 1..1000) /\ (pc' = "middle")))
\| ((pc="middle") /\ ((i' = i + 1) /\ (pc' = "done")))
```

(We'll turn this into TLA syntax on the next slide.)

101-10

TLA+ Syntax Replacement

```
((pc="start") /\ ((i' ∈ 1..1000) /\ (pc' = "middle")))
\| ((pc="middle") /\ ((i' = i +1) /\ (pc' = "done")))
```

102-1

TLA+ Syntax Replacement

```
((pc="start") /\ ((i' ∈ 1..1000) /\ (pc' = "middle")))
\| ((pc="middle") /\ ((i' = i +1) /\ (pc' = "done")))
```

becomes

102-2

TLA+ Syntax Replacement

```
((pc="start") /\ ((i' ∈ 1..1000) /\ (pc' = "middle"))
\| ((pc="middle") /\ ((i' = i +1) /\ (pc' = "done")))
```

becomes

```
\| /\ pc = "start"
  /\ i' ∈ 1..1000
  /\ pc' = "middle"
\| /\ pc = "middle"
  /\ i' = i + 1
  /\ pc' = "done"
```

102-3

Comparison

```
int i;
void main () {
    i = someNumber();
    i = i + 1;
}
```

103-1

Comparison

```
int i;                                \/\ /\ pc = "start"
void main () {                         /\ i' ∈ 1..1000
    i = someNumber();                  /\ pc' = "middle"
    i = i + 1;                      \/\ /\ pc = "middle"
}                                         /\ i' = i + 1
                                         /\ pc' = "done"
```

103-2

Notes

- In C, = means “assignment”. In math, it means equality, like you learned in grade 1. What does “assignment” actually mean? It’s a very complicated concept.

Notes

104-1

Notes

- In C, = means “assignment”. In math, it means equality, like you learned in grade 1. What does “assignment” actually mean? It’s a very complicated concept.
- `someNumber()` returning a value from 1 to 1000 is very hard to specify in C. It represents non-determinism. Almost no programming languages were designed to express non-determinism.

104-2

104-3

Notes

- In C, = means “assignment”. In math, it means equality, like you learned in grade 1. What does “assignment” actually mean? It’s a very complicated concept.
- `someNumber()` returning a value from 1 to 1000 is very hard to specify in C. It represents non-determinism. Almost no programming languages were designed to express non-determinism.
- Think about how `someNumber()` would actually have to be implemented.

104-4

Notes

- In C, = means “assignment”. In math, it means equality, like you learned in grade 1. What does “assignment” actually mean? It’s a very complicated concept.
- `someNumber()` returning a value from 1 to 1000 is very hard to specify in C. It represents non-determinism. Almost no programming languages were designed to express non-determinism.
- Think about how `someNumber()` would actually have to be implemented.
- Non-determinism in math is easy. $i' \in 1..1000$ is non-determinism. “ i can be any value from 1 to 1000 in the next state”.

104-5

An aside

An aside

- Specifying at this level is incredibly rare.

105-1

105-2

An aside

- Specifying at this level is incredibly rare.
- Simple code like this is almost never specified in TLA+.

105-3

An aside

- Specifying at this level is incredibly rare.
- Simple code like this is almost never specified in TLA+.
- It **can** be specified, but it's not where the power lies.

105-4

An aside

- Specifying at this level is incredibly rare.
- Simple code like this is almost never specified in TLA+.
- It **can** be specified, but it's not where the power lies.
- The point of this exercise has been to go step by step from something you already understand to full TLA+ syntax.

105-5

An aside

- Specifying at this level is incredibly rare.
- Simple code like this is almost never specified in TLA+.
- It **can** be specified, but it's not where the power lies.
- The point of this exercise has been to go step by step from something you already understand to full TLA+ syntax.
- This spec is 6 lines long. The TLA+ spec is longer. Usually, TLA+ specs are orders of magnitude **shorter** than the systems they specify.

105-6

Next State

106-1

Next State

```
\/ /\ pc = "start"
/\ i' \in 1..1000
/\ pc' = "middle"
/\ /\ pc = "middle"
/\ i' = i + 1
/\ pc' = "done"
```

106-2

Next State

```
\/ /\ pc = "start"
/\ i' \in 1..1000
/\ pc' = "middle"
/\ /\ pc = "middle"
/\ i' = i + 1
/\ pc' = "done"
```

- This formula defines all the valid state pairs.

Next State

```
\/ /\ pc = "start"
/\ i' \in 1..1000
/\ pc' = "middle"
/\ /\ pc = "middle"
/\ i' = i + 1
/\ pc' = "done"
```

- This formula defines all the valid state pairs.
- It does NOT say “if pc='middle' then set i to i + 1 and set pc to ‘done’ ”.

106-3

106-4

Next State

```
\forall \exists pc = "start"
  \exists i' \in 1..1000
    \exists pc' = "middle"
\forall \exists pc = "middle"
  \exists i' = i + 1
  \exists pc' = "done"
```

- This formula defines all the valid state pairs.
- It does NOT say “if `pc='middle'` then set `i` to `i + 1` and set `pc` to ‘done’”.
- So... What’s actually going on? What is this really saying?

106-5

Behaviours

Behaviours

- Think back to our Die Hard example

107-1

Behaviours

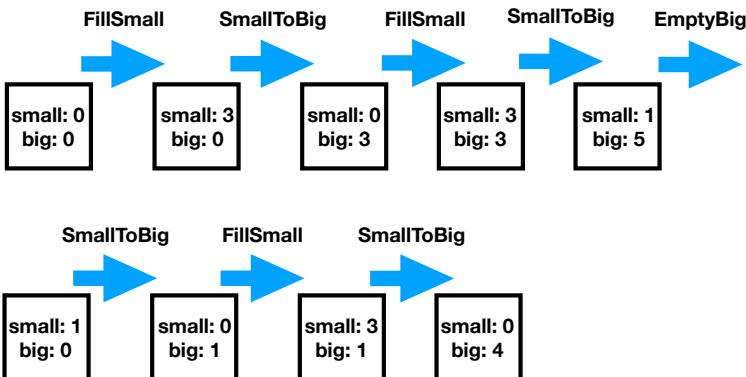
- Think back to our Die Hard example
- We showed three possible behaviours

107-2

107-3

Behaviours

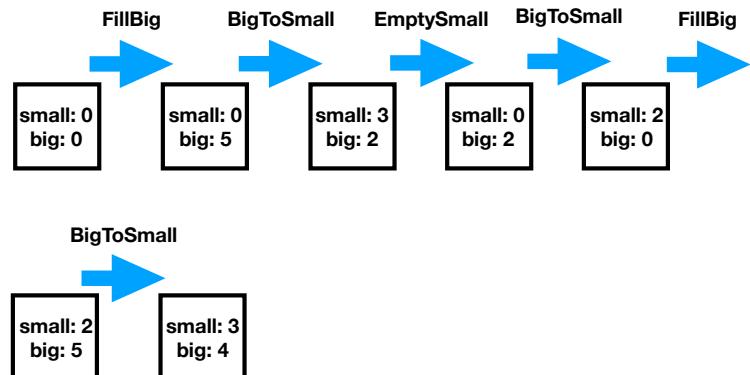
- Think back to our Die Hard example
- We showed three possible behaviours



107-4

Behaviours

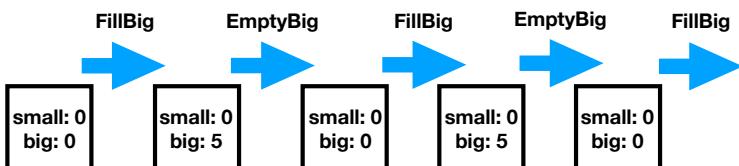
- Think back to our Die Hard example
- We showed three possible behaviours



107-5

Behaviours

- Think back to our Die Hard example
- We showed three possible behaviours



107-6

Behaviours

- Think back to our Die Hard example
- We showed three possible behaviours
- An INFINITE number of behaviours are possible

107-7

Behaviours

- Think back to our Die Hard example
- We showed three possible behaviours
- An INFINITE number of behaviours are possible
- Not only behaviours that Fill/Empty/Pour, but an infinite number of values can be assigned to **big** and **small**

107-8

Behaviours

- Think back to our Die Hard example
- We showed three possible behaviours
- An INFINITE number of behaviours are possible
- Not only behaviours that Fill/Empty/Pour, but an infinite number of values can be assigned to **big** and **small**



107-9

Behaviours

108-1

Behaviours

- The universe presents an infinite combination of behaviours, where an infinite number of **values** can be assigned to our **variables**.

108-2

Behaviours

- The universe presents an infinite combination of behaviours, where an infinite number of **values** can be assigned to our **variables**.
- From any **state**, an infinite number of **next states** are possible.

108-3

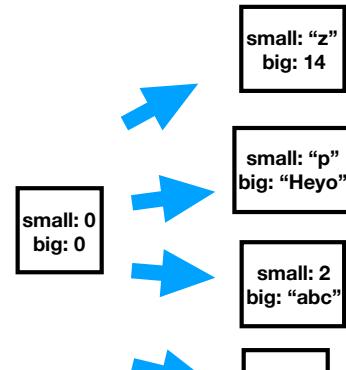
Behaviours

- The universe presents an infinite combination of behaviours, where an infinite number of **values** can be assigned to our **variables**.
- From any **state**, an infinite number of **next states** are possible.
- The point of our **formula** is to restrict which of those **state pairs** are permitted. All the combinations of allowed state pairs make up our possible behaviours.

108-5

Behaviours

- The universe presents an infinite combination of behaviours, where an infinite number of **values** can be assigned to our **variables**.
- From any **state**, an infinite number of **next states** are possible.



108-4

Behaviours

- The universe presents an infinite combination of behaviours, where an infinite number of **values** can be assigned to our **variables**.
- From any **state**, an infinite number of **next states** are possible.
- The point of our **formula** is to restrict which of those **state pairs** are permitted. All the combinations of allowed state pairs make up our possible behaviours.
- This is the foundational concept of TLA+.

108-6

Behaviours

- The universe presents an infinite combination of behaviours, where an infinite number of **values** can be assigned to our **variables**.
- From any **state**, an infinite number of **next states** are possible.
- The point of our **formula** is to restrict which of those **state pairs** are permitted. All the combinations of allowed state pairs make up our possible behaviours.
- This is the foundational concept of TLA+.
- The formula restricts which subset of infinite behaviours is allowed, by specifying the allowed transitions from one state to another.

108-7

Behaviours

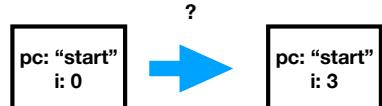
```
\!/\! pc = "start"
\!/\! i' \in 1..1000
\!/\! pc' = "middle"
\!/\! pc = "middle"
\!/\! i' = i + 1
\!/\! pc' = "done"
```



109-1

Behaviours

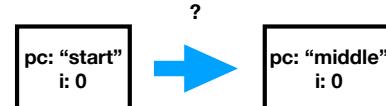
```
\!/\! pc = "start"
\!/\! i' \in 1..1000
\!/\! pc' = "middle"
\!/\! pc = "middle"
\!/\! i' = i + 1
\!/\! pc' = "done"
```



109-2

Behaviours

```
\!/\! pc = "start"
\!/\! i' \in 1..1000
\!/\! pc' = "middle"
\!/\! pc = "middle"
\!/\! i' = i + 1
\!/\! pc' = "done"
```



109-3

Behaviours

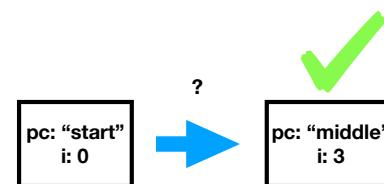
```
\ / \ pc = "start"
  \ i' ∈ 1..1000
    \ pc' = "middle"
\ / \ pc = "middle"
  \ i' = i + 1
  \ pc' = "done"
```



109-4

Behaviours

```
\ / \ pc = "start"
  \ i' ∈ 1..1000
    \ pc' = "middle"
\ / \ pc = "middle"
  \ i' = i + 1
  \ pc' = "done"
```



109-5

Behaviours

```
\ / \ pc = "start"
  \ i' ∈ 1..1000
    \ pc' = "middle"
\ / \ pc = "middle"
  \ i' = i + 1
  \ pc' = "done"
```



110-1

Behaviours

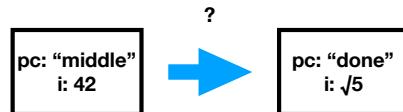
```
\ / \ pc = "start"
  \ i' ∈ 1..1000
    \ pc' = "middle"
\ / \ pc = "middle"
  \ i' = i + 1
  \ pc' = "done"
```



110-2

Behaviours

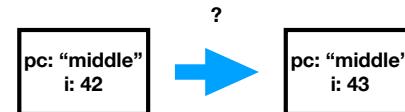
```
\/ /\ pc = "start"
 /\ i' ∈ 1..1000
 /\ pc' = "middle"
/\ /\ pc = "middle"
 /\ i' = i + 1
 /\ pc' = "done"
```



110-3

Behaviours

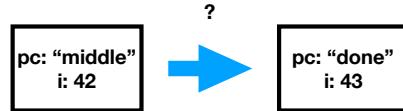
```
\/ /\ pc = "start"
 /\ i' ∈ 1..1000
 /\ pc' = "middle"
/\ /\ pc = "middle"
 /\ i' = i + 1
 /\ pc' = "done"
```



110-4

Behaviours

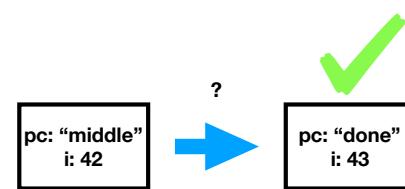
```
\/ /\ pc = "start"
 /\ i' ∈ 1..1000
 /\ pc' = "middle"
/\ /\ pc = "middle"
 /\ i' = i + 1
 /\ pc' = "done"
```



110-5

Behaviours

```
\/ /\ pc = "start"
 /\ i' ∈ 1..1000
 /\ pc' = "middle"
/\ /\ pc = "middle"
 /\ i' = i + 1
 /\ pc' = "done"
```



110-6

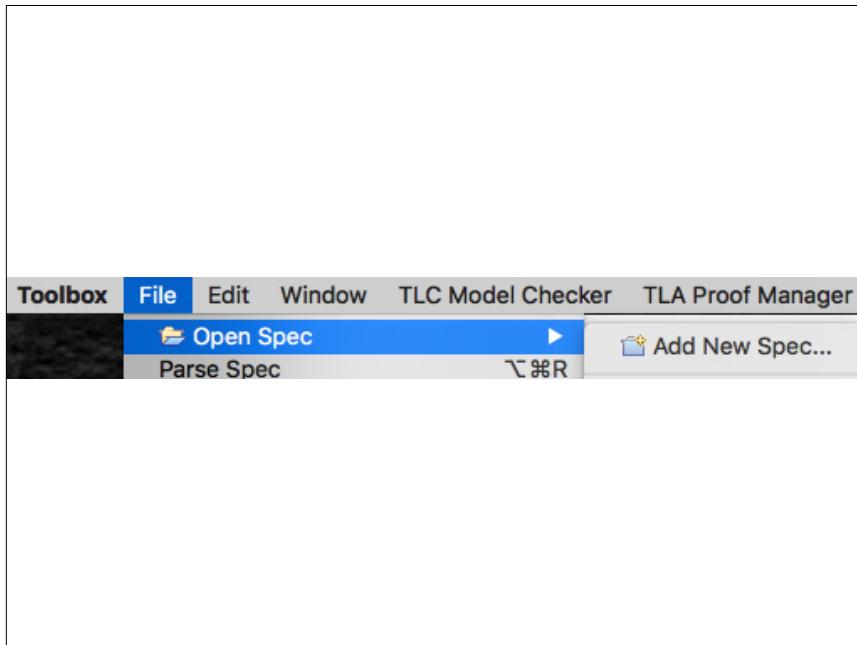
Final Spec for our C Program

111-1

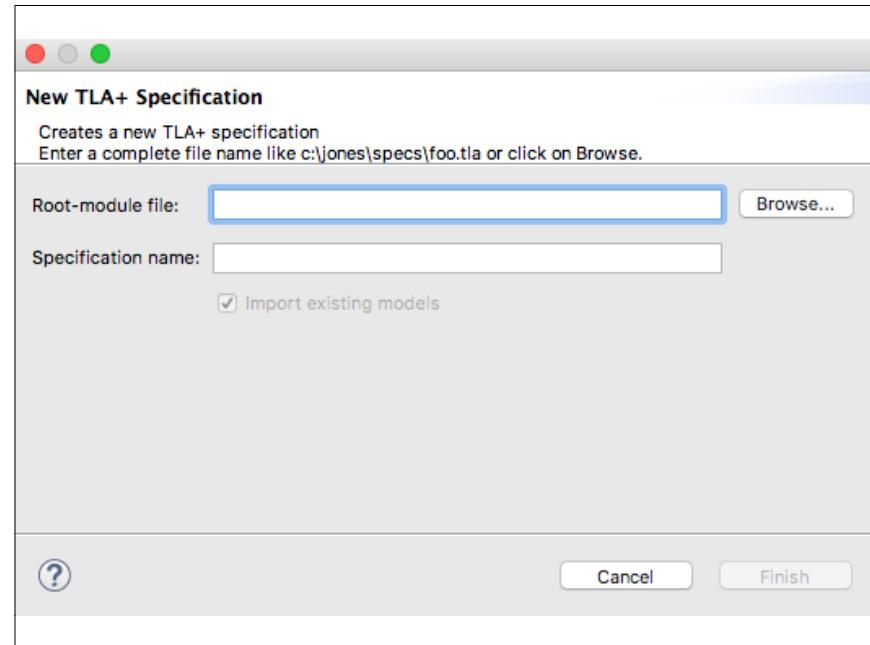
Final Spec for our C Program

```
— MODULE SimpleProgram —  
  
EXTENDS Integers  
VARIABLES i, pc  
  
Init == (pc="start") /\ (i=0)  
  
Next ==  
  /\ /\ pc = "start"  
  /\ i' \in 1..1000  
  /\ pc' = "middle"  
  /\ /\ pc = "middle"  
  /\ i' = i + 1  
  /\ pc' = "done"
```

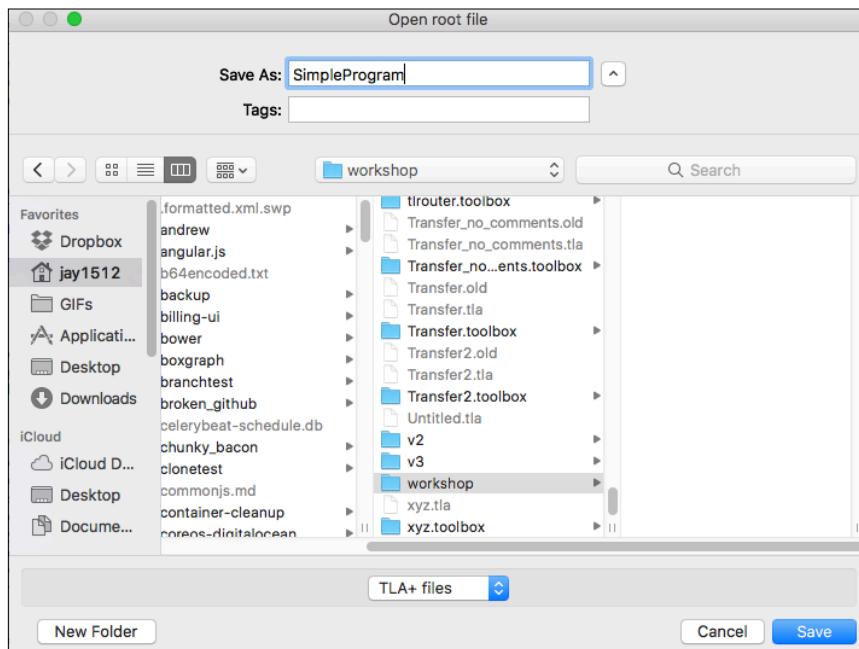
111-2



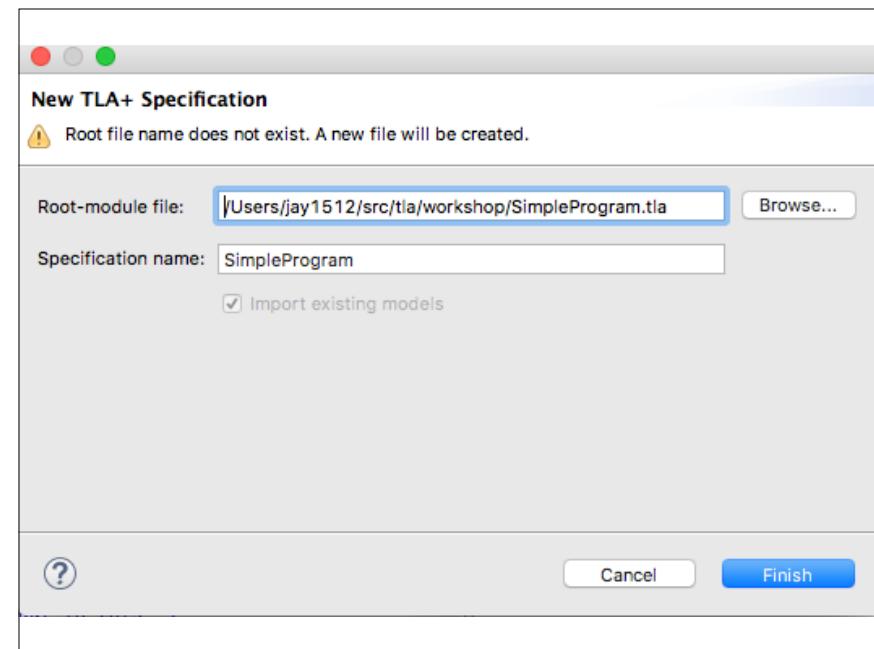
112



113



114



115

The screenshot shows a TLA+ editor window titled "SimpleProgram.tla" with the file path "/Users/jay1512/src/tla/workshop/SimpleProgram.tla". The code editor displays the following TLA+ module:

```
1 ----- MODULE SimpleProgram -----
2
3
4
5 /* Modification History
6 * Created Thu Sep 14 13:03:35 EDT 2017 by jay1512
7
```

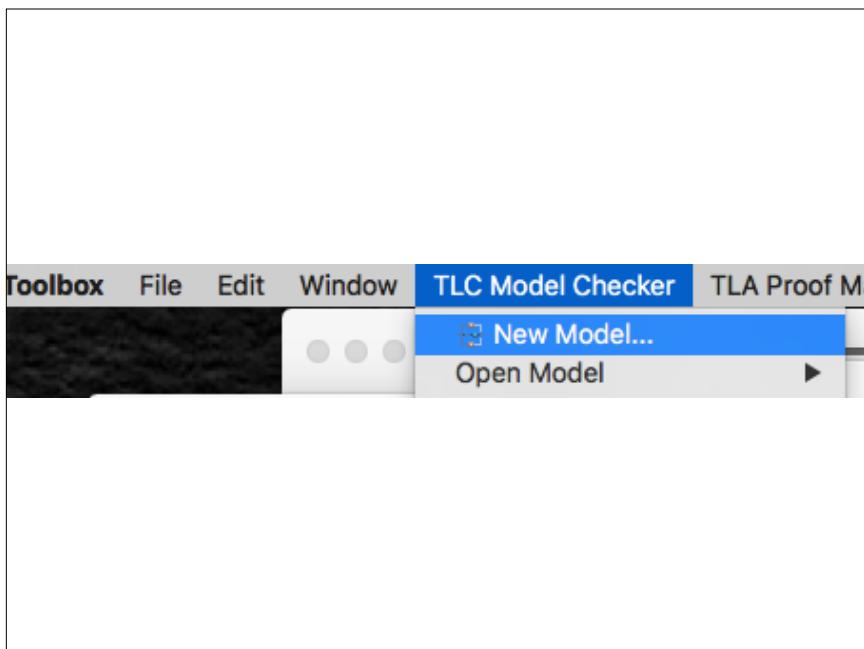
116

The screenshot shows a TLA+ editor window titled "SimpleProgram.tla" with the file path "/Users/jay1512/src/tla/workshop/SimpleProgram.tla". The code editor displays the following TLA+ module, with line 13 highlighted in blue:

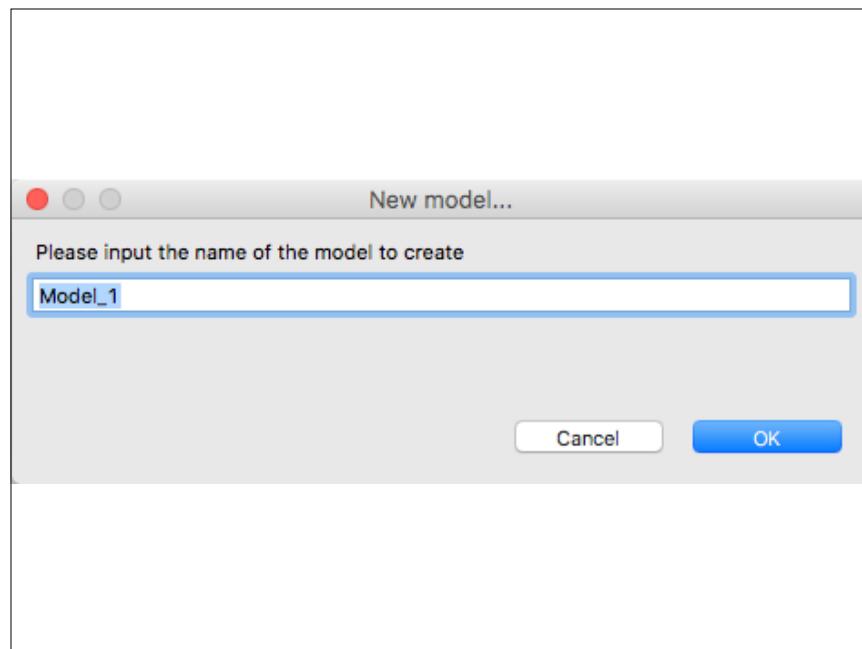
```
1 ----- MODULE SimpleProgram -----
2 EXTENDS Integers
3 VARIABLES i, pc
4
5 Init == (pc="start") ∧ (i=0)
6
7 Next ==
8   ∨ ∧ pc = "start"
9   ∧ i' \in 1..1000
10  ∧ pc' = "middle"
11  ∨ ∧ pc = "middle"
12  ∧ i' = i + 1
13  ∧ pc' = "done"
14
15
16 /* Modification History
17 * Last modified Thu Sep 14 13:04:45 EDT 2017 by jay1512
```

In GitHub at [specs/SimpleProgram.tla](#)

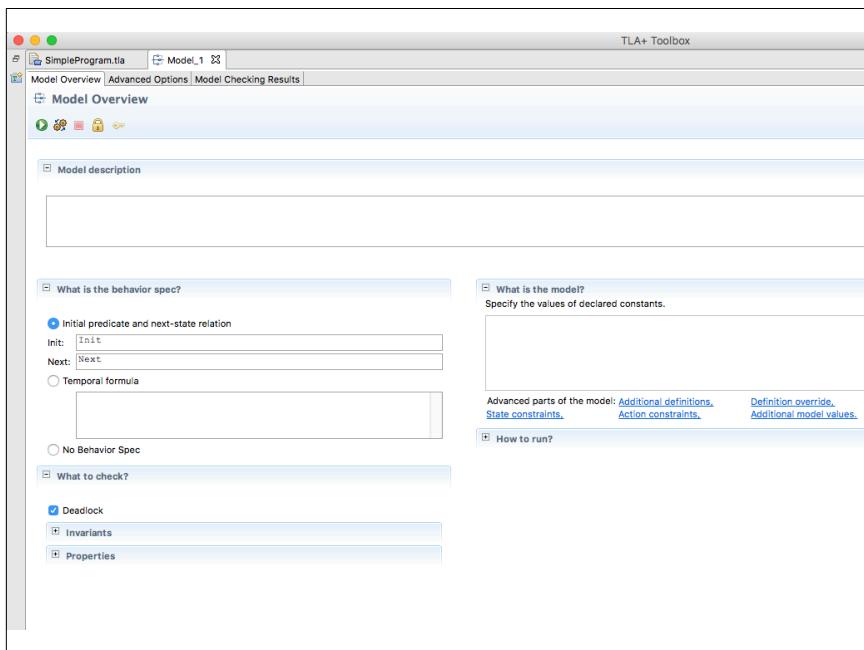
117



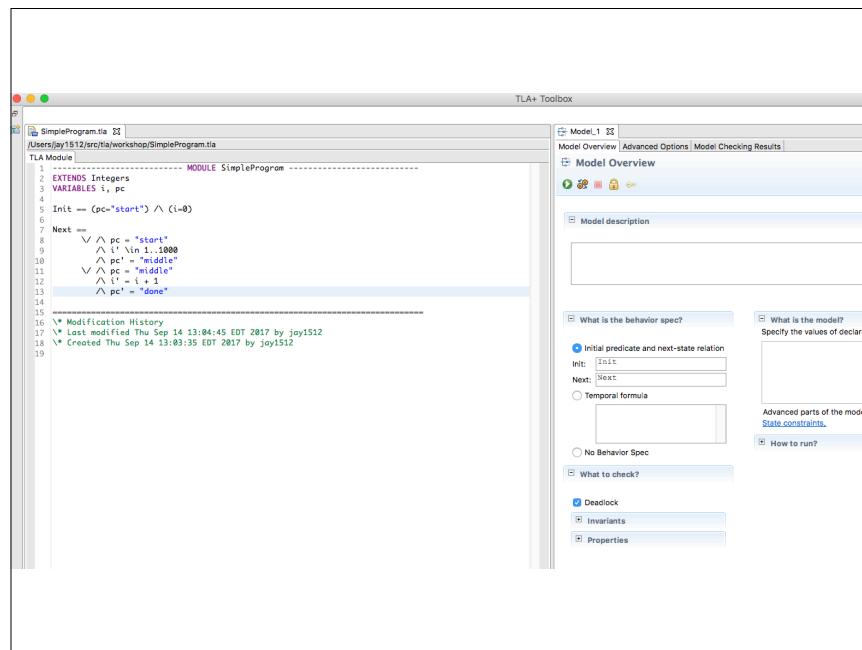
118



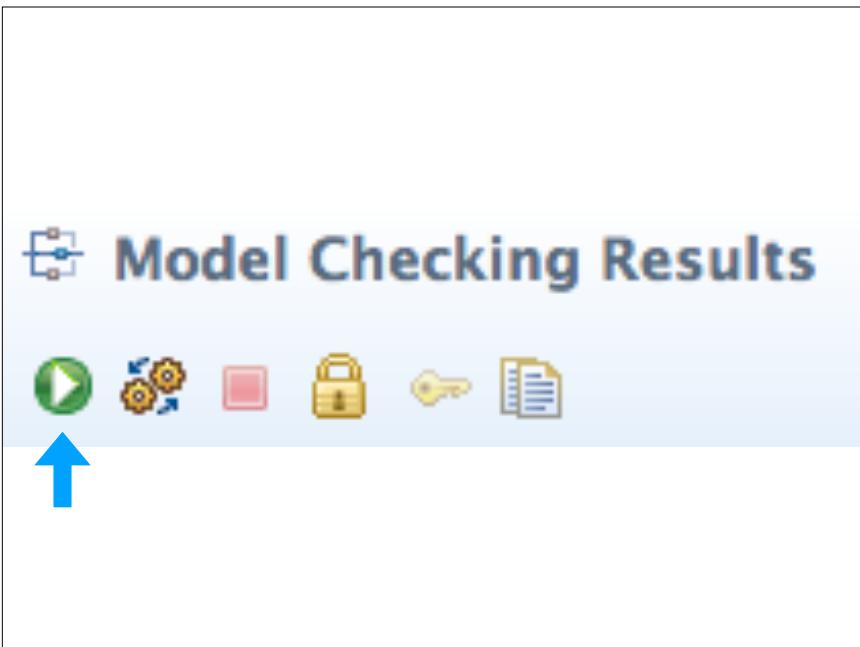
119



120



121



122

Model Overview | Advanced Options | Model Checking Results

Model Checking Results State space exploration incomplete

General

Start time: Thu Sep 14 13:06:15 EDT 2017
End time: Thu Sep 14 13:06:15 EDT 2017
Last checkpoint time:
Current status: Not running
Errors detected: 1 Error
Fingerprint collision probability:

Statistics

Time	Diameter	States Found	Distinct States	Queue Size
2017-09-14 13:06...	2	1011	1011	998

Coverage analysis

Module	SimpleProg	SimpleProg	SimpleProg	SimpleProg
i	"start"			
pc				

Error-Trace Exploration

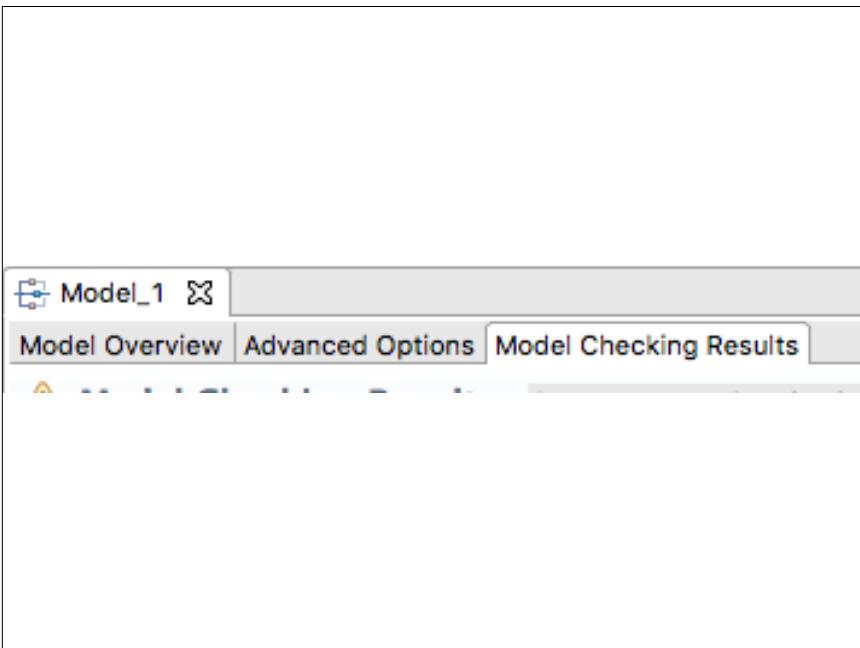
Name	Value
<Initial predicate>	State (num = 1)
i	0
pc	
<Action line 8, col 10 to 11>	State (num = 2)
i	3
pc	"middle"
<Action line 11, col 10 to 11>	State (num = 3)
i	4
pc	"done"

TLC Errors

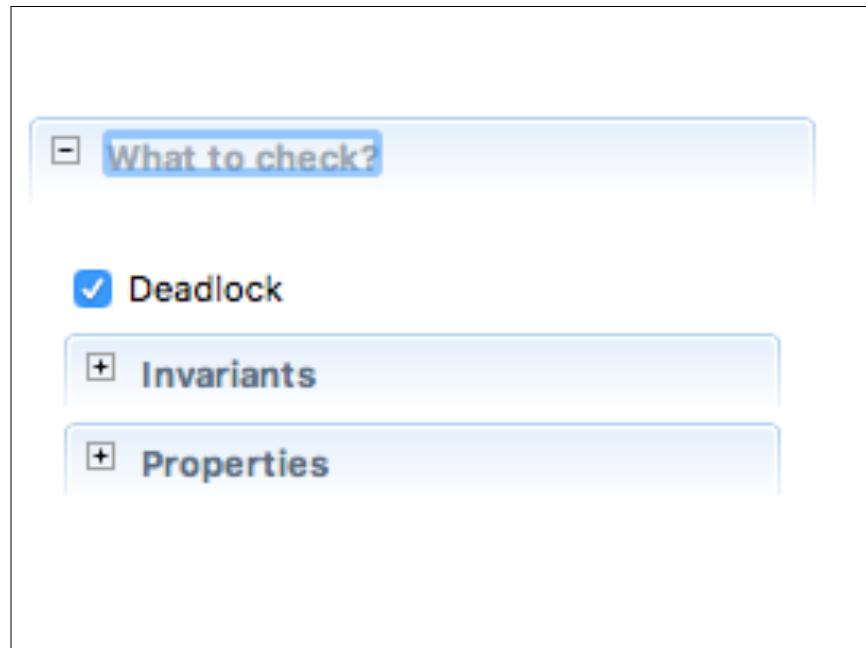
Model_1

Deadlock reached.

123



124



125

What to check?

Deadlock

Invariants

Properties

126

Model_1

Model Overview | Advanced Options | Model Checking Results

Model Checking Results

General

Start time:	Thu Sep 14 13:06:41 EDT 2017
End time:	Thu Sep 14 13:06:41 EDT 2017
Last checkpoint time:	
Current status:	Not running
Errors detected:	No errors
Fingerprint collision probability:	calculated: 0.0, observed: 1.1E-13

Statistics

Time	Diameter	States Found	Distinct States	Queue Size
2017-09-14 13:06...	3	2001	2001	0

Coverage at 2017-09-14 13:06:41

Module	Location
SimpleProg...	line 10, col 13 to line 10, col 2
SimpleProg...	line 12, col 13 to line 12, col 2
SimpleProg...	line 13, col 13 to line 13, col 2
SimpleProg...	line 9, col 13 to line 9, col 26

Evaluate Constant Expression

Expression:	Value:
-------------	--------

127-1

Model_1

Model Overview | Advanced Options | Model Checking Results

Model Checking Results

General

Start time:	Thu Sep 14 13:06:41 EDT 2017
End time:	Thu Sep 14 13:06:41 EDT 2017
Last checkpoint time:	
Current status:	Not running
Errors detected:	No errors
Fingerprint collision probability:	calculated: 0.0, observed: 1.1E-13

Statistics

Time	Diameter	States Found	Distinct States	Queue Size
2017-09-14 13:06...	3	2001	2001	0

Coverage at 2017-09-14 13:06:41

Module	Location
SimpleProg...	line 10, col 13 to line 10, col 2
SimpleProg...	line 12, col 13 to line 12, col 2
SimpleProg...	line 13, col 13 to line 13, col 2
SimpleProg...	line 9, col 13 to line 9, col 26

Evaluate Constant Expression

Expression:	Value:
-------------	--------

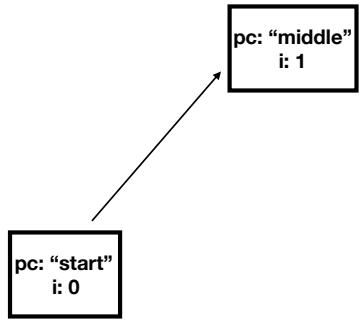
127-2

All Possible State Transitions

pc: "start"
i: 0

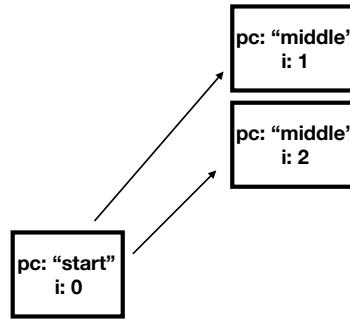
128-1

All Possible State Transitions



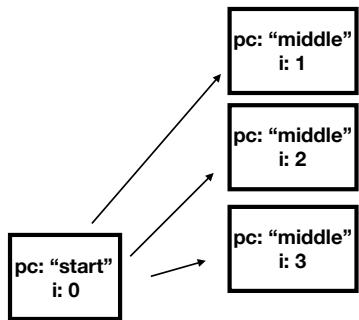
128-2

All Possible State Transitions



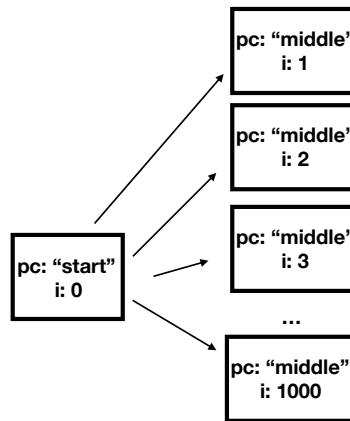
128-3

All Possible State Transitions



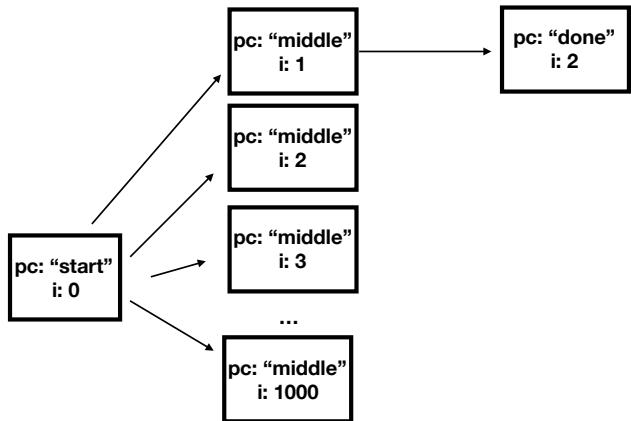
128-4

All Possible State Transitions



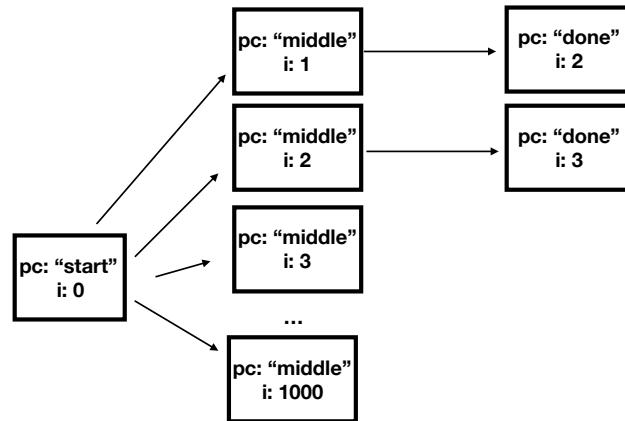
128-5

All Possible State Transitions



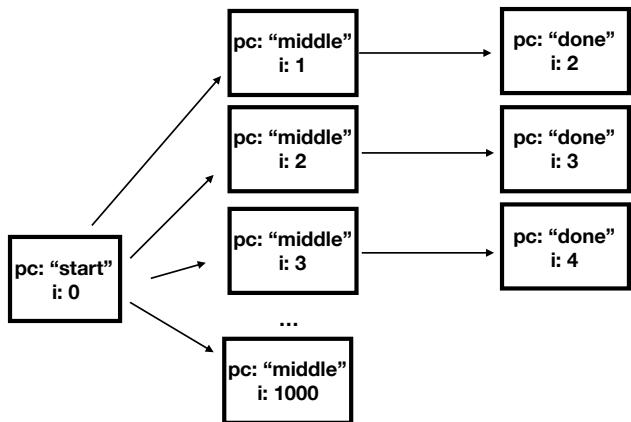
128-6

All Possible State Transitions



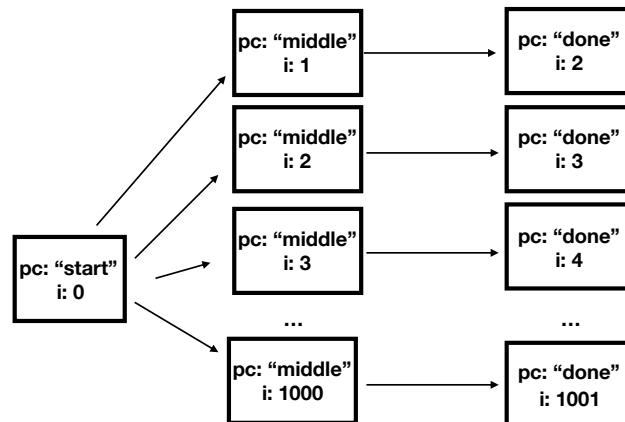
128-7

All Possible State Transitions



128-8

All Possible State Transitions



128-9

Sub-definitions

When programming, we don't stuff all our code into a single subroutine. We split it up!
TLA+ allows for the same thing.

```
Next ==  
  \/\ /\ pc = "start"  
  /\ i' \in 1..1000  
  /\ pc' = "middle"  
  \/\ /\ pc = "middle"  
  /\ i' = i + 1  
  /\ pc' = "done"
```

129-1

Sub-definitions

When programming, we don't stuff all our code into a single subroutine. We split it up!
TLA+ allows for the same thing.

```
Next ==  
  \/\ /\ pc = "start"  
  /\ i' \in 1..1000  
  /\ pc' = "middle"  
  \/\ /\ pc = "middle"  
  /\ i' = i + 1  
  /\ pc' = "done"
```

129-2

Sub-definitions

When programming, we don't stuff all our code into a single subroutine. We split it up!
TLA+ allows for the same thing.

```
Next ==  
  \/\ /\ pc = "start"  
  /\ i' \in 1..1000  
  /\ pc' = "middle"  
  \/\ /\ pc = "middle"  
  /\ i' = i + 1  
  /\ pc' = "done"
```

129-3

Sub-definitions

When programming, we don't stuff all our code into a single subroutine. We split it up!
TLA+ allows for the same thing.

```
Next ==  
  \/\ /\ pc = "start"  
  /\ i' \in 1..1000  
  /\ pc' = "middle"  
  \/\ /\ pc = "middle"  
  /\ i' = i + 1  
  /\ pc' = "done"  
  
Pick ==  
  /\ pc = "start"  
  /\ i' \in 1..1000  
  /\ pc' = "middle"
```

129-4

Sub-definitions

When programming, we don't stuff all our code into a single subroutine. We split it up!
TLA+ allows for the same thing.

```
Next ==  
  /\ /\ pc = "start"  
  /\ i' \in 1..1000  
  /\ pc' = "middle"  
  /\ /\ pc = "middle"  
  /\ i' = i + 1  
  /\ pc' = "done"  
  
Pick ==  
  /\ pc = "start"  
  /\ i' \in 1..1000  
  /\ pc' = "middle"  
  
Add ==  
  /\ pc = "middle"  
  /\ i' = i + 1  
  /\ pc' = "done"
```

129-5

Sub-definitions

When programming, we don't stuff all our code into a single subroutine. We split it up!
TLA+ allows for the same thing.

```
Next ==  
  /\ /\ pc = "start"  
  /\ i' \in 1..1000  
  /\ pc' = "middle"  
  /\ /\ pc = "middle"  
  /\ i' = i + 1  
  /\ pc' = "done"  
  
Pick ==  
  /\ pc = "start"  
  /\ i' \in 1..1000  
  /\ pc' = "middle"  
  
Add ==  
  /\ pc = "middle"  
  /\ i' = i + 1  
  /\ pc' = "done"  
  
Next == Pick \/ Add
```

129-6

```
EXTENDS Integers  
VARIABLES i, pc  
  
Init == (pc="start") /\ (i=0)  
  
Pick ==  
  /\ pc = "start"  
  /\ i' \in 1..1000  
  /\ pc' = "middle"  
  
Add ==  
  /\ pc = "middle"  
  /\ i' = i + 1  
  /\ pc' = "done"  
  
Next == Pick \/ Add
```

specs/SimpleProgram2.tla

130

What to check?

Deadlock

Invariants

Formulas true in every **reachable** state.

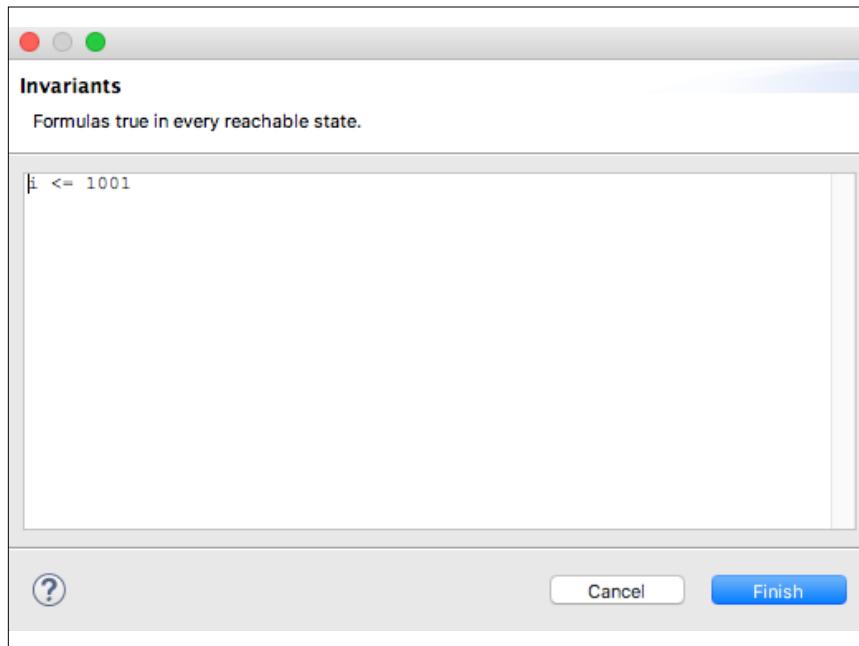
Add

Edit

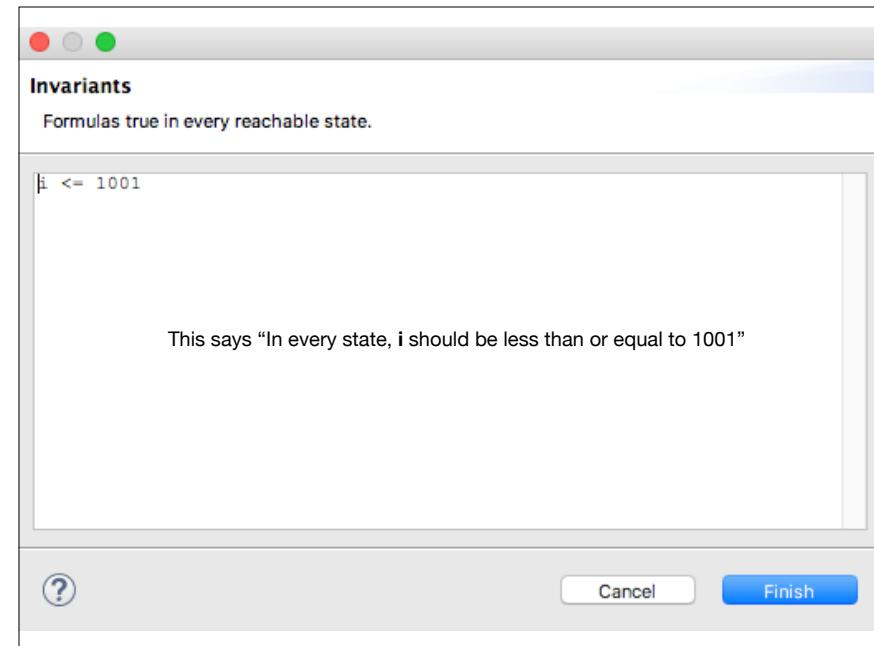
Remove

Properties

131



132-1



132-2

Run the model again,
everything should still be
fine

133

Change the Add action

```
Add ==  
  ∧ pc = "middle"  
  ∧ i' = i + 2  
  ∧ pc' = "done"
```

134-1

Change the Add action

```
Add ==  
  ∧ pc = "middle"  
  ∧ i' = i + 2  
  ∧ pc' = "done"
```

134-2

TLC Errors

Model_1

Invariant $i \leq 1001$ is violated.

Error-Trace Exploration

Error-Trace

Name	Value
Initial predicate	State (num = 1) i: 0 pc: "start"
Action line 10, col 3 to line 12, col 19 of module Simple...	State (num = 2) i: 1000 pc: "middle"
Action line 15, col 3 to line 17, col 17 of module Simple...	State (num = 3) i: 1002 pc: "done"

135-1

TLC Errors

Model_1

Invariant $i \leq 1001$ is violated.

Error-Trace Exploration

Error-Trace

Name	Value
Initial predicate	State (num = 1) i: 0 pc: "start"
Action line 10, col 3 to line 12, col 19 of module Simple...	State (num = 2) i: 1000 pc: "middle"
Action line 15, col 3 to line 17, col 17 of module Simple...	State (num = 3) i: 1002 pc: "done"

135-2

135-3

Error Trace

Error-Trace Exploration	
Error-Trace	
Name	Value
▽ ▲ <Initial predicate>	State (num = 1)
■ i	0
■ pc	"start"
▽ ▲ <Action line 10, col 3 to line 12, col 19 of module Simple...>	State (num = 2)
■ i	1000
■ pc	"middle"
▽ ▲ <Action line 15, col 3 to line 17, col 17 of module Simple...>	State (num = 3)
■ i	1002
■ pc	"done"

136-1

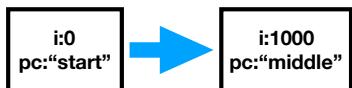
Error Trace

Error-Trace Exploration	
Error-Trace	
Name	Value
▽ ▲ <Initial predicate>	State (num = 1)
■ i	0
■ pc	"start"
▽ ▲ <Action line 10, col 3 to line 12, col 19 of module Simple...>	State (num = 2)
■ i	1000
■ pc	"middle"
▽ ▲ <Action line 15, col 3 to line 17, col 17 of module Simple...>	State (num = 3)
■ i	1002
■ pc	"done"

136-2

Error Trace

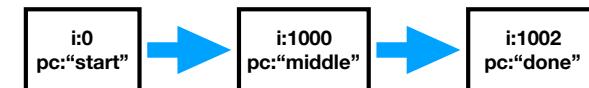
Error-Trace Exploration	
Error-Trace	
Name	Value
▽ ▲ <Initial predicate>	State (num = 1)
■ i	0
■ pc	"start"
▽ ▲ <Action line 10, col 3 to line 12, col 19 of module Simple...>	State (num = 2)
■ i	1000
■ pc	"middle"
▽ ▲ <Action line 15, col 3 to line 17, col 17 of module Simple...>	State (num = 3)
■ i	1002
■ pc	"done"



136-3

Error Trace

Error-Trace Exploration	
Error-Trace	
Name	Value
▽ ▲ <Initial predicate>	State (num = 1)
■ i	0
■ pc	"start"
▽ ▲ <Action line 10, col 3 to line 12, col 19 of module Simple...>	State (num = 2)
■ i	1000
■ pc	"middle"
▽ ▲ <Action line 15, col 3 to line 17, col 17 of module Simple...>	State (num = 3)
■ i	1002
■ pc	"done"



136-4



137

Die Hard Variables

VARIABLES big, small

Die Hard Initial States

```
Init == (big = 0) /\ (small = 0)
```

139

Die Hard Next States

140-1

Die Hard Next States

1.We can always fill **small**, i.e. in the next state **small=3**

140-2

Die Hard Next States

1.We can always fill **small**, i.e. in the next state **small=3**

2.We can always fill **big**, i.e. in the next state **big = 5**

140-3

Die Hard Next States

1.We can always fill **small**, i.e. in the next state **small=3**

2.We can always fill **big**, i.e. in the next state **big = 5**

3.We can always empty **small**, i.e. in the next state **small=0**

140-4

Die Hard Next States

1.We can always fill **small**, i.e. in the next state **small=3**

2.We can always fill **big**, i.e. in the next state **big = 5**

3.We can always empty **small**, i.e. in the next state **small=0**

4.We can always empty **big**, i.e. in the next state **big=0**

140-5

Die Hard Next States

- 1.We can always fill **small**, i.e. in the next state **small=3**
- 2.We can always fill **big**, i.e. in the next state **big = 5**
- 3.We can always empty **small**, i.e. in the next state **small=0**
- 4.We can always empty **big**, i.e. in the next state **big=0**
- 5.We can always pour **small** into **big**

140-6

Die Hard Next States

- 1.We can always fill **small**, i.e. in the next state **small=3**
- 2.We can always fill **big**, i.e. in the next state **big = 5**
- 3.We can always empty **small**, i.e. in the next state **small=0**
- 4.We can always empty **big**, i.e. in the next state **big=0**
- 5.We can always pour **small** into **big**
6. We can always pour **big** into **small**

140-7

Die Hard Next States

```
FillSmall  
FillBig  
EmptySmall  
EmptyBig  
SmallToBig  
BigToSmall
```

141

FillSmall

142-1

FillSmall

```
FillSmall == /\ small' = 3
```

142-2

FillSmall

```
FillSmall == /\ small' = 3  
      /\ big' = big
```

142-3

EmptySmall

143-1

EmptySmall

```
EmptySmall == /\ small' = 0
```

143-2

EmptySmall

```
EmptySmall == /\ small' = 0  
          /\ big' = big
```

143-3

SmallToBig

- Two possible cases:
 1. There **is** room in **big** for all the water in **small**
 2. There **is not** room in **big** for all the water in **small**

144

SmallToBig

Case 1: Enough room in **big**

We put all the water from the **small** jug into the **big** jug, which has the effect of emptying the **small** jug

145-1

SmallToBig

Case 1: Enough room in **big**

We put all the water from the **small** jug into the **big** jug, which has the effect of emptying the **small** jug

```
/\ big' = big + small
```

145-2

SmallToBig

Case 1: Enough room in **big**

We put all the water from the **small** jug into the **big** jug, which has the effect of emptying the **small** jug

```
/\ big' = big + small  
/\ small' = 0
```

145-3

SmallToBig

Case 2: NOT enough room in **big**

We pour what we can from **small** into **big**. **big** gets filled by this action. **small** is emptied by however much was poured out

146-1

SmallToBig

Case 2: NOT enough room in **big**

We pour what we can from **small** into **big**. **big** gets filled by this action. **small** is emptied by however much was poured out

```
/\ big' = 5  
/\ small' = small - (5 - big)
```

146-2

SmallToBig

Case 2: NOT enough room in **big**

We pour what we can from **small** into **big**. **big** gets filled by this action. **small** is emptied by however much was poured out

```
/\ big' = 5  
/\ small' = small - (5 - big)
```

146-3

SmallToBig

Putting it all together

147-1

SmallToBig

Putting it all together

```
SmallToBig == IF big + small < 5
```

147-2

SmallToBig

Putting it all together

```
SmallToBig == IF big + small < 5  
    THEN /\ big' = big + small
```

147-3

SmallToBig

Putting it all together

```
SmallToBig == IF big + small < 5  
    THEN /\ big' = big + small  
        /\ small' = 0
```

147-4

SmallToBig

Putting it all together

```
SmallToBig == IF big + small < 5
    THEN /\ big' = big + small
        /\ small' = 0
    ELSE /\ small' = small - (5 - big)
```

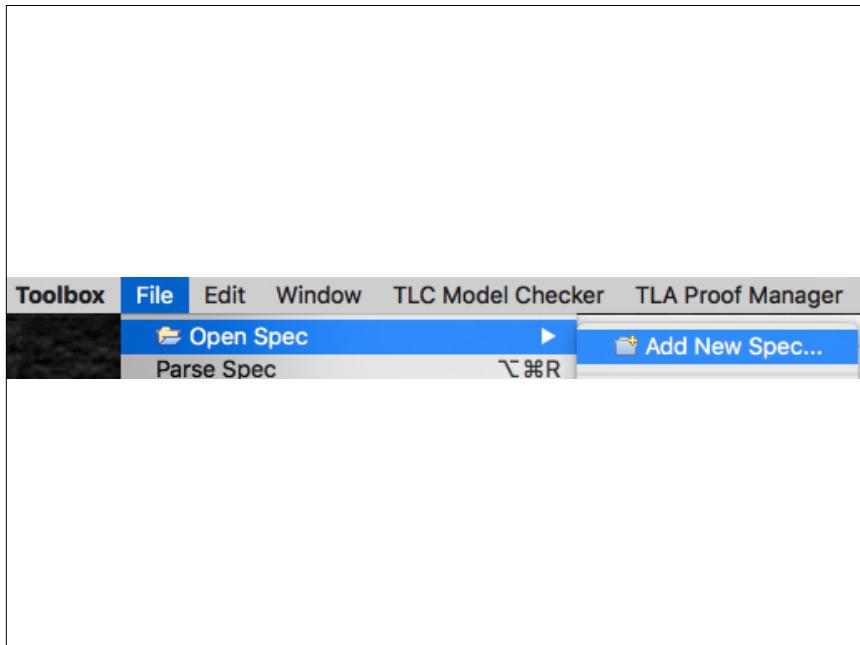
147-5

SmallToBig

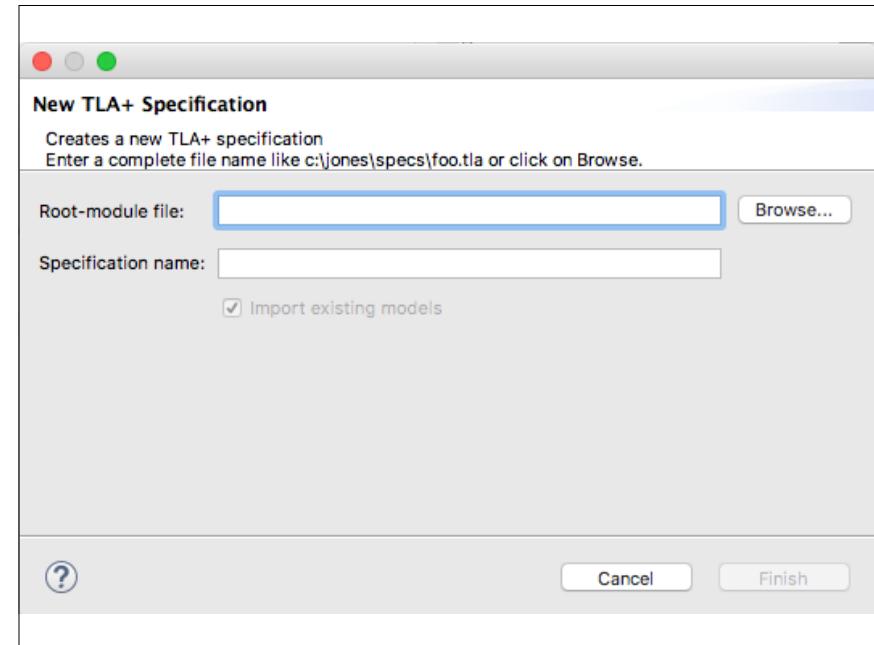
Putting it all together

```
SmallToBig == IF big + small < 5
    THEN /\ big' = big + small
        /\ small' = 0
    ELSE /\ small' = small - (5 - big)
        /\ big' = 5
```

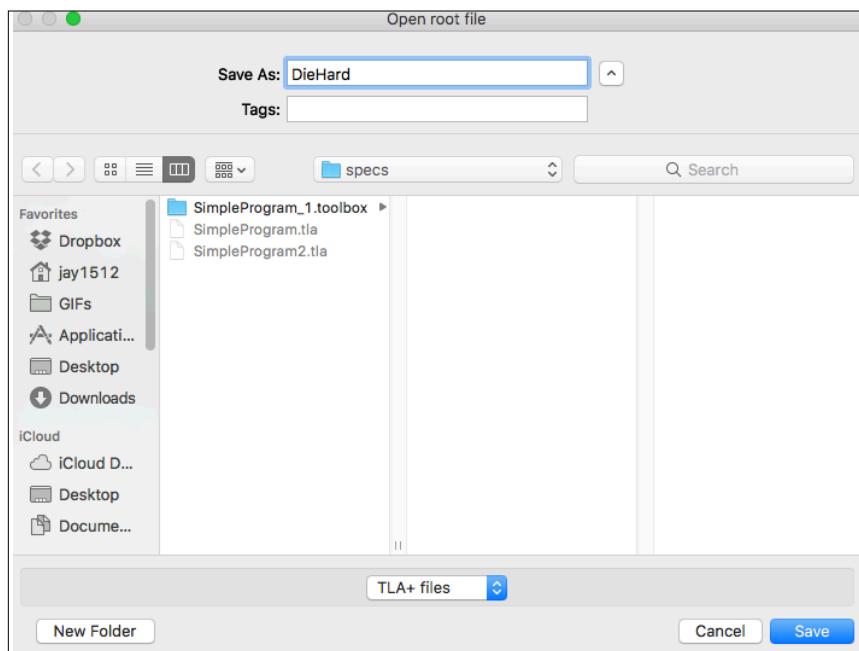
147-6



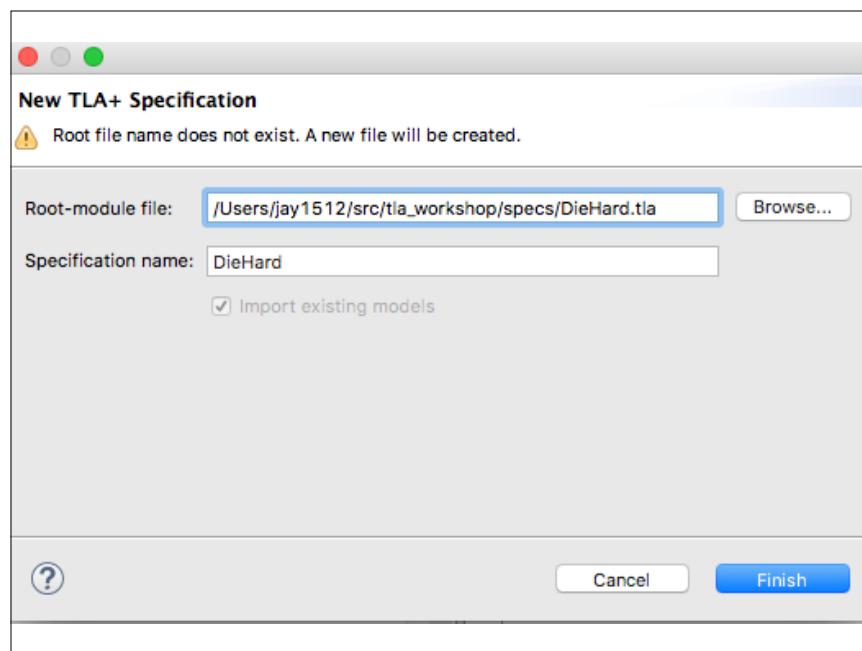
148



149



150



151

```
EXTENDS Integers

VARIABLES small, big

TypeOK == small \in 0..3
           big \in 0..5

Init == big = 0
       small = 0

FillSmall == small' = 3
             big' = big

FillBig ==

EmptySmall == small' = 0
              big' = big

EmptyBig ==

SmallToBig == IF big + small <= 5
                  THEN big' = big + small
                     small' = 0
                  ELSE big' = 5
                     small' = small - (5 - big)

BigToSmall ==

Next == \ FillSmall
        \ FillBig
        \ EmptySmall
        \ EmptyBig
        \ SmallToBig
        \ BigToSmall
```

152

```
----- MODULE DieHard -----
EXTENDS Integers

VARIABLES small, big

TypeOK == small \in 0..3
           big \in 0..5

Init == big = 0
       small = 0

FillSmall == small' = 3
             big' = big

FillBig ==

EmptySmall == small' = 0
              big' = big

EmptyBig ==

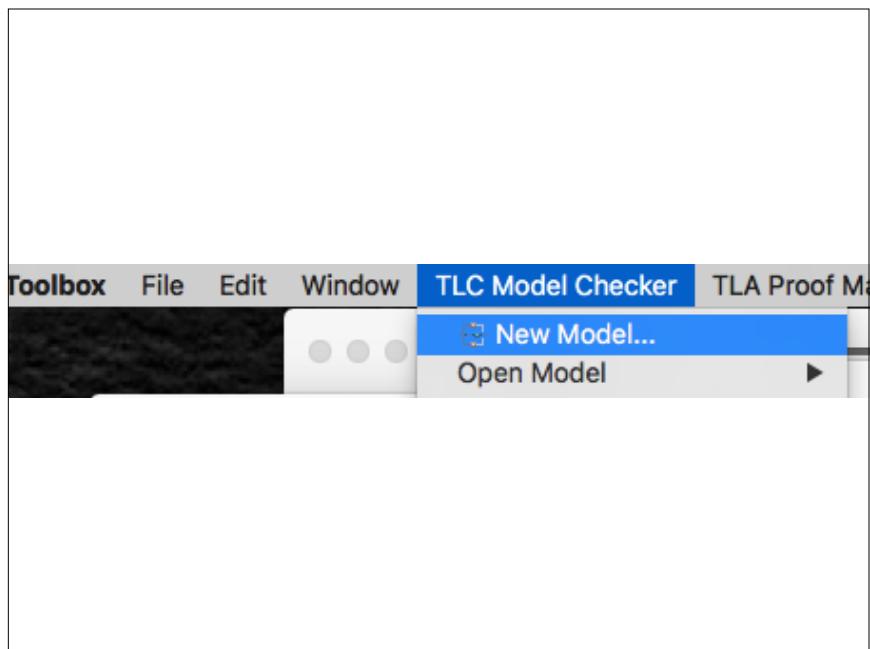
SmallToBig == IF big + small <= 5
                  THEN big' = big + small
                     small' = 0
                  ELSE big' = 5
                     small' = small - (5 - big)

BigToSmall ==

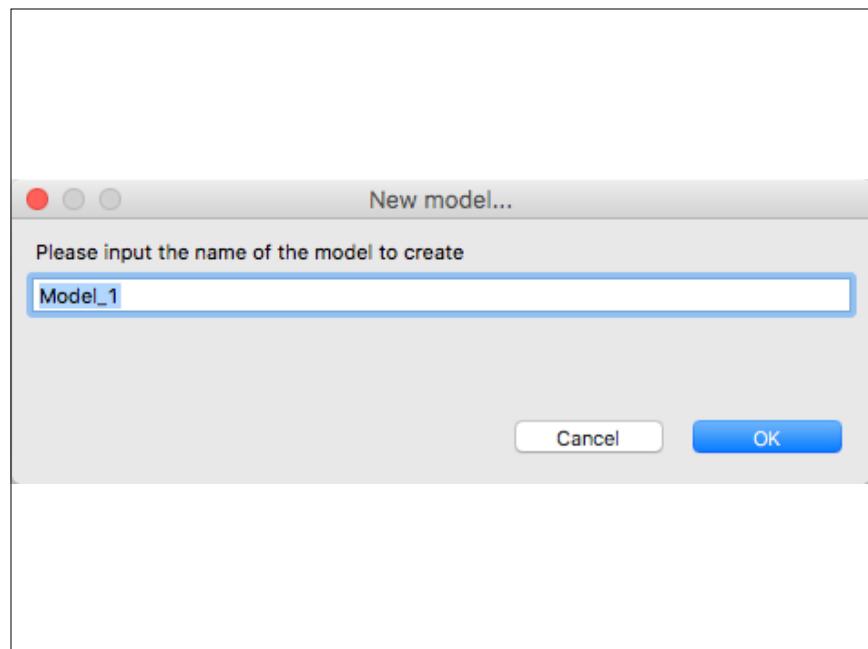
Next == \ FillSmall
        \ FillBig
        \ EmptySmall
        \ EmptyBig
        \ SmallToBig
        \ BigToSmall
```

specs/DieHard_incomplete.tla

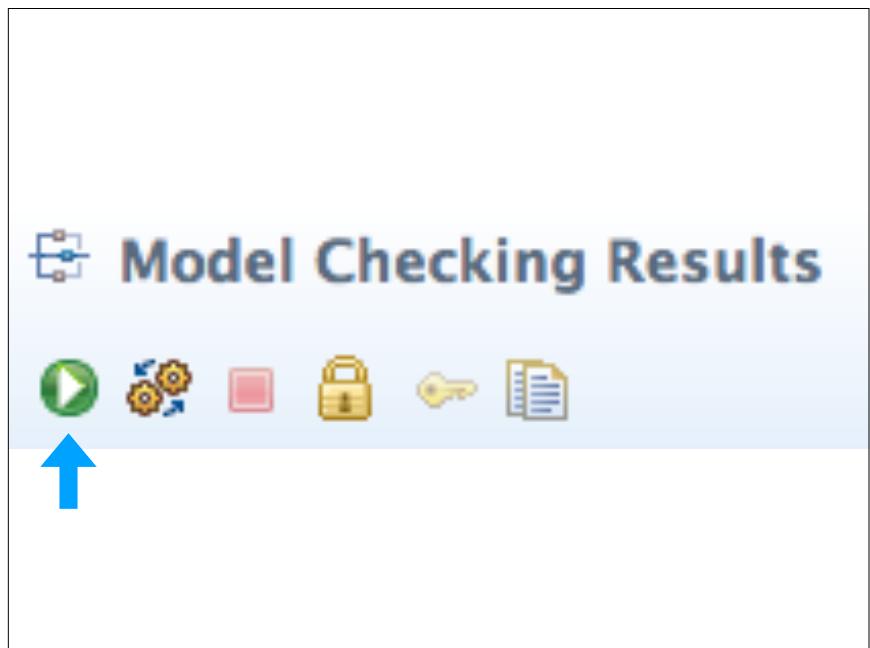
153



154



155



156

The screenshot shows the 'Model Checking Results' interface with the 'Statistics' section expanded. It displays state space progress and coverage details.

State space progress (click column header for graph)

Time	Diameter	States Found	Distinct States	Queue Size
2017-09-14 14:13...	10	97	16	0

Coverage at 2017-09-14 14:13:33

Module	Location	Count
DieHard	line 12, col 17 to line 12, col 26	16
DieHard	line 13, col 17 to line 13, col 28	16
DieHard	line 15, col 15 to line 15, col 24	16
DieHard	line 16, col 15 to line 16, col 28	16
DieHard	line 18, col 18 to line 18, col 27	16
DieHard	line 19, col 18 to line 19, col 29	16
DieHard	line 21, col 16 to line 21, col 25	16

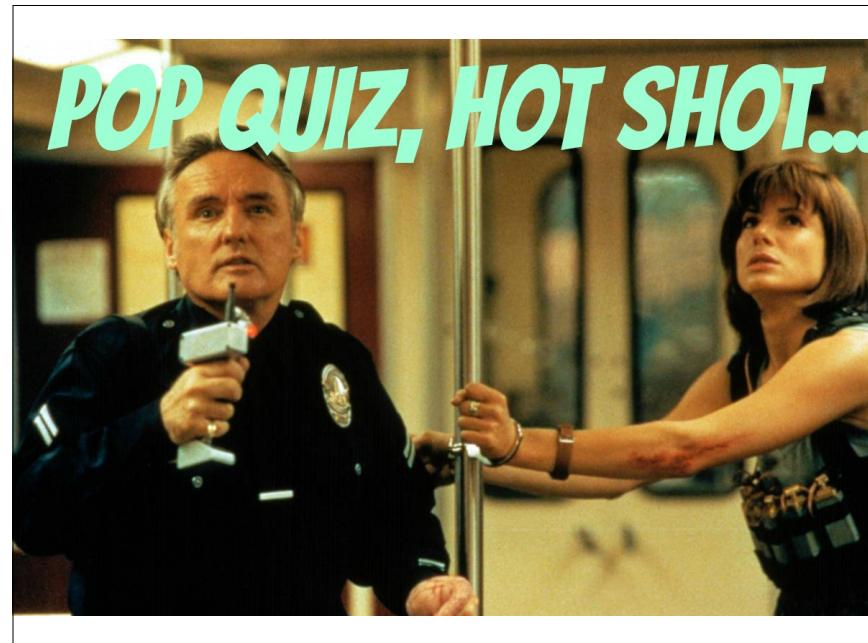
157-1

Statistics				
State space progress (click column header for graph)				
Time	Diameter	States Found	Distinct States	Queue Size
2017-09-14 14:13...	10	97	16	0
Module	Location	Count		
DieHard	line 12, col 17 to line 12, col 26	16		
DieHard	line 13, col 17 to line 13, col 28	16		
DieHard	line 15, col 15 to line 15, col 24	16		
DieHard	line 16, col 15 to line 16, col 28	16		
DieHard	line 18, col 18 to line 18, col 27	16		
DieHard	line 19, col 18 to line 19, col 29	16		
DieHard	line 21, col 16 to line 21, col 25	16		

157-2

Statistics				
State space progress (click column header for graph)				
Time	Diameter	States Found	Distinct States	Queue Size
2017-09-14 14:13...	10	97	16	0
Module	Location	Count		
DieHard	line 12, col 17 to line 12, col 26	16		
DieHard	line 13, col 17 to line 13, col 28	16		
DieHard	line 15, col 15 to line 15, col 24	16		
DieHard	line 16, col 15 to line 16, col 28	16		
DieHard	line 18, col 18 to line 18, col 27	16		
DieHard	line 19, col 18 to line 19, col 29	16		
DieHard	line 21, col 16 to line 21, col 25	16		

157-3



```

EXTENDS Integers
VARIABLES small, big

TypeOK ==  $\wedge$  small  $\in$  0..3
           $\wedge$  big  $\in$  0..5

Init ==  $\wedge$  big = 0
         $\wedge$  small = 0

FillSmall ==  $\wedge$  small' = 3
              $\wedge$  big' = big

FillBig ==  $\wedge$  big' = 5
              $\wedge$  small' = small

EmptySmall ==  $\wedge$  small' = 0
               $\wedge$  big' = big

EmptyBig ==  $\wedge$  big' = 0
               $\wedge$  small' = small

SmallToBig == IF big + small <= 5
              THEN  $\wedge$  big' = big + small
                      $\wedge$  small' = 0
              ELSE  $\wedge$  big' = 5
                      $\wedge$  small' = small - (5 - big)

BigToSmall == IF big + small <= 3
              THEN  $\wedge$  big' = 0
                      $\wedge$  small' = big + small
              ELSE  $\wedge$  big' = small - (3 - big)
                      $\wedge$  small' = 3

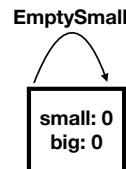
Next ==  $\vee$  FillSmall
        $\vee$  FillBig
        $\vee$  EmptySmall
        $\vee$  EmptyBig
        $\vee$  SmallToBig
        $\vee$  BigToSmall
  
```

158

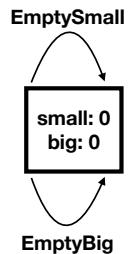
159



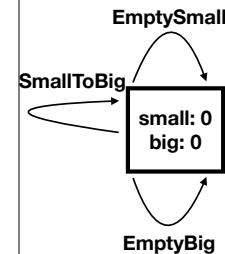
160-1



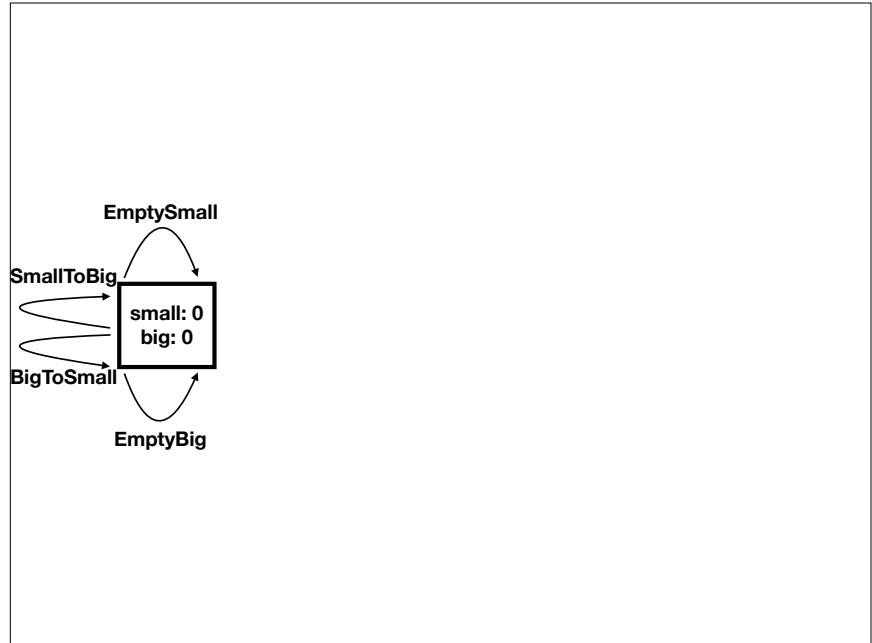
160-2



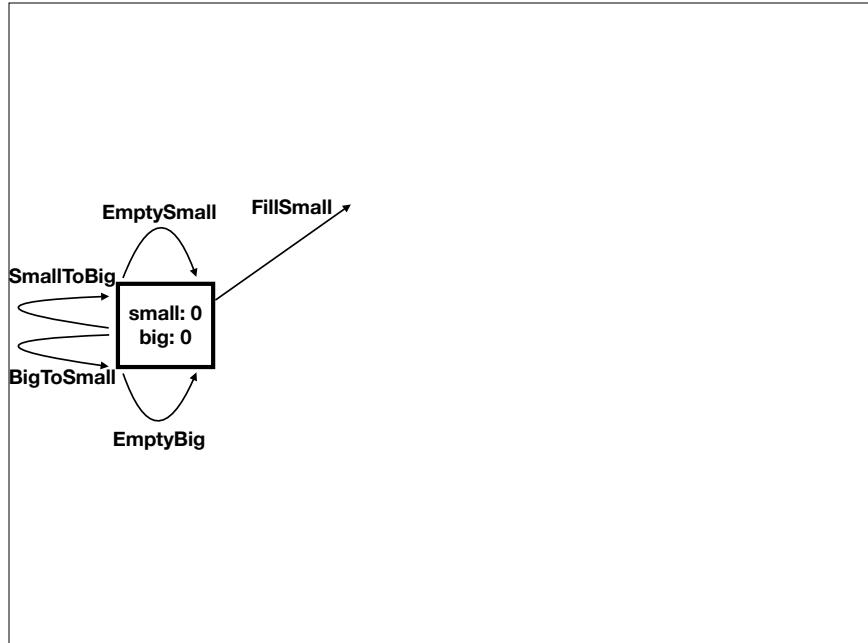
160-3



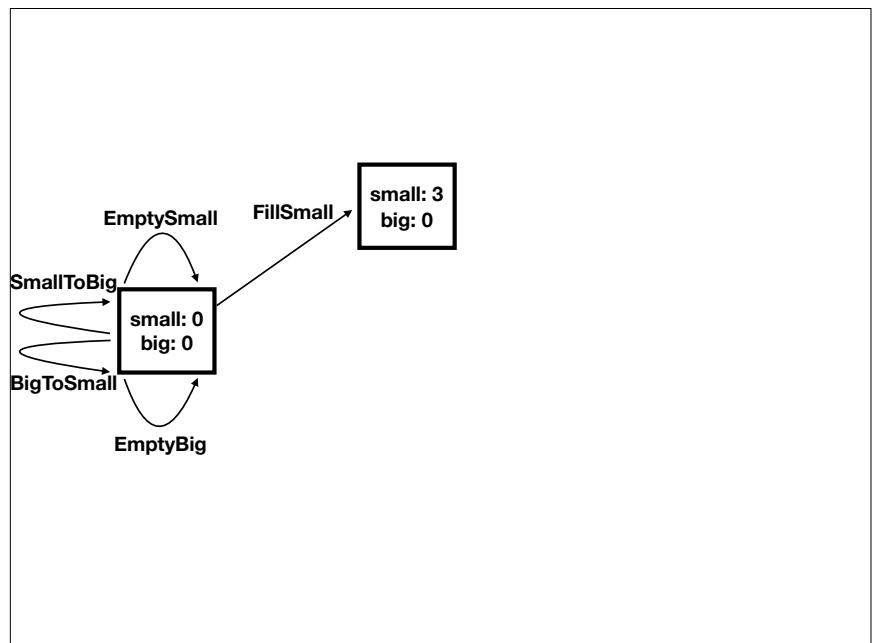
160-4



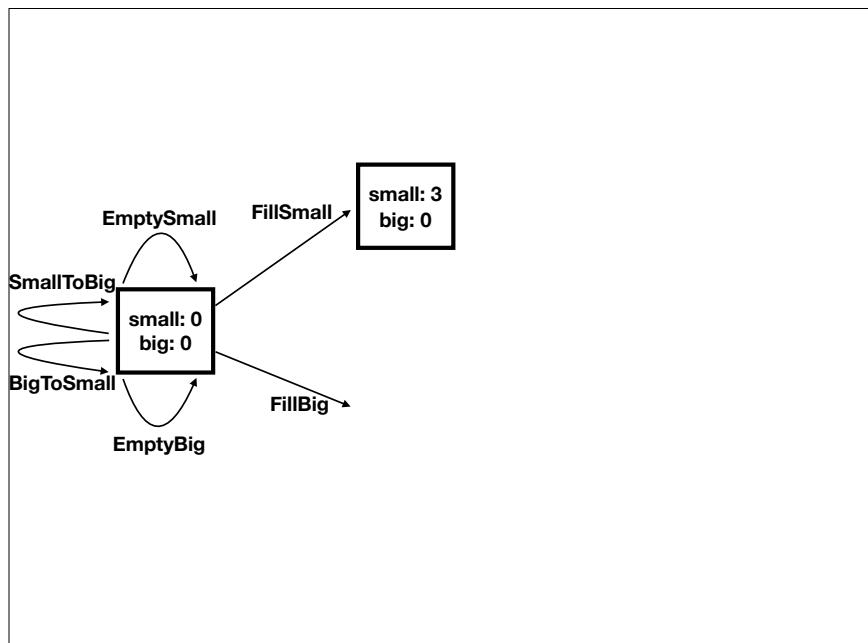
160-5



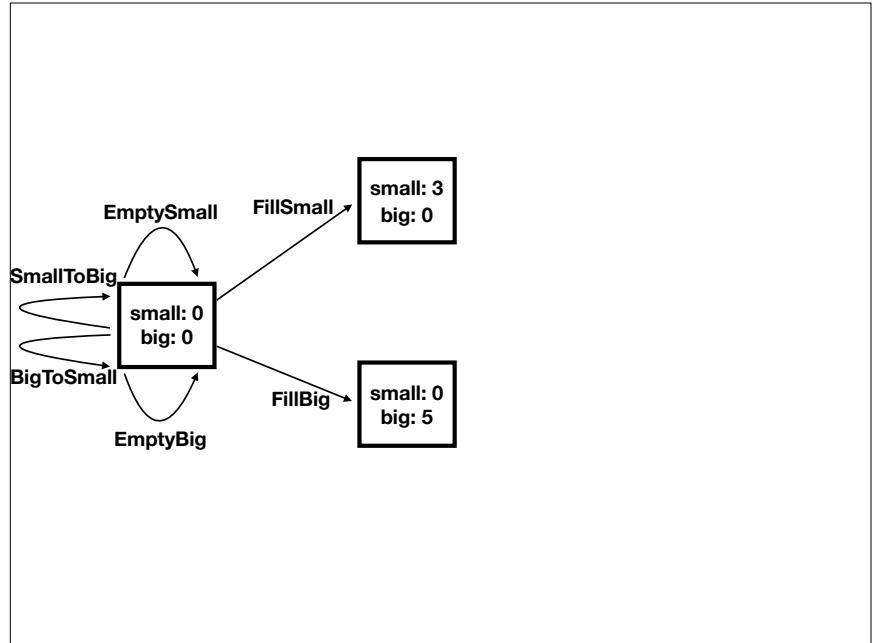
160-6



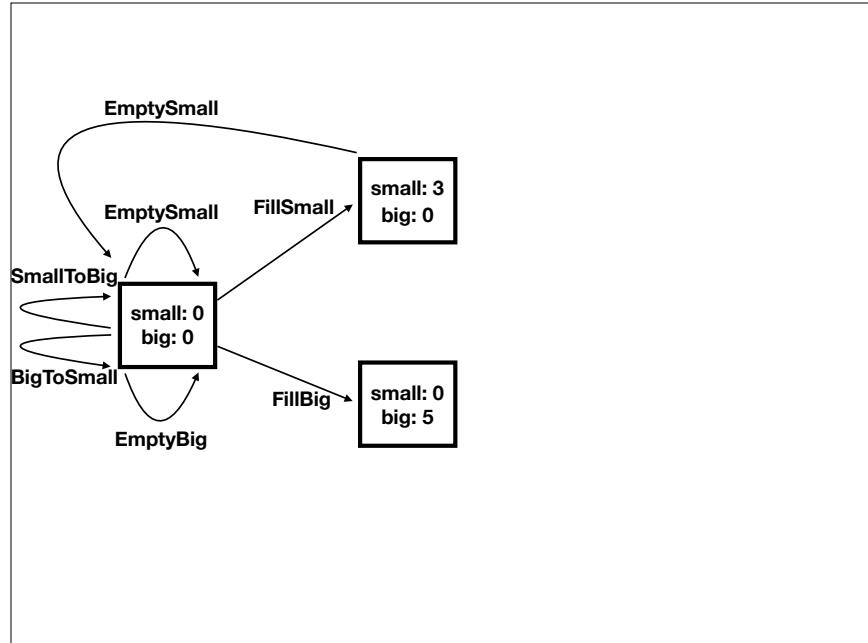
160-7



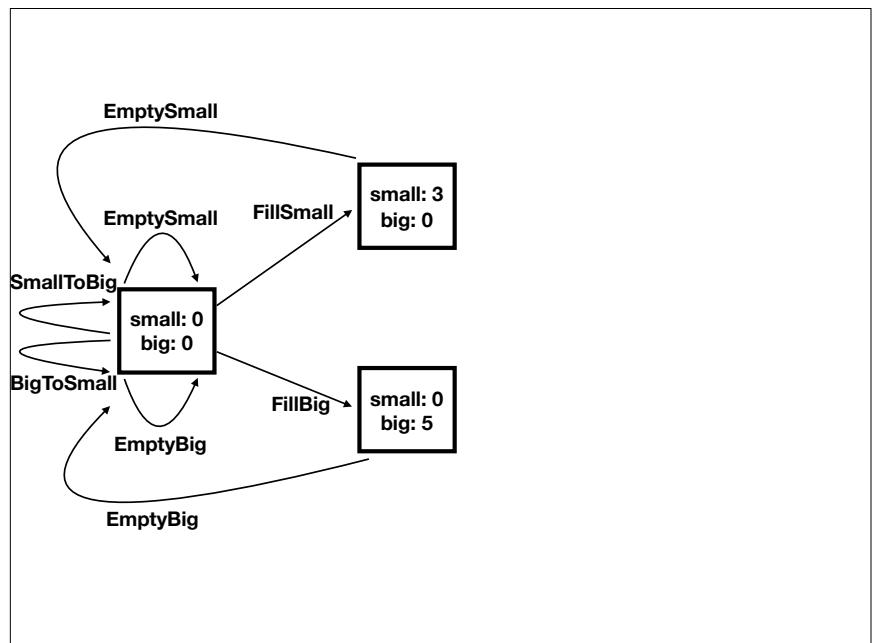
160-8



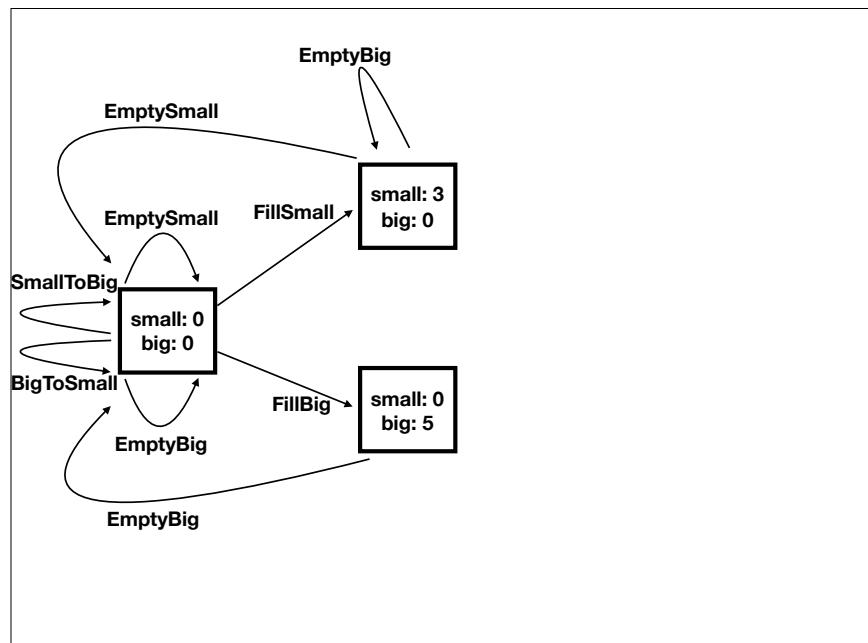
160-9



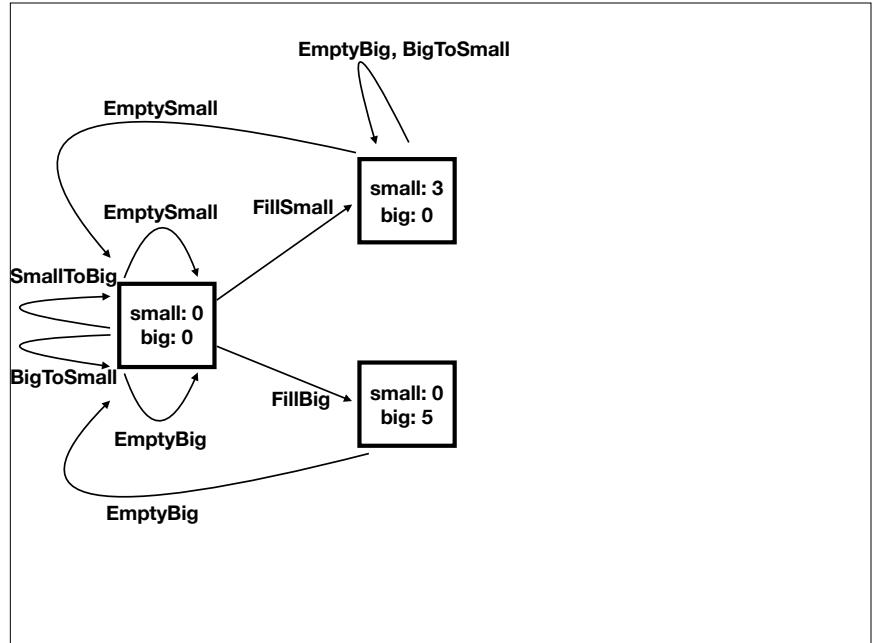
160-10



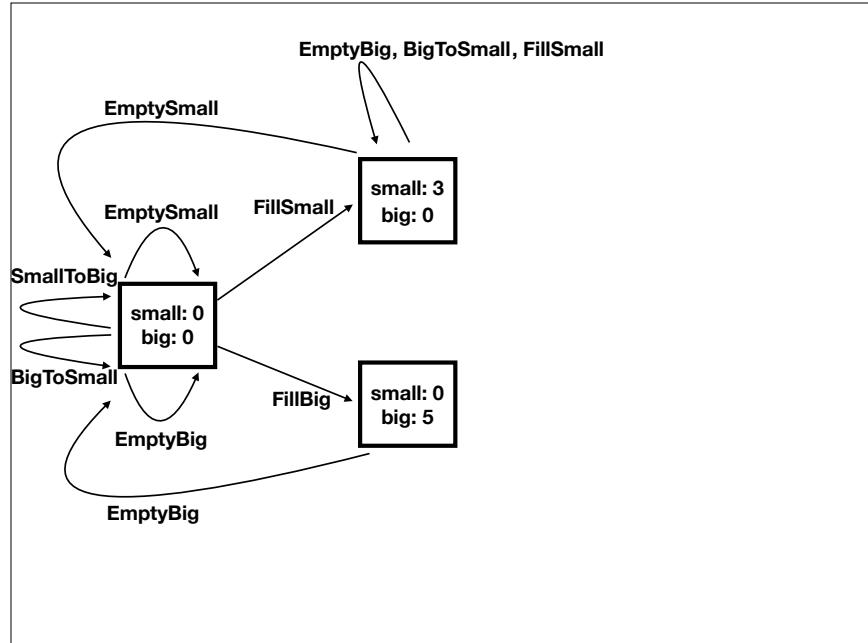
160-11



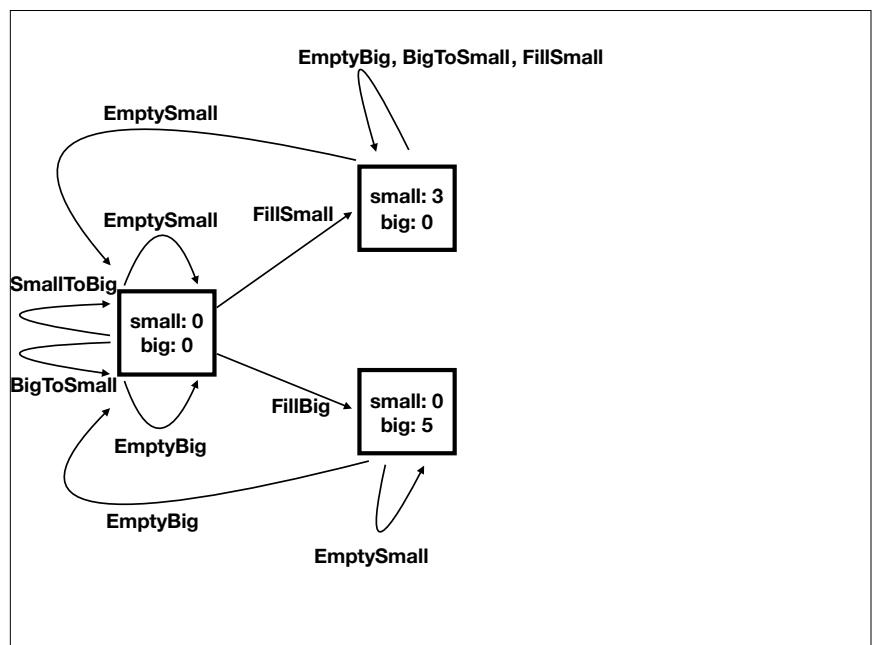
160-12



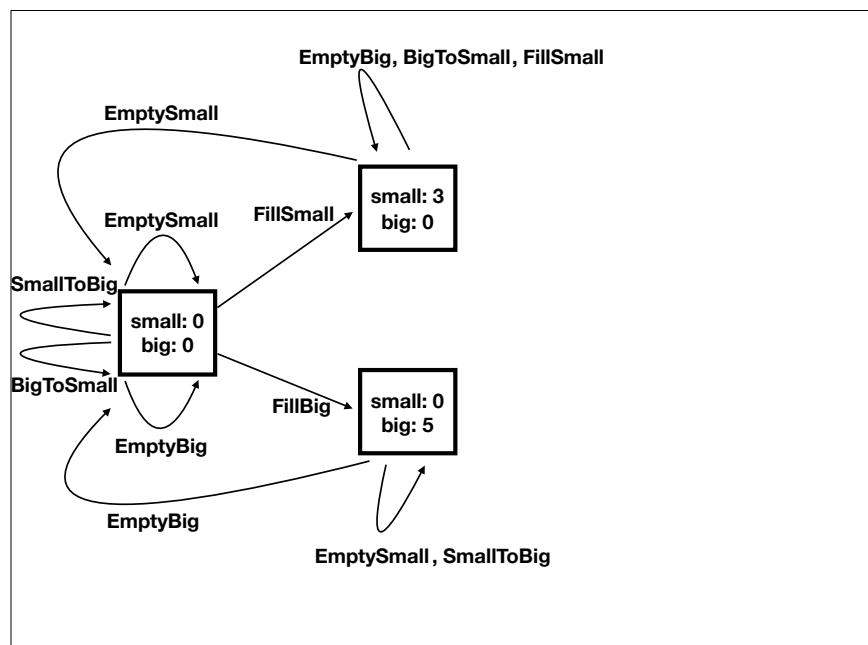
160-13



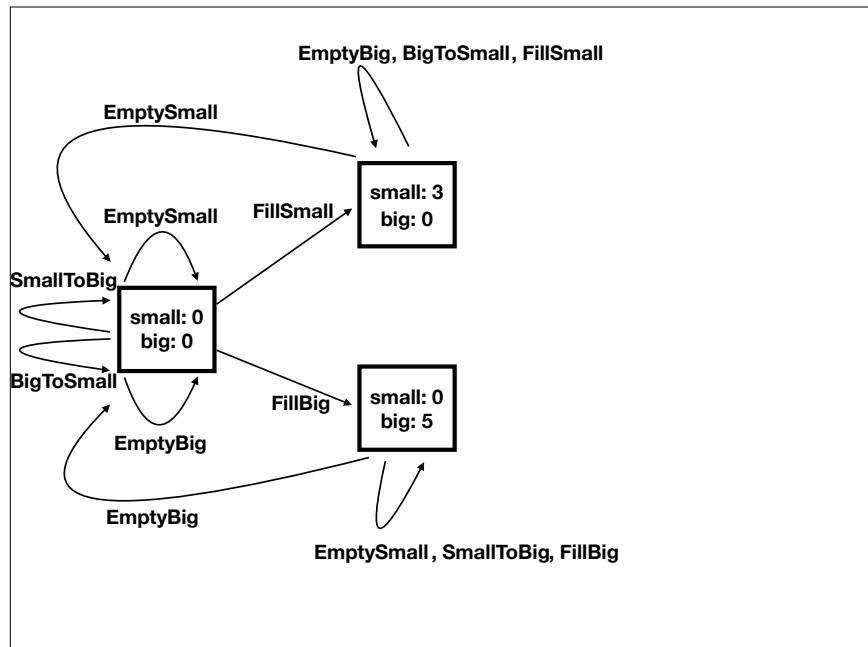
160-14



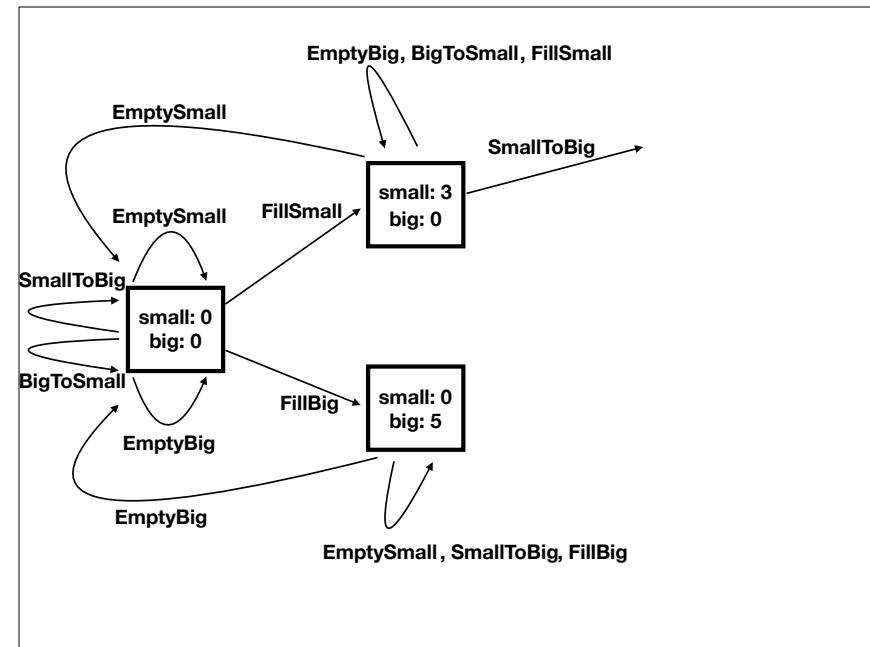
160-15



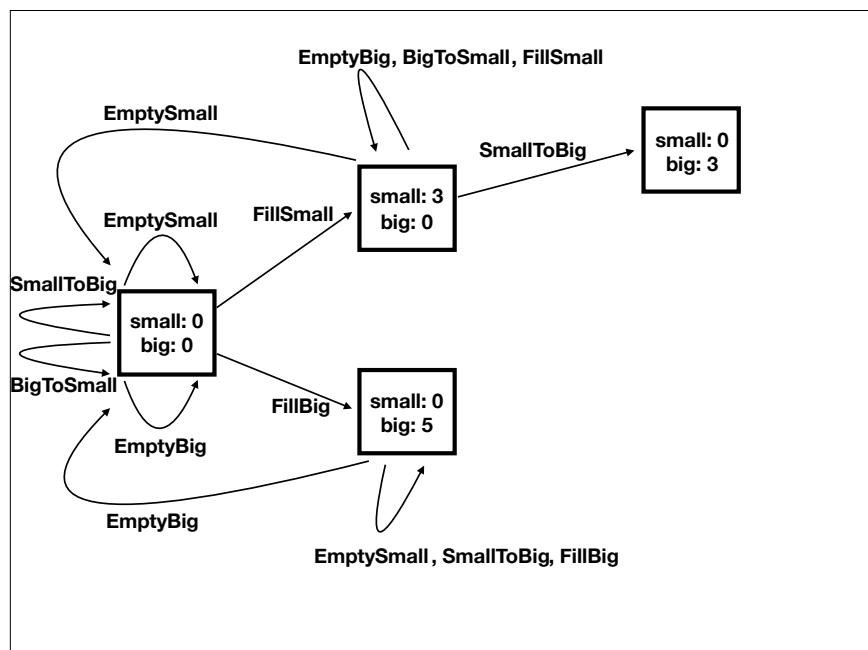
160-16



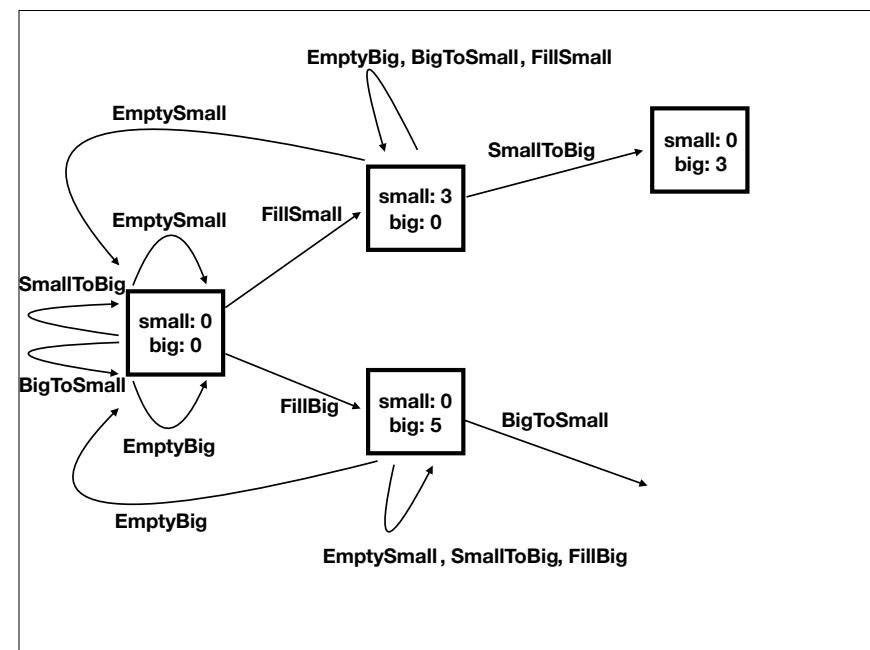
160-17



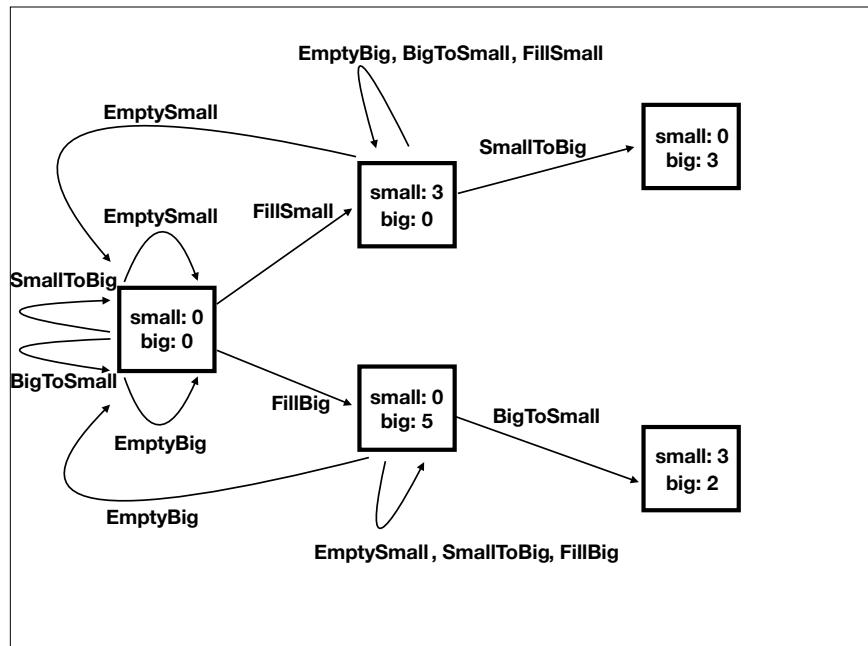
160-18



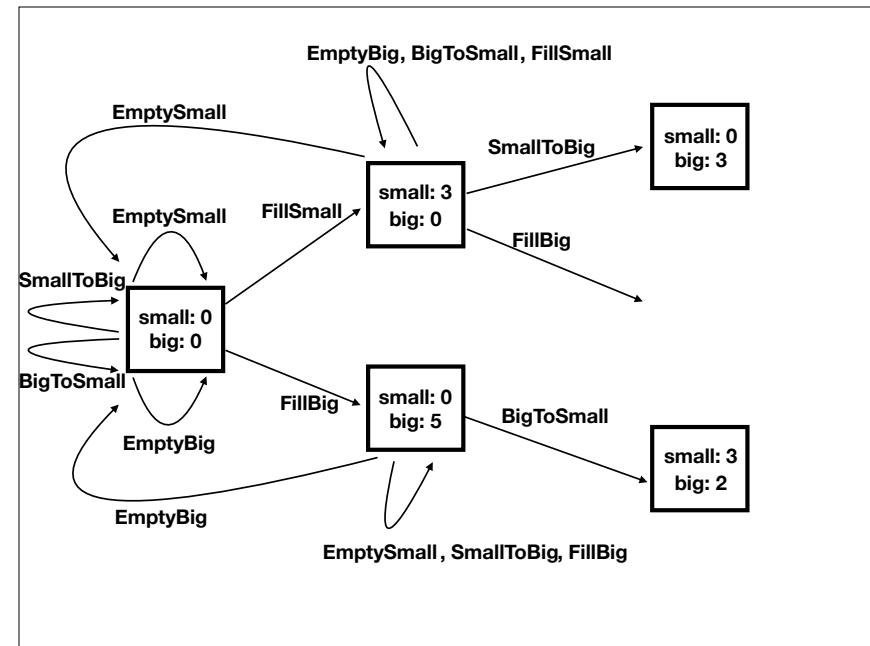
160-19



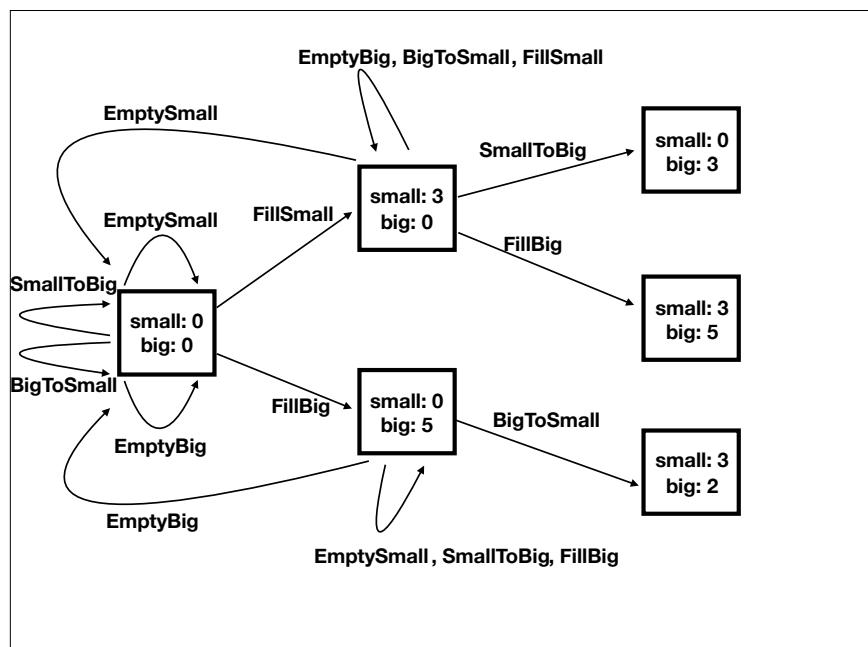
160-20



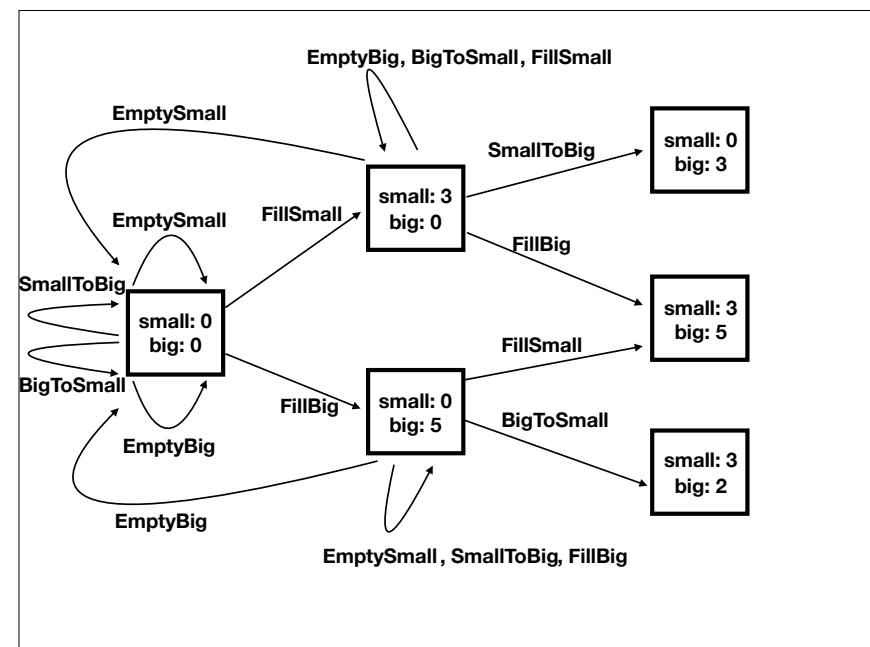
160-21



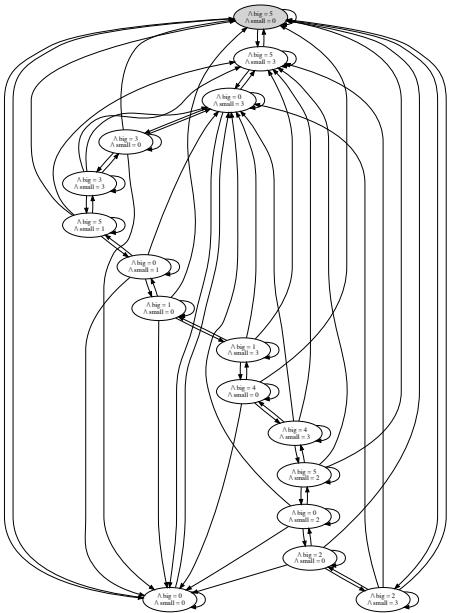
160-22



160-23



160-24



161

Die Hard

Die Hard

- Given the restrictions determined by our formula, TLC has found a total of 16 **unique** states.

162-1

Die Hard

- Given the restrictions determined by our formula, TLC has found a total of 16 **unique** states.
- Remember that a state is a particular assignment of **values** to our **variables**. Thus, 16 unique and valid combinations of values assigned to **big** and **small**.

162-2

162-3

Die Hard

- Given the restrictions determined by our formula, TLC has found a total of 16 **unique** states.
- Remember that a state is a particular assignment of **values** to our **variables**. Thus, 16 unique and valid combinations of values assigned to **big** and **small**.
- and so what? What does this get us? Who cares?

162-4

Invariants

- It's time to explore the true power of TLA+ and TLC

163-1

Invariants

- It's time to explore the true power of TLA+ and TLC
- An invariant is “**something that should always be true**”

163-2

163-3

Invariants

- It's time to explore the true power of TLA+ and TLC
- An invariant is “**something that should always be true**”
 - Or “**something that should never occur**”

163-4

Invariants

- It's time to explore the true power of TLA+ and TLC
- An invariant is “**something that should always be true**”
 - Or “**something that should never occur**”
- TLC excels at generating every possible state that can occur given your formula

163-5

Invariants

- It's time to explore the true power of TLA+ and TLC
- An invariant is “**something that should always be true**”
 - Or “**something that should never occur**”
- TLC excels at generating every possible state that can occur given your formula
- You can also tell it to check that a certain thing is **always** true (or **never** true) in any of those states, i.e. check invariants

163-6

What to check?

Deadlock

Invariants

Formulas true in every reachable state.

--

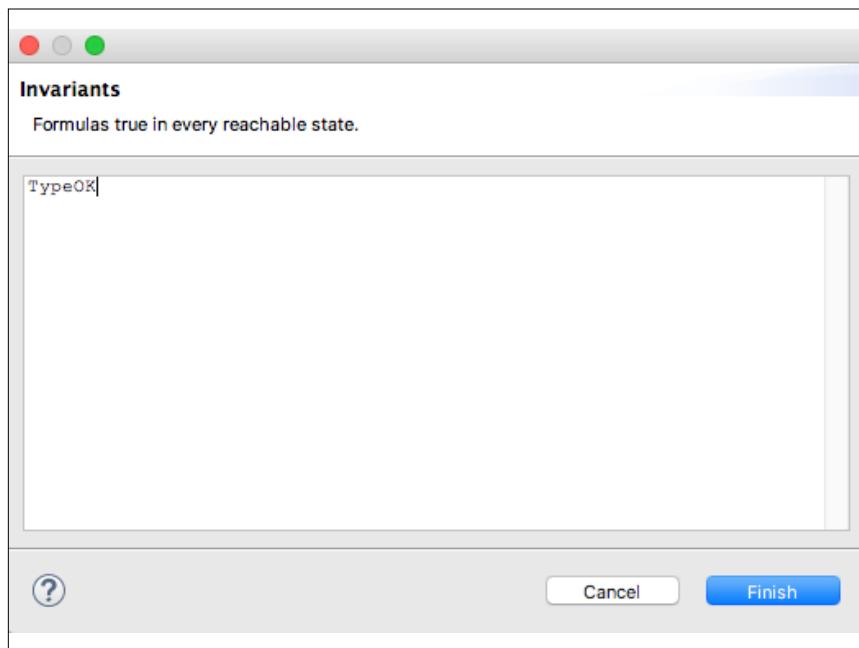
Add

Edit

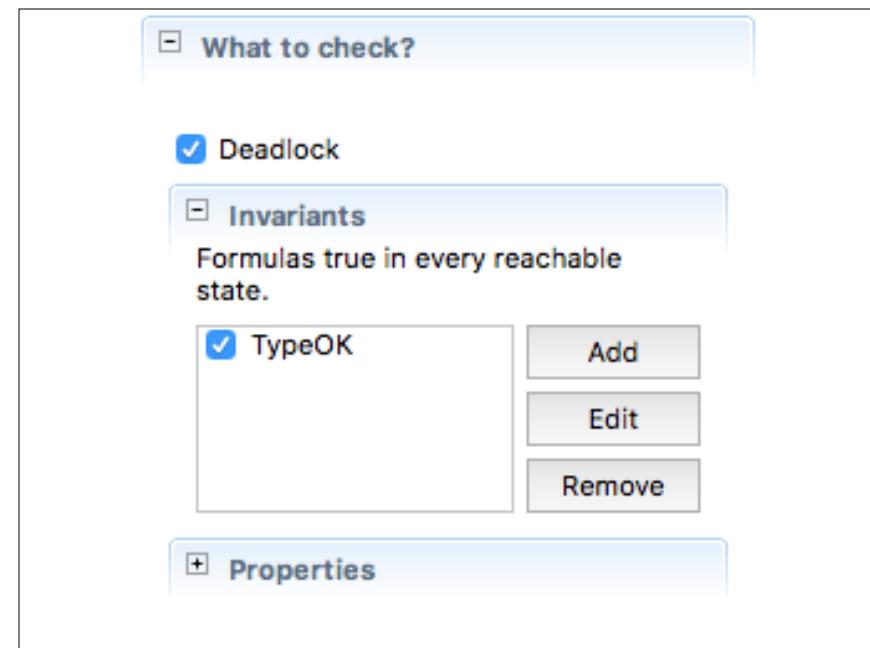
Remove

Properties

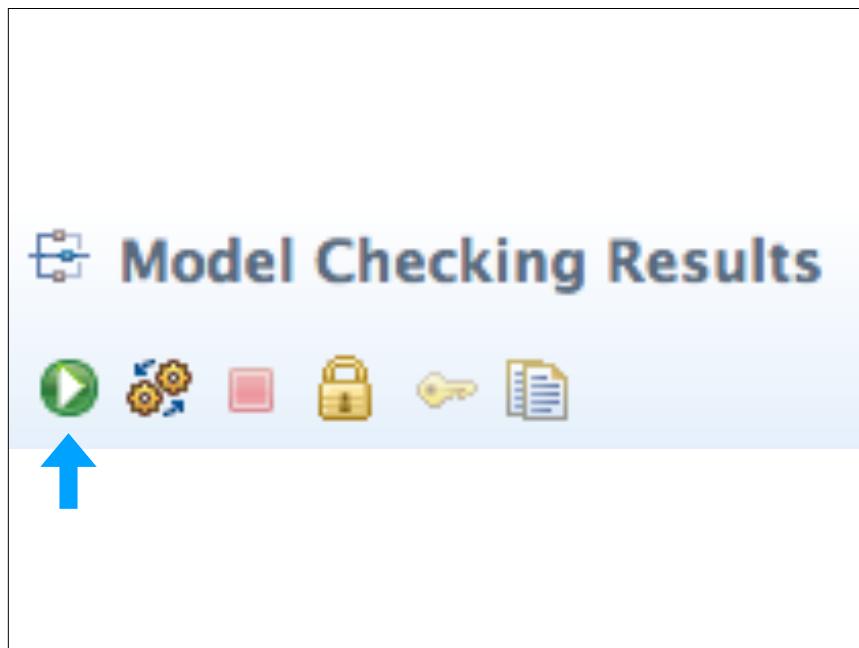
164



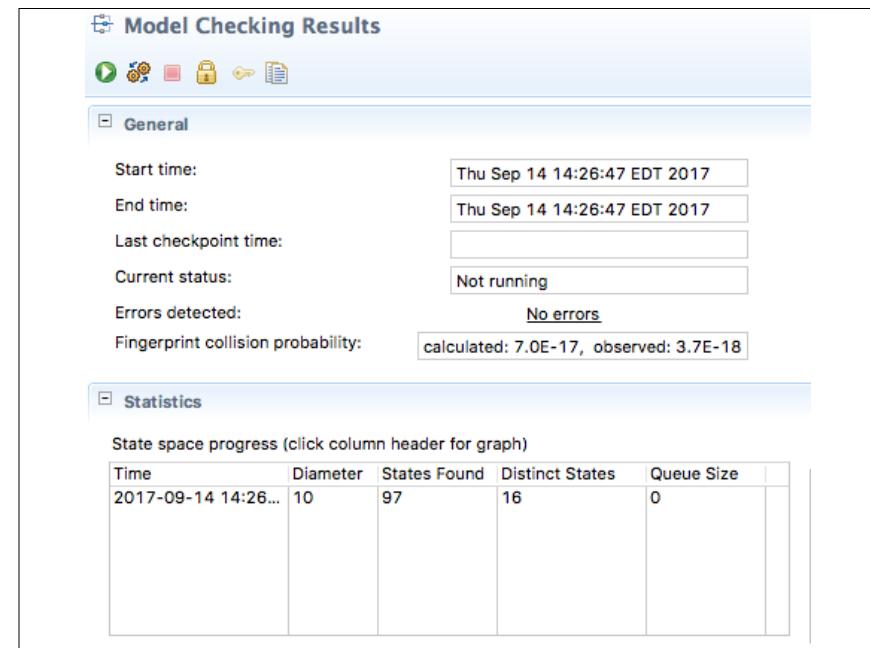
165



166



167



168

```
Init ==  $\wedge$  big = 6  
       $\wedge$  small = 0
```

169



170

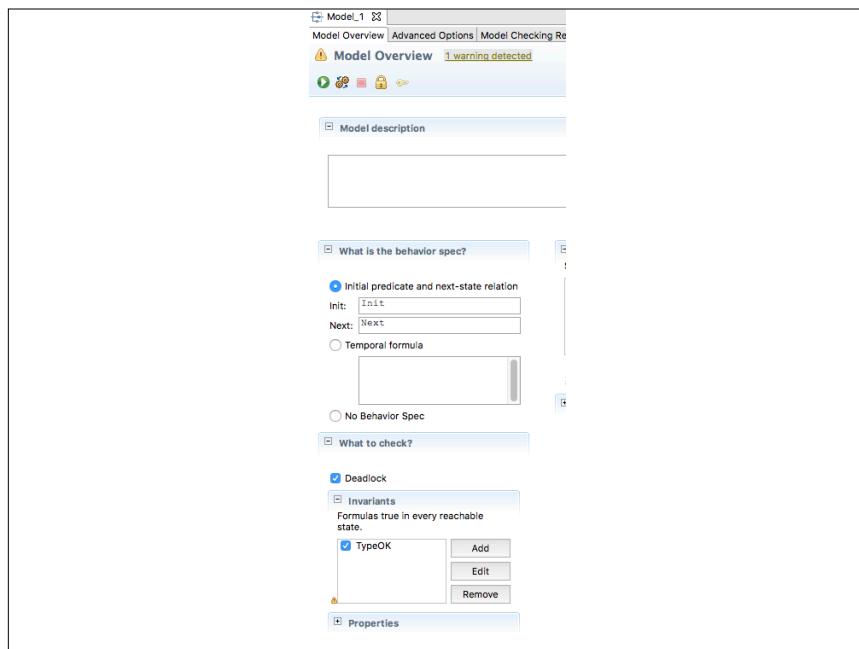
TypeOK

```
TypeOK ==  $\wedge$  small  $\in$  0..3  
       $\wedge$  big  $\in$  0..5
```

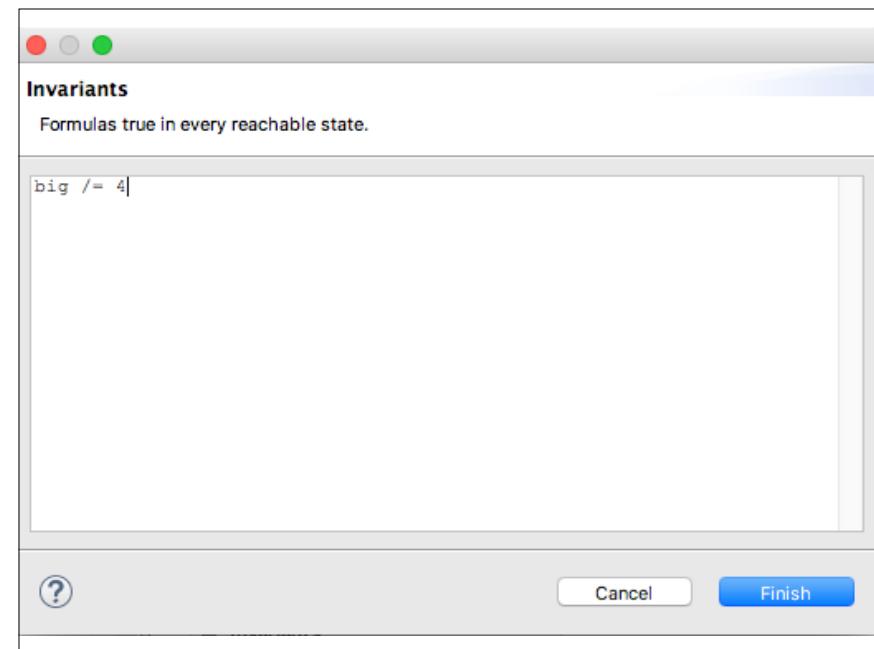
171

```
Init ==  $\wedge$  big = 5  
       $\wedge$  small = 0
```

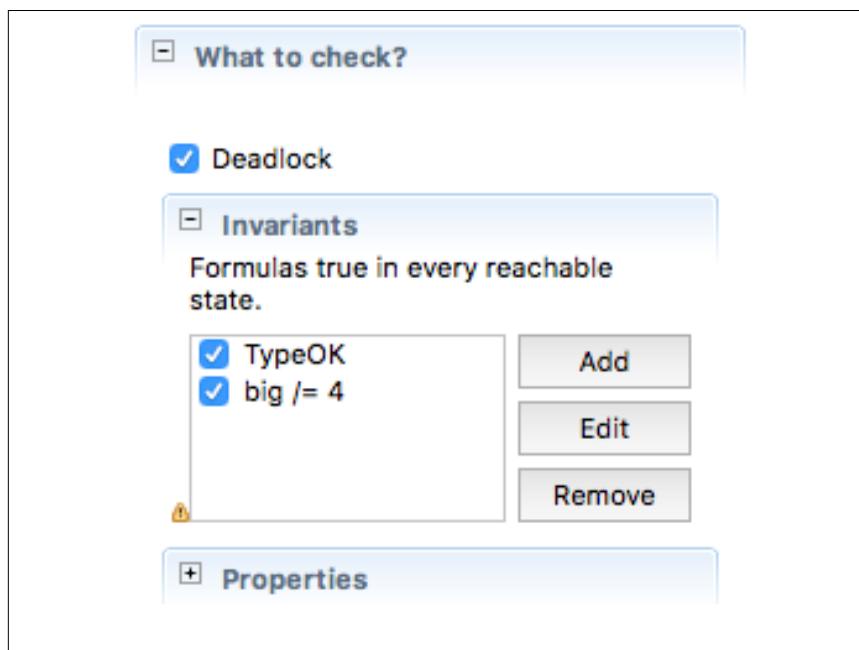
172



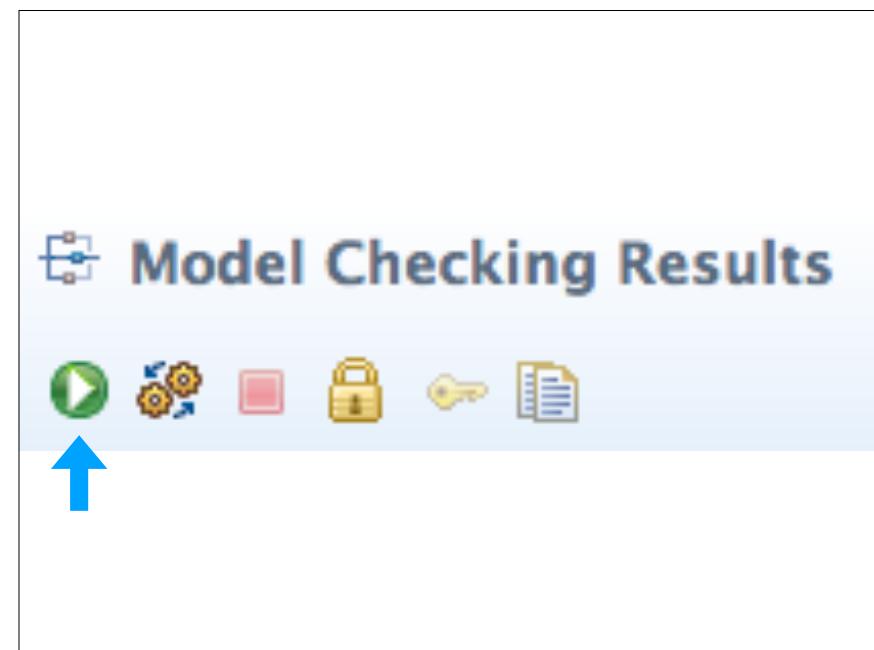
173



174



175



176

TLC Errors

Model_1

Invariant big != 4 is violated.

Error-Trace Exploration

Error-Trace

Name	Value
<Initial predicate>	State (num = 1)
big	5
small	0
<Action line 11, col 14...>	State (num = 2)
big	5
small	0
<Action line 20, col 13...>	State (num = 3)
big	0
small	3
<Action line 23, col 15...>	State (num = 4)
big	3
small	0
<Action line 11, col 14...>	State (num = 5)
big	3
small	3
<Action line 23, col 15...>	State (num = 6)
big	1
small	1
<Action line 20, col 13...>	State (num = 7)
big	0
small	1
<Action line 23, col 15...>	State (num = 8)
big	1
small	0
<Action line 11, col 14...>	State (num = 9)
big	3
small	0
<Action line 23, col 15...>	State (num = 10)
big	4
small	0

177

Error-Trace

Name	Value
<Initial predicate>	State (num = 1)
big	5
small	0
<Action line 11, col 14...>	State (num = 2)
big	5
small	0
<Action line 20, col 13...>	State (num = 3)
big	3
small	0
<Action line 23, col 15...>	State (num = 4)
big	0
small	3
<Action line 11, col 14...>	State (num = 5)
big	3
small	3
<Action line 23, col 15...>	State (num = 6)
big	5
small	1
<Action line 20, col 13...>	State (num = 7)
big	0
small	1
<Action line 23, col 15...>	State (num = 8)
big	1
small	0
<Action line 11, col 14...>	State (num = 9)
big	1
small	3
<Action line 23, col 15...>	State (num = 10)
big	4
small	0

178-1

Error-Trace

Name	Value
<Initial predicate>	State (num = 1)
big	5
small	0
<Action line 11, col 14...>	State (num = 2)
big	5
small	3
<Action line 20, col 13...>	State (num = 3)
big	0
small	3
<Action line 23, col 15...>	State (num = 4)
big	3
small	0
<Action line 11, col 14...>	State (num = 5)
big	3
small	3
<Action line 23, col 15...>	State (num = 6)
big	5
small	1
<Action line 20, col 13...>	State (num = 7)
big	0
small	1
<Action line 23, col 15...>	State (num = 8)
big	1
small	0
<Action line 11, col 14...>	State (num = 9)
big	1
small	3
<Action line 23, col 15...>	State (num = 10)
big	4
small	0

178-2

Error-Trace

Name	Value
<Initial predicate>	State (num = 1)
big	5
small	0
<Action line 11, col 14...>	State (num = 2)
big	5
small	3
<Action line 20, col 13...>	State (num = 3)
big	0
small	3
<Action line 23, col 15...>	State (num = 4)
big	3
small	0
<Action line 11, col 14...>	State (num = 5)
big	3
small	3
<Action line 23, col 15...>	State (num = 6)
big	5
small	1
<Action line 20, col 13...>	State (num = 7)
big	0
small	1
<Action line 23, col 15...>	State (num = 8)
big	1
small	0
<Action line 11, col 14...>	State (num = 9)
big	1
small	3
<Action line 23, col 15...>	State (num = 10)
big	4
small	0

178-3

Woah

179-1

Woah

- This is amazingly powerful.

179-2

Woah

- This is amazingly powerful.
- TLC tries every single possible combination of states (i.e. behaviour/execution), and checks invariants for every state in every behaviour.

179-3

Woah

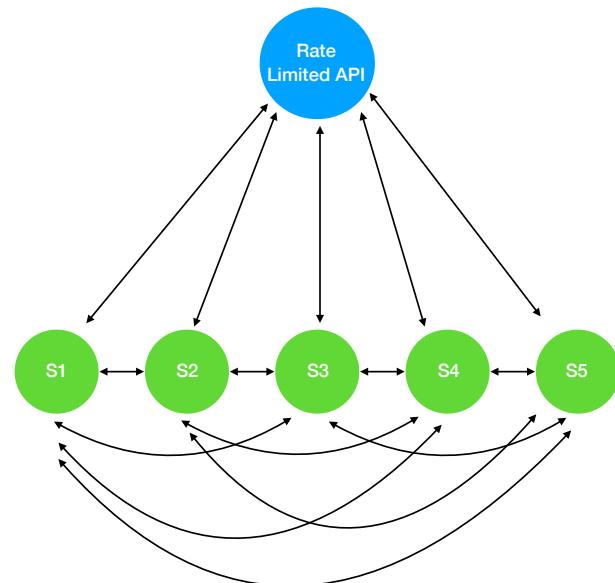
- This is amazingly powerful.
- TLC tries every single possible combination of states (i.e. behaviour/execution), and checks invariants for every state in every behaviour.
- The entire state history is kept, providing the best “stack trace” you’ve ever seen.

179-4

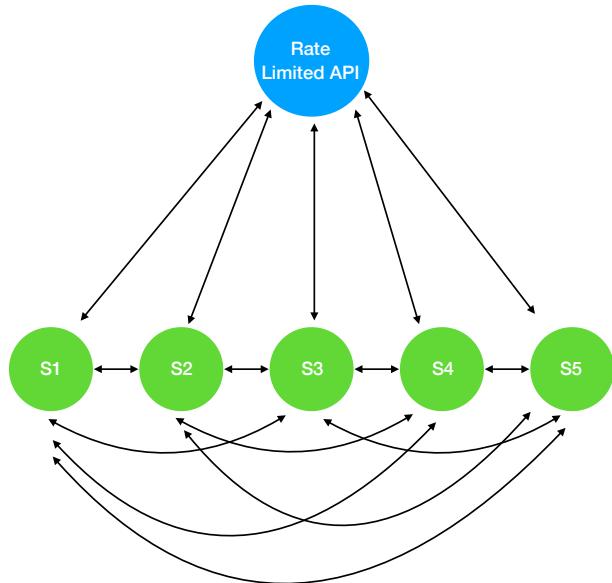
Woah

- This is amazingly powerful.
- TLC tries every single possible combination of states (i.e. behaviour/execution), and checks invariants for every state in every behaviour.
- The entire state history is kept, providing the best “stack trace” you’ve ever seen.
- For concurrent and distributed systems, this allows for detection of incredibly subtle interactions, race conditions, etc.

179-5



180-1



180-2

AWS Fault-Tolerant Replication Algorithm

181-1

AWS Fault-Tolerant Replication Algorithm

- Engineer took an existing designed and implemented algorithm at AWS, which had been analyzed for months with dozens of pages of informal proofs.

181-2

AWS Fault-Tolerant Replication Algorithm

- Engineer took an existing designed and implemented algorithm at AWS, which had been analyzed for months with dozens of pages of informal proofs.
- Wrote a TLA+ specification for the algorithm.
- TLC found three serious bugs, one of which had a 35+-step trace.

181-4

AWS Fault-Tolerant Replication Algorithm

- Engineer took an existing designed and implemented algorithm at AWS, which had been analyzed for months with dozens of pages of informal proofs.
- Wrote a TLA+ specification for the algorithm.

181-3

Transaction Commit

182-1

Transaction Commit

- Consider a replicated database.

182-2

Transaction Commit

- Consider a replicated database.
- Multiple “resource managers,” each carrying a full replica.

182-3

Transaction Commit

- Consider a replicated database.
- Multiple “resource managers,” each carrying a full replica.
- For any transaction sent to the database, all resource managers **must come to a consensus** on whether the transaction should be committed or aborted.

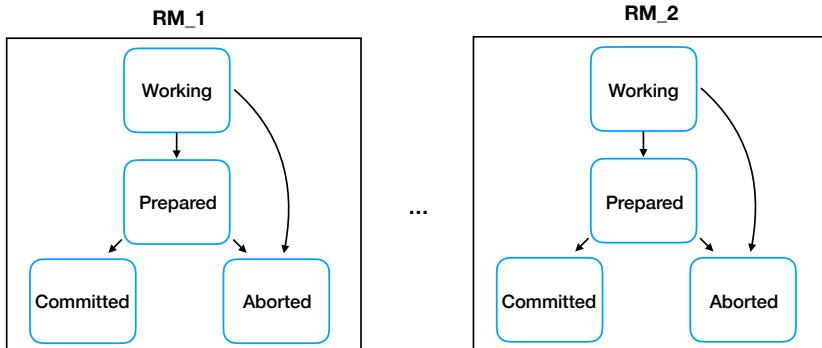
182-4

Resource Managers



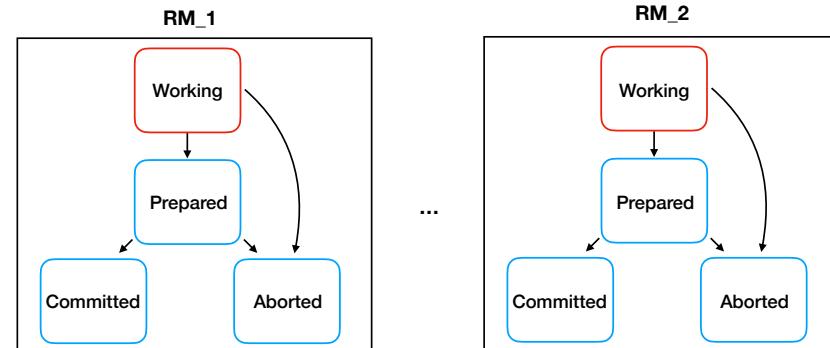
183

Resource Managers



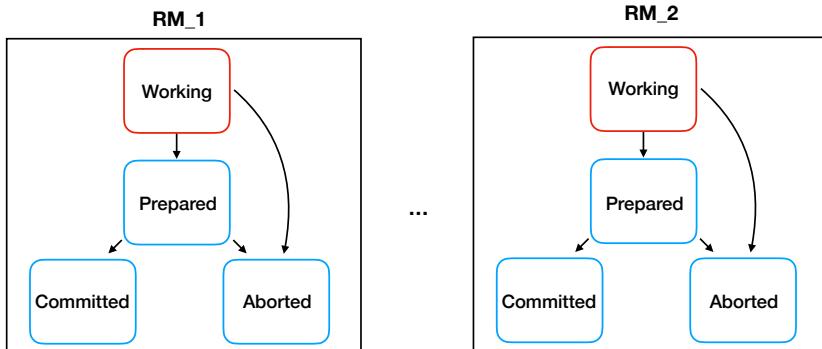
184

Resource Managers



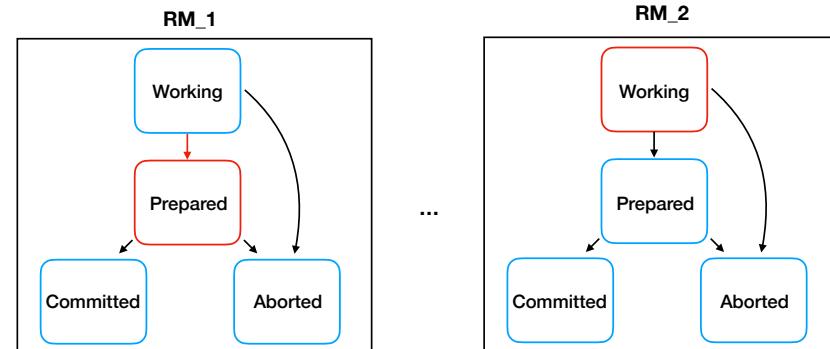
185

Resource Managers



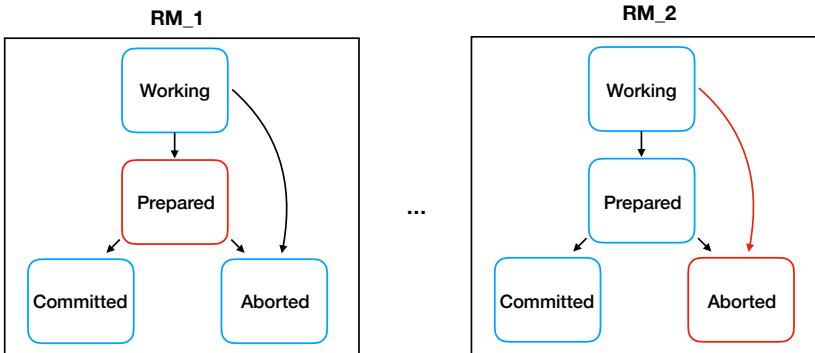
186

Resource Managers



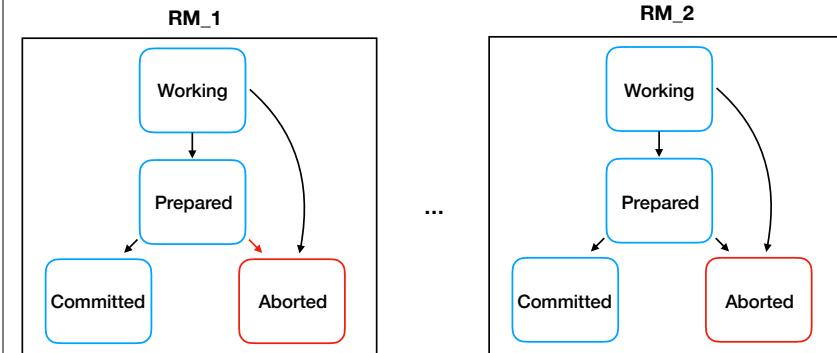
187

Resource Managers



188

Resource Managers



189

Resource Managers

```
rmState["rm1": "working"  
rmState["rm2": "working"]
```

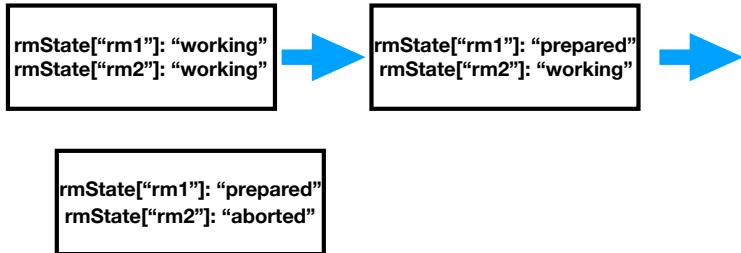
190-1

Resource Managers

```
rmState["rm1": "working"  
rmState["rm2": "working"] → rmState["rm1": "prepared"  
rmState["rm2": "working"]
```

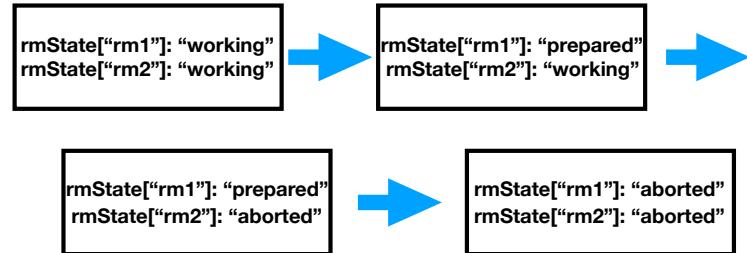
190-2

Resource Managers



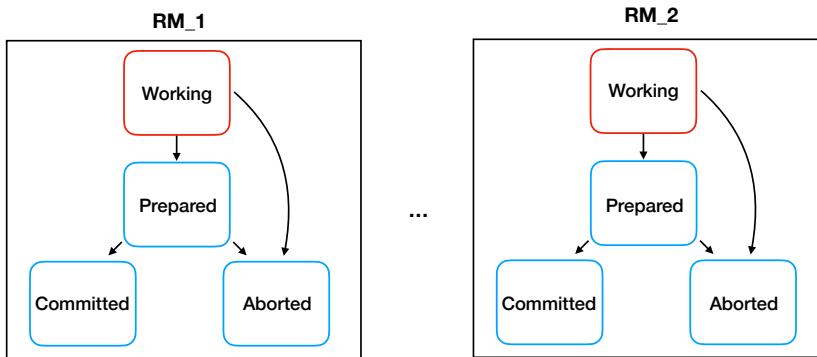
190-3

Resource Managers



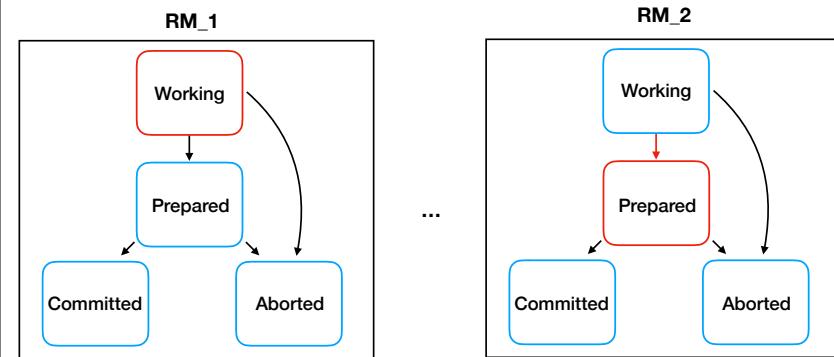
190-4

Resource Managers



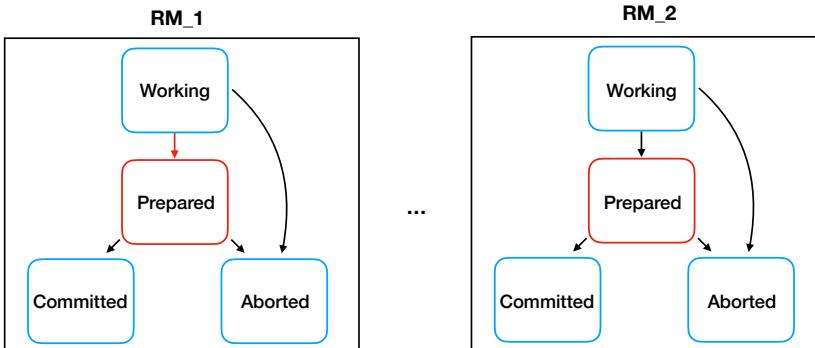
191

Resource Managers



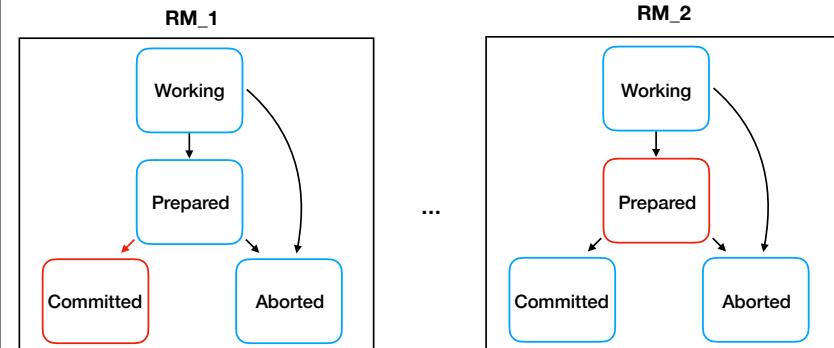
192

Resource Managers



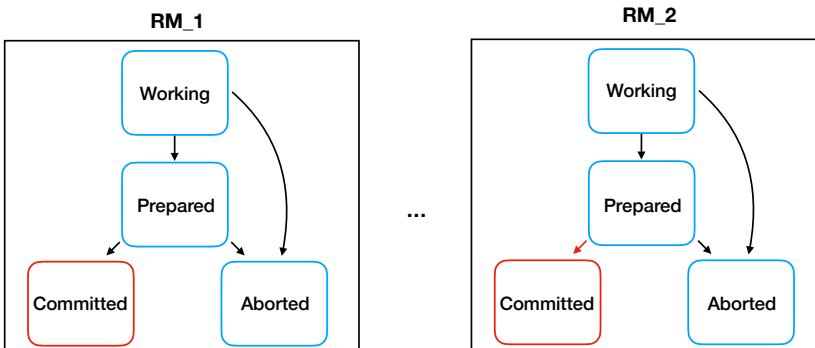
193

Resource Managers



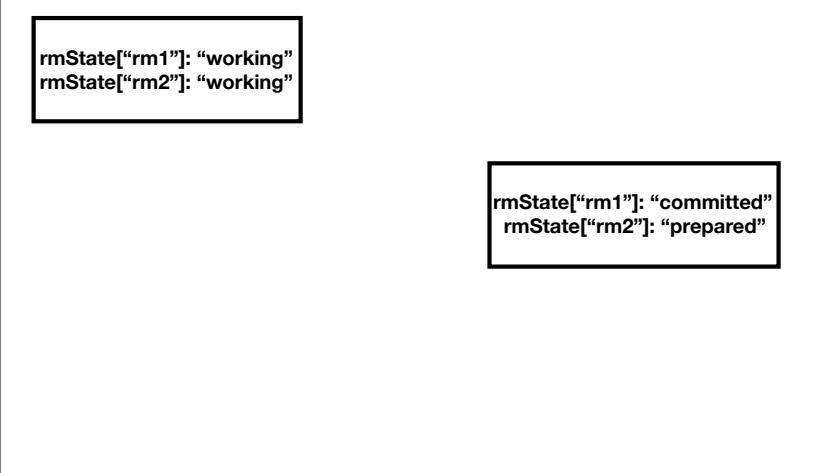
194

Resource Managers



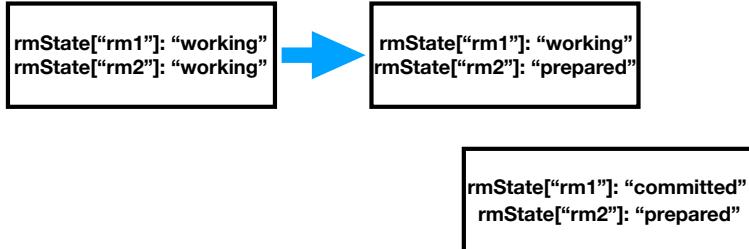
195

Resource Managers



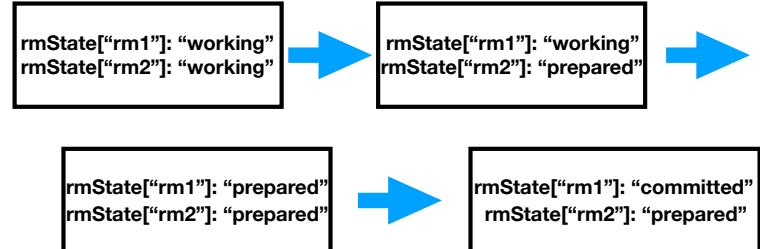
196-1

Resource Managers



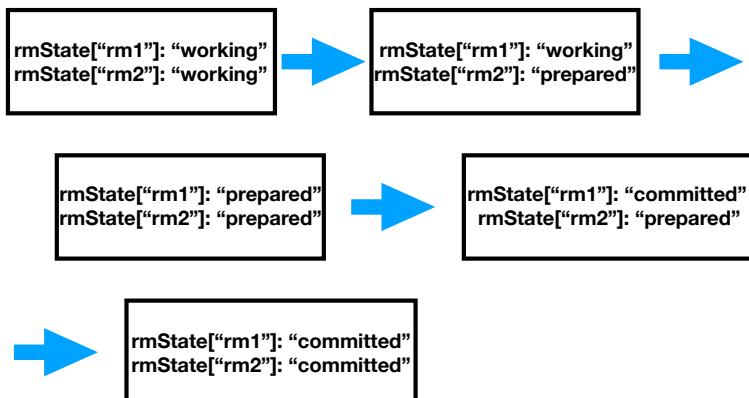
196-2

Resource Managers



196-3

Resource Managers



196-4

Resource Managers

197-1

Resource Managers

- We just looked at two possible behaviours.

197-2

Resource Managers

- We just looked at two possible behaviours.
- How many in total are possible?

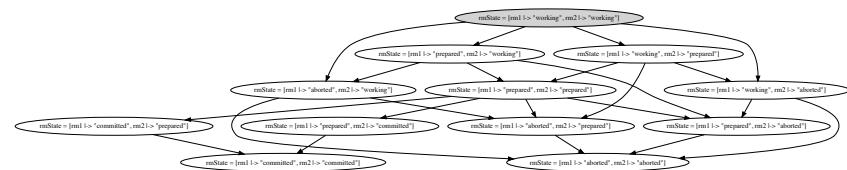
197-3

Resource Managers

- We just looked at two possible behaviours.
- How many in total are possible?
- Lots and lots! And with every additional Resource Manager added, it grows and grows.

197-4

2 Resource Managers



12 Distinct States

198

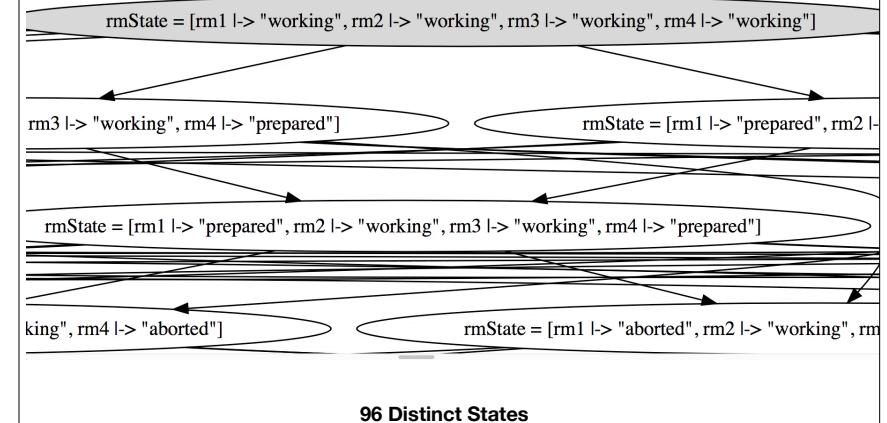
3 Resource Managers



34 Distinct States

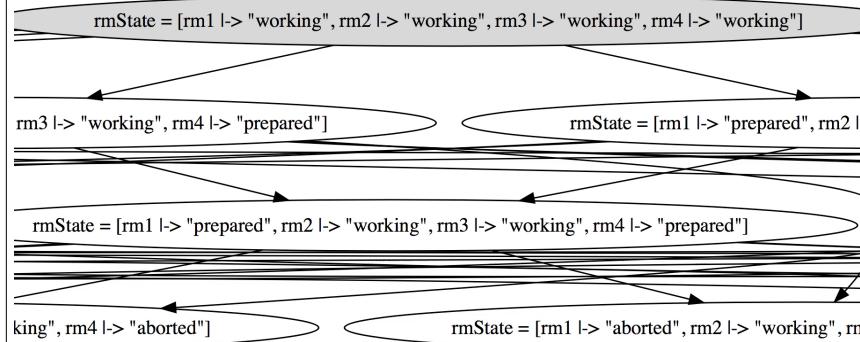
199

4 Resource Managers

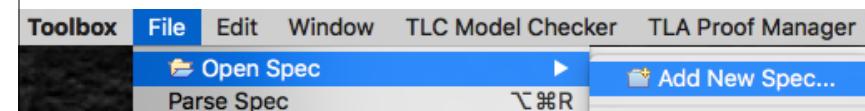


200-1

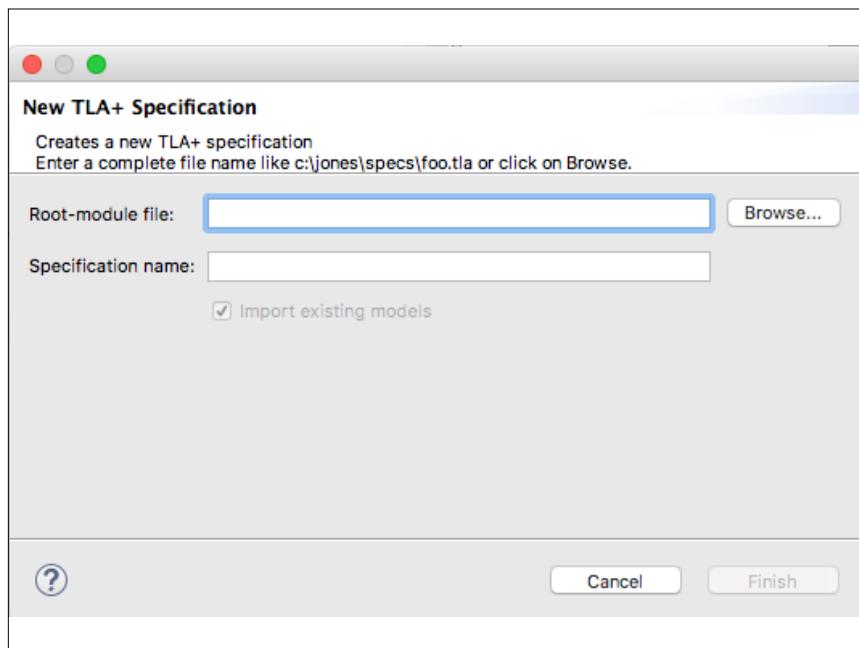
4 Resource Managers



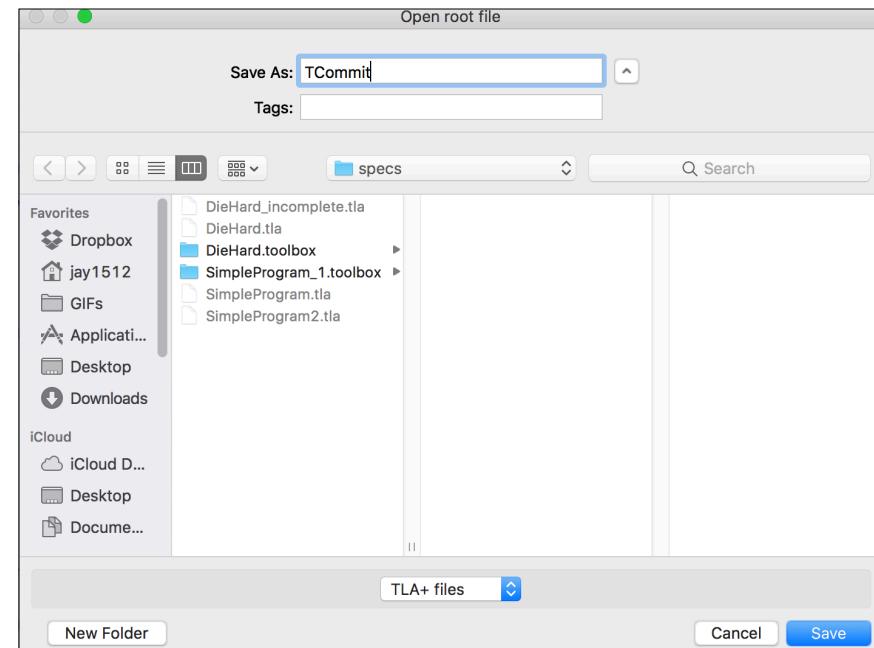
200-2



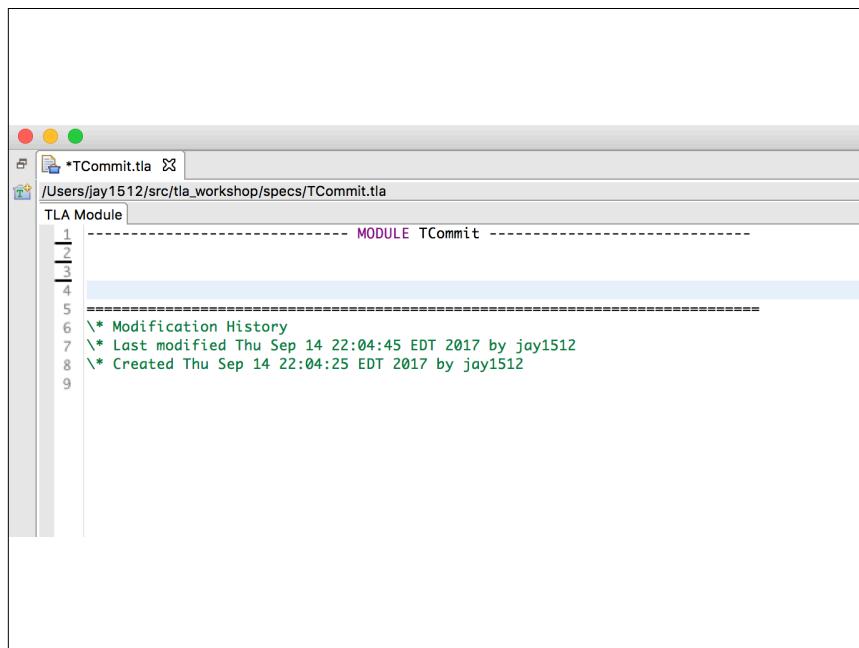
201



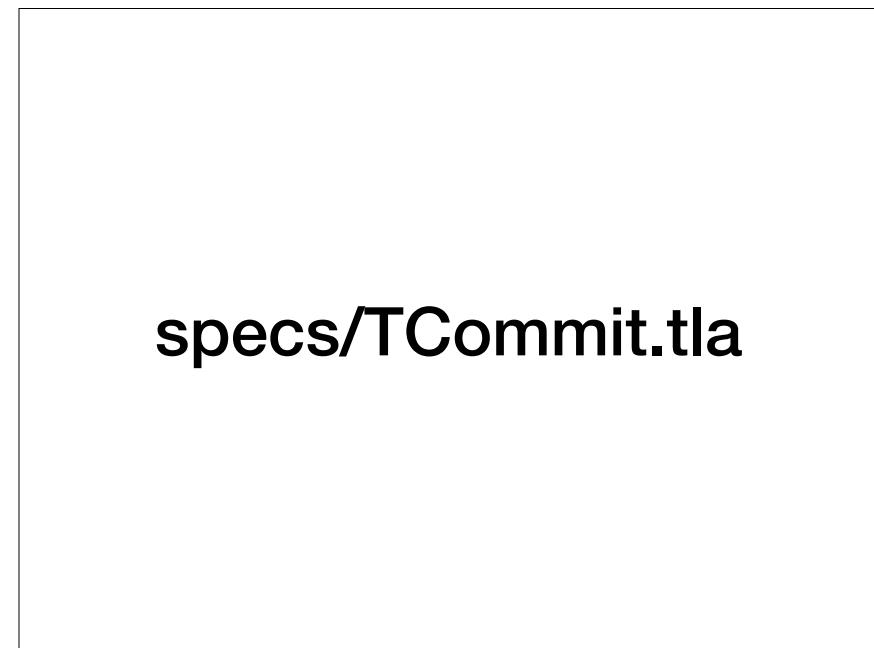
202



203



204



specs/TCommit.tla

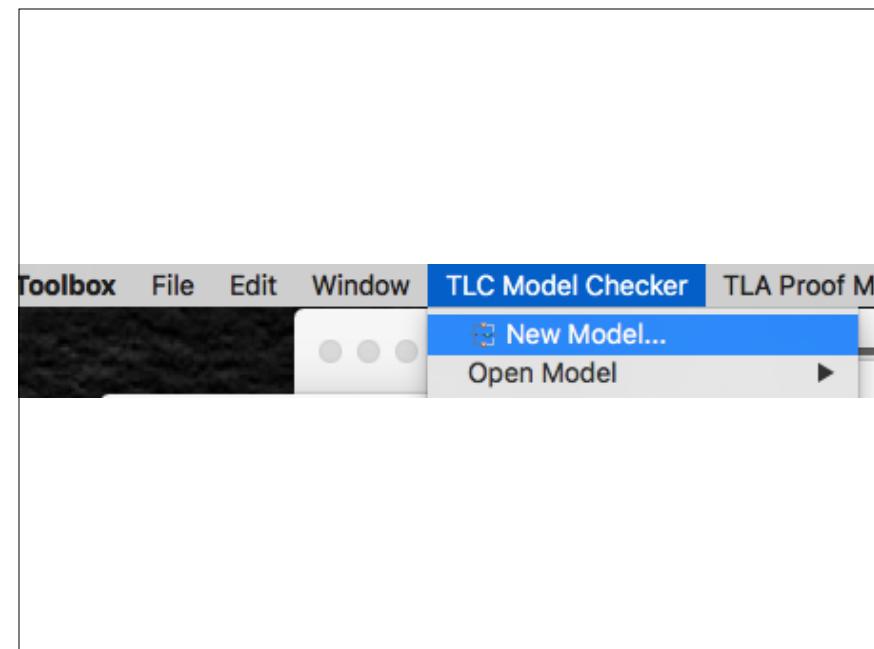
205

```

1 (* This specification is explained in "Transaction Commit", Lecture 5 of *)
2 (* the TLA+ Video Course. *)
3 (*-----*)
4 CONSTANT RM           /* The set of participating resource managers
5
6 VARIABLE rmState   /* rmState[r] is the state of resource manager r.
7
8 rmTypeOK ==          (*-----*)
9   (r : RM) rmState[r] = "working"
10
11 TCTypeOK ==          (*-----*)
12   (r : RM) rmTypeOK[r] = true
13   rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
14
15 TCInit == rmState = [r : RM -> "working"]
16
17 (* The initial predicate *)
18 (*-----*)
19
20 canCommit == \A r : RM : rmState[r] \in {"prepared", "committed"}
21
22 (* True iff all RMs are in the "prepared" or "committed" state. *)
23 (*-----*)
24
25 notCommitted == \A r : RM : rmState[r] \neq "committed"
26
27 (* True iff no resource manager has decided to commit. *)
28 (*-----*)
29
30 Prepare(r) ==          (*-----*)
31   rmState[r] = "working" & rmState \setminus {r} = "prepared"
32
33 Decide(r) == \V r : RM : rmState[r] = "prepared"
34
35 TCNext == \E r : RM : Prepare(r) \vee Decide(r)
36
37 (* The next-state action *)
38 (*-----*)
39
40

```

206



207

Model Overview 3 errors detected

Model description

What is the behavior spec?

Initial predicate and next-state relation

Init:

Next:

Temporal formula

No Behavior Spec

What is the model?

Specify the values of declared constants.

RM <-

How to run?

Advanced parts of the model: [Additional def](#), [State constraints](#), [Action constraints](#)

208

Model Overview 1 errors detected

Model description

What is the behavior spec?

Initial predicate and next-state relation

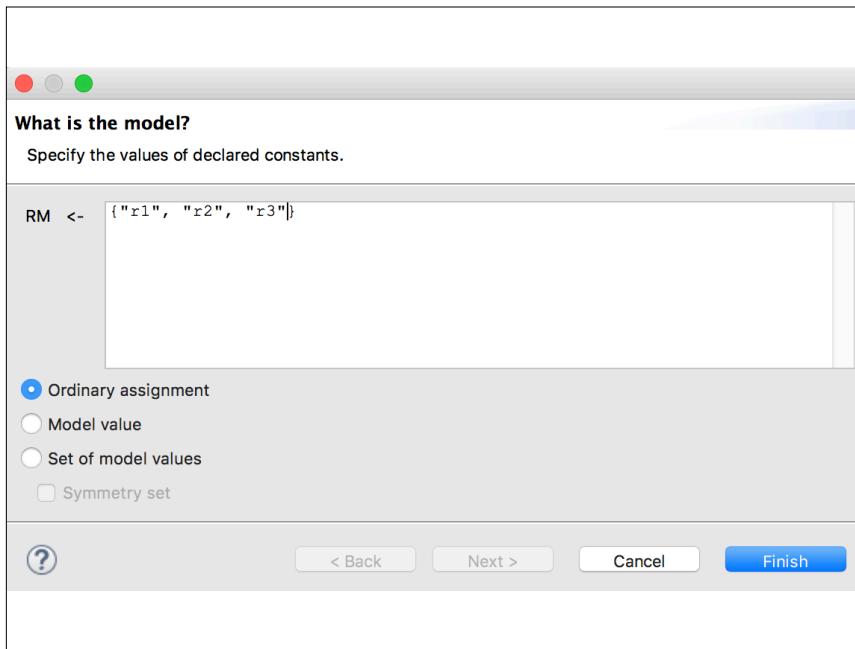
Init:

Next:

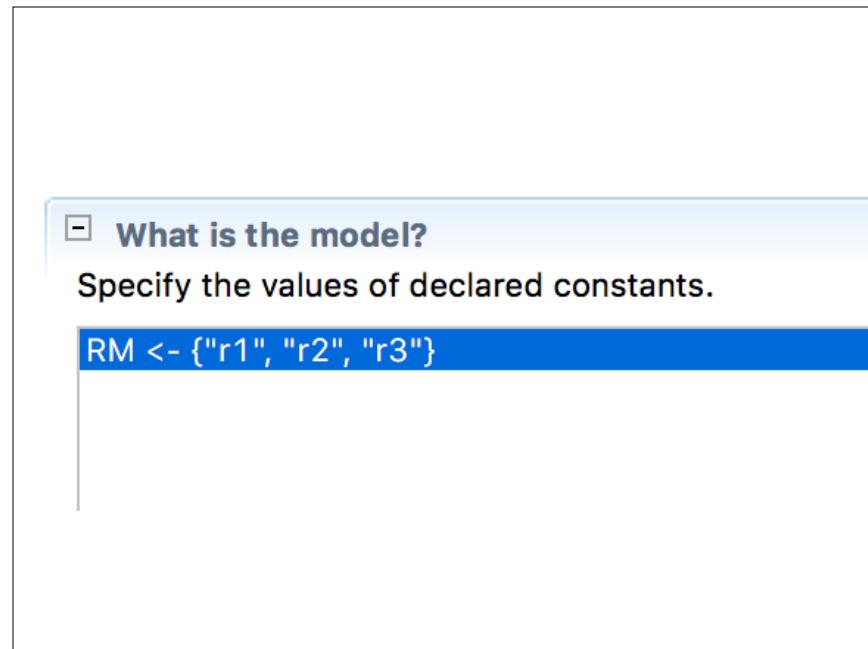
Temporal formula

No Behavior Spec

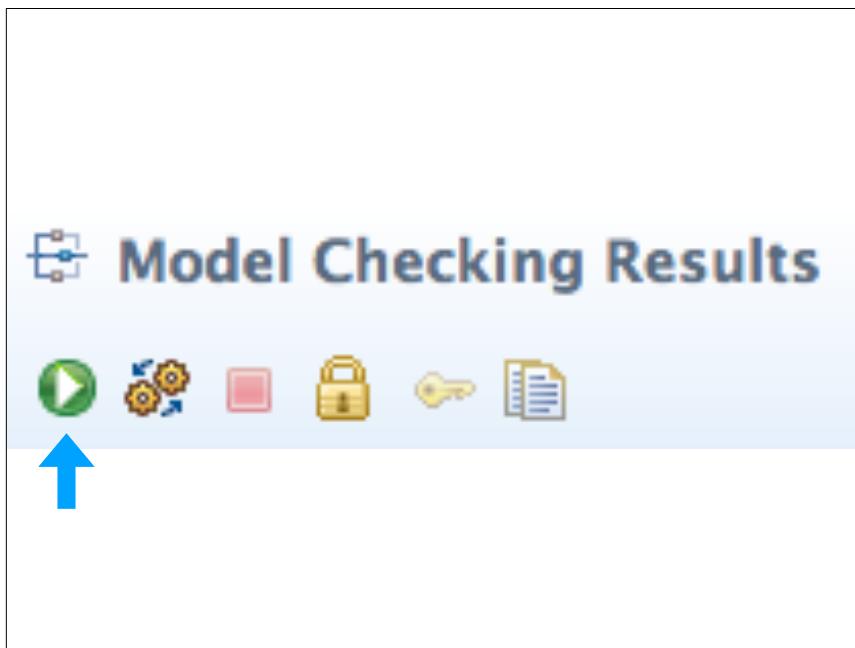
209



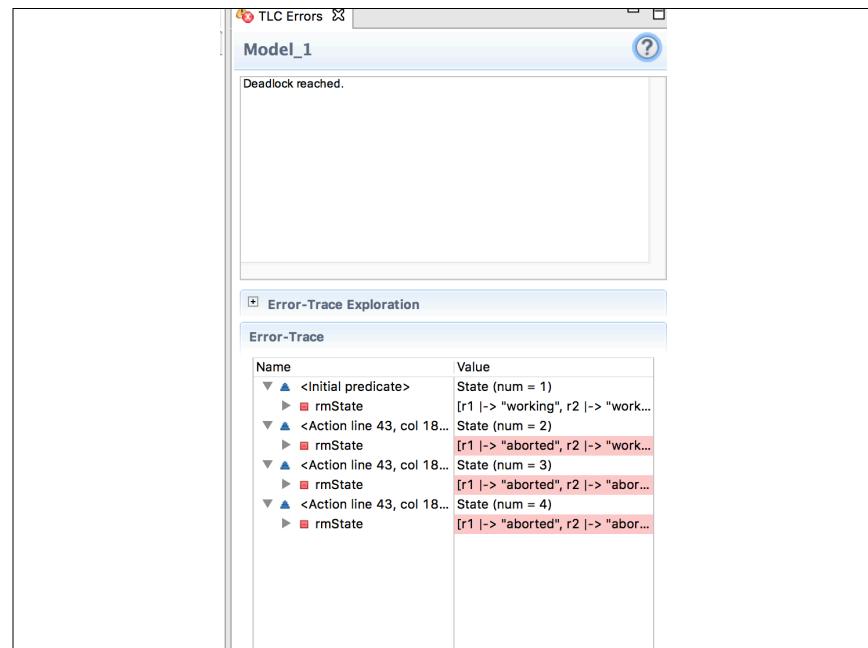
210



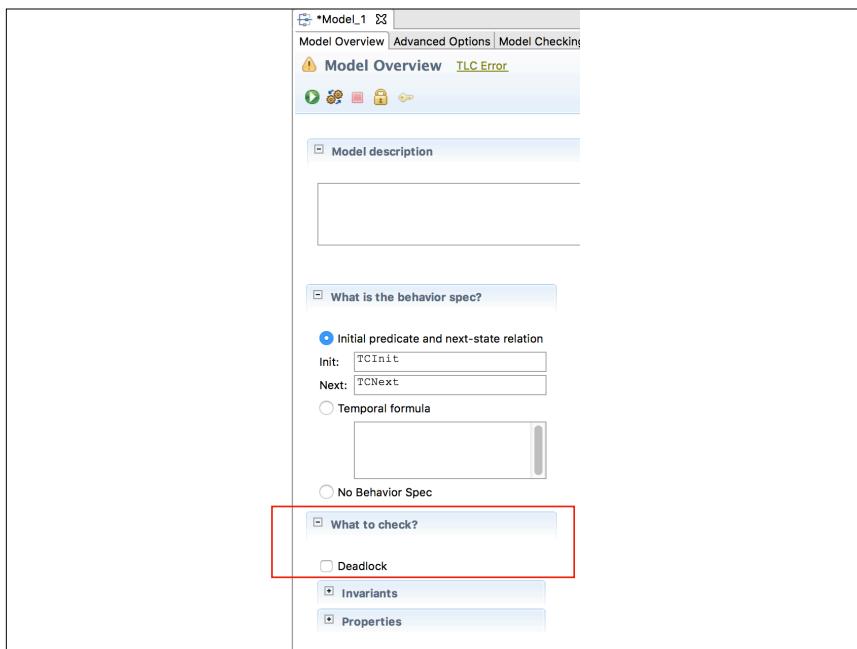
211



212



213



214

What to check?

Deadlock

Invariants

Formulas true in every reachable state.

TCTypeOK
 TCConsistent

Add
Edit
Remove

Properties

215

Model Checking Results

General

Start time: Mon Sep 18 11:08:24 EDT 2017
End time: Mon Sep 18 11:08:25 EDT 2017
Last checkpoint time:
Current status: Not running
Errors detected: No errors
Fingerprint collision probability: calculated: 1.1E-16, observed: 2.9E-17

Statistics

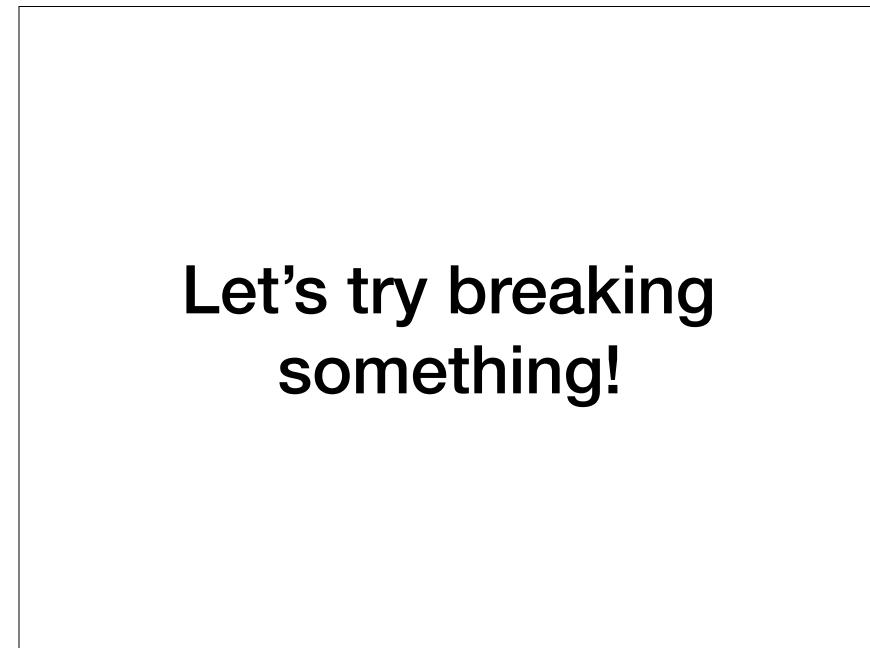
State space progress (click column header for graph)

Time	Diameter	States Found	Distinct States	Queue Size	Coverage at
2017-09-18 11:08...	7	94	34	0	2017-09-18 11:08:25

Module	Location	Count
TCommit	line 37, col 18 to line 37, col 62	27
TCommit	line 41, col 21 to line 41, col 66	12
TCommit	line 44, col 21 to line 44, col 64	54

216

Let's try breaking something!



217

```

Decide(r) == \vee \wedge rmState[r] = "prepared"
  \* \wedge canCommit
    \wedge rmState' = [rmState EXCEPT ![r] = "committed"]
\vee \wedge rmState[r] \in {"working", "prepared"}
\wedge notCommitted
\wedge rmState' = [rmState EXCEPT ![r] = "aborted"]

```

218-1

```

Decide(r) == \vee \wedge rmState[r] = "prepared"
  \* \wedge canCommit
    \wedge rmState' = [rmState EXCEPT ![r] = "committed"]
\vee \wedge rmState[r] \in {"working", "prepared"}
\wedge notCommitted
\wedge rmState' = [rmState EXCEPT ![r] = "aborted"]

```

- The `*` is a TLA+ single-line comment, we're temporarily removing the **canCommit** requirement.

218-2

```

Decide(r) == \vee \wedge rmState[r] = "prepared"
  \* \wedge canCommit
    \wedge rmState' = [rmState EXCEPT ![r] = "committed"]
\vee \wedge rmState[r] \in {"working", "prepared"}
\wedge notCommitted
\wedge rmState' = [rmState EXCEPT ![r] = "aborted"]

```

- The `*` is a TLA+ single-line comment, we're temporarily removing the **canCommit** requirement.
- This means that **canCommit** no longer needs to be true to allow a state transition where some RM **r** goes from “prepared” to “committed.”

218-3

```

Decide(r) == \vee \wedge rmState[r] = "prepared"
  \* \wedge canCommit
    \wedge rmState' = [rmState EXCEPT ![r] = "committed"]
\vee \wedge rmState[r] \in {"working", "prepared"}
\wedge notCommitted
\wedge rmState' = [rmState EXCEPT ![r] = "aborted"]

```

- The `*` is a TLA+ single-line comment, we're temporarily removing the **canCommit** requirement.
- This means that **canCommit** no longer needs to be true to allow a state transition where some RM **r** goes from “prepared” to “committed.”
- There could now be some other RM in an “aborted” state, and the specification would permit this transition.

218-4

Model_1

Model Overview | Advanced Options | Model Checking Results

Model Checking Results 2 warnings detected

General

- Start time: Mon Sep 18 11:15:45 EDT 2017
- End time: Mon Sep 18 11:15:46 EDT 2017
- Last checkpoint time:
- Current status: Not running
- Errors detected: 1 Error
- Fingerprint collision probability:

Statistics

State space progress (click column header for graph)				
Time	Diameter	States Found	Distinct States	Queue Size
2017-09-18 11:15...	5	45	31	18

Error-Trace Exploration

Invariant TCConsistent is violated.

Error-Trace

Name	Value
Initial predicate	State (num = 1)
rmState	[rm1 -> "working", rm2 -> "working", rm3 -> "working"]
Action line 36, col 15 to line 37,...	State (num = 2)
rmState	[rm1 -> "prepared", rm2 -> "working", rm3 -> "working"]
Action line 42, col 18 to line 44,...	State (num = 3)
rmState	[rm1 -> "prepared", rm2 -> "aborted", rm3 -> "working"]
Action line 39, col 18 to line 41,...	State (num = 4)
rmState	[rm1 -> "committed", rm2 -> "aborted", rm3 -> "working"]

219-1

Model_1

Model Overview | Advanced Options | Model Checking Results

Model Checking Results 2 warnings detected

General

- Start time: Mon Sep 18 11:15:45 EDT 2017
- End time: Mon Sep 18 11:15:46 EDT 2017
- Last checkpoint time:
- Current status: Not running
- Errors detected: 1 Error
- Fingerprint collision probability:

Statistics

State space progress (click column header for graph)				
Time	Diameter	States Found	Distinct States	Queue Size
2017-09-18 11:15...	5	45	31	18

Error-Trace Exploration

Invariant TCConsistent is violated.

Error-Trace

Name	Value
Initial predicate	State (num = 1)
rmState	[rm1 -> "working", rm2 -> "working", rm3 -> "working"]
Action line 36, col 15 to line 37,...	State (num = 2)
rmState	[rm1 -> "prepared", rm2 -> "working", rm3 -> "working"]
Action line 42, col 18 to line 44,...	State (num = 3)
rmState	[rm1 -> "prepared", rm2 -> "aborted", rm3 -> "working"]
Action line 39, col 18 to line 41,...	State (num = 4)
rmState	[rm1 -> "committed", rm2 -> "aborted", rm3 -> "working"]

219-2

Model_1

Model Overview | Advanced Options | Model Checking Results

Model Checking Results 2 warnings detected

General

- Start time: Mon Sep 18 11:15:45 EDT 2017
- End time: Mon Sep 18 11:15:46 EDT 2017
- Last checkpoint time:
- Current status: Not running
- Errors detected: 1 Error
- Fingerprint collision probability:

Statistics

State space progress (click column header for graph)				
Time	Diameter	States Found	Distinct States	Queue Size
2017-09-18 11:15...	5	45	31	18

Error-Trace Exploration

Invariant TCConsistent is violated.

Error-Trace

Name	Value
Initial predicate	State (num = 1)
rmState	[rm1 -> "working", rm2 -> "working", rm3 -> "working"]
Action line 36, col 15 to line 37,...	State (num = 2)
rmState	[rm1 -> "prepared", rm2 -> "working", rm3 -> "working"]
Action line 42, col 18 to line 44,...	State (num = 3)
rmState	[rm1 -> "prepared", rm2 -> "aborted", rm3 -> "working"]
Action line 39, col 18 to line 41,...	State (num = 4)
rmState	[rm1 -> "committed", rm2 -> "aborted", rm3 -> "working"]

219-3

Model_1

Model Overview | Advanced Options | Model Checking Results

Model Checking Results 2 warnings detected

Error-Trace

Invariant TCConsistent is violated.

Error-Trace

Name	Value
Initial predicate	State (num = 1)
rmState	[rm1 -> "working", rm2 -> "working", rm3 -> "working"]
Action line 36, col 15 to line 37, col 62 of module...	State (num = 2)
rmState	[rm1 -> "prepared", rm2 -> "working", rm3 -> "working"]
Action line 42, col 18 to line 44, col 64 of module...	State (num = 3)
rmState	[rm1 -> "prepared", rm2 -> "aborted", rm3 -> "working"]
Action line 39, col 18 to line 41, col 66 of module...	State (num = 4)
rmState	[rm1 -> "committed", rm2 -> "aborted", rm3 -> "working"]

220

Error-Trace	
Name	Value
▼ ▲ <Initial predicate>	State (num = 1)
► ■ rmState	[rm1 -> "working", rm2 -> "working", rm3 -> "working"]
▼ ▲ <Action line 36, col 15 to line 37, col 62 of module...>	State (num = 2)
► ■ rmState	[rm1 -> "prepared", rm2 -> "working", rm3 -> "working"]
▼ ▲ <Action line 42, col 18 to line 44, col 64 of module...>	State (num = 3)
► ■ rmState	[rm1 -> "prepared", rm2 -> "aborted", rm3 -> "working"]
▼ ▲ <Action line 39, col 18 to line 41, col 66 of module...>	State (num = 4)
► ■ rmState	[rm1 -> "committed", rm2 -> "aborted", rm3 -> "working"]

Double-click this row

221

```

30 -----
31 (* We now define the actions that may be performed by the RMs, and then *)
32 (* define the complete next-state action of the specification to be the *)
33 (* disjunction of the possible RM actions. *)
34 -----
35 Prepare(r) == /\ rmState[r] = "working"
36           /\ rmState' = [rmState EXCEPT ![r] = "prepared"]
37
38 Decide(r)  == \/
39   /\ rmState[r] = "prepared"
40   /* /\ canCommit
41   /\ rmState' = [rmState EXCEPT ![r] = "committed"] */
42   \/
43   /\ rmState[r] \in {"working", "prepared"}
44   /\ notCommitted
45   /\ rmState' = [rmState EXCEPT ![r] = "aborted"]

```

222

Resource Managers

rmState["rm1": "working"
rmState["rm2": "working"
rmState["rm3": "working"]

223-1

Resource Managers

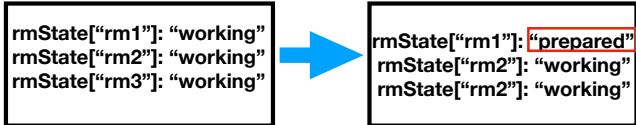
rmState["rm1": "working"
rmState["rm2": "working"
rmState["rm3": "working"]



rmState["rm1": "prepared"
rmState["rm2": "working"
rmState["rm2": "working"]]

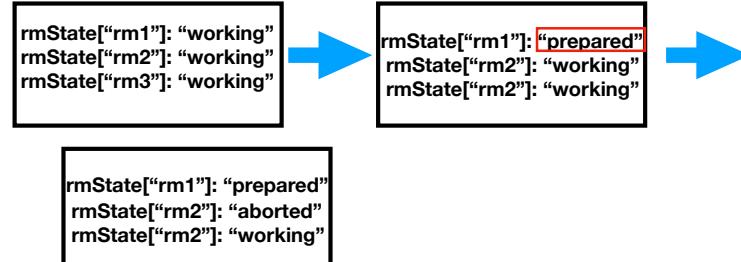
223-2

Resource Managers



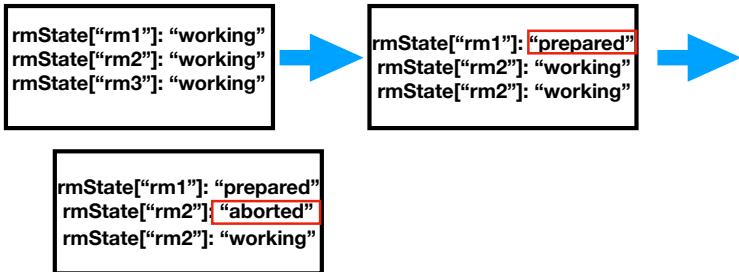
223-3

Resource Managers



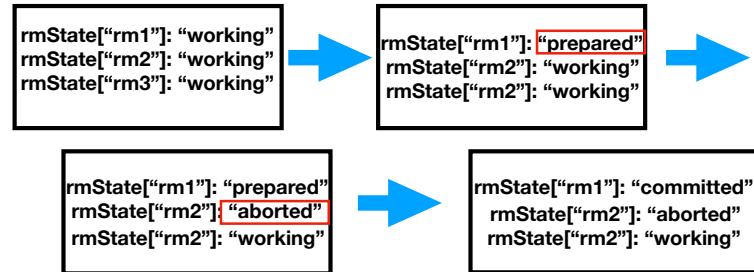
223-4

Resource Managers



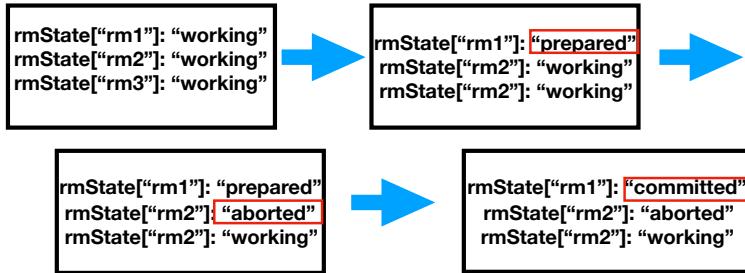
223-5

Resource Managers



223-6

Resource Managers



223-7

Syntax time

224

Functions

- A TLA+ function is a *true* mathematical function. The term “function” we use in most programming languages is completely wrong. Those things should generally be called “sub-routines”

225-1

225-2

Functions

- A TLA+ function is a *true* mathematical function. The term “function” we use in most programming languages is completely wrong. Those things should generally be called “sub-routines”
- A TLA+ function is a set of **key:value** pairs. Kind of like an array, kind of like a Python dict or a Ruby hash

225-3

Functions

- A TLA+ function is a *true* mathematical function. The term “function” we use in most programming languages is completely wrong. Those things should generally be called “sub-routines”
- A TLA+ function is a set of **key:value** pairs. Kind of like an array, kind of like a Python dict or a Ruby hash
- All **keys** in a function must be of the same type. i.e. they all must be numbers, or they all must be strings, or they all must be sets.

225-4

Functions

- A TLA+ function is a *true* mathematical function. The term “function” we use in most programming languages is completely wrong. Those things should generally be called “sub-routines”
- A TLA+ function is a set of **key:value** pairs. Kind of like an array, kind of like a Python dict or a Ruby hash
- All **keys** in a function must be of the same type. i.e. they all must be numbers, or they all must be strings, or they all must be sets.
- The **values** can be of mixed types

225-5

Functions

- A TLA+ function is a *true* mathematical function. The term “function” we use in most programming languages is completely wrong. Those things should generally be called “sub-routines”
- A TLA+ function is a set of **key:value** pairs. Kind of like an array, kind of like a Python dict or a Ruby hash
- All **keys** in a function must be of the same type. i.e. they all must be numbers, or they all must be strings, or they all must be sets.
- The **values** can be of mixed types
- The set of **keys** are called the DOMAIN, the set of **values** are the RANGE or IMAGE

225-6

Functions

226-1

Functions

- TLA+ provides lots of ways to create functions. We'll only look at the commonly used ones

226-2

Functions

- TLA+ provides lots of ways to create functions. We'll only look at the commonly used ones
- $[i \in S \rightarrow e]$ This creates a function whose domain is all the elements of **S** (i.e. the keys are the elements of **S**), and the corresponding values are created by evaluating the expression **e**

226-3

Functions

- TLA+ provides lots of ways to create functions. We'll only look at the commonly used ones
- $[i \in S \rightarrow e]$ This creates a function whose domain is all the elements of **S** (i.e. the keys are the elements of **S**), and the corresponding values are created by evaluating the expression **e**

$[i \in \{2,3,4\} \rightarrow i^2] = (2 :> 4 @@ 3 :> 6 @@ 4 :> 8)$

226-4

Functions

- TLA+ provides lots of ways to create functions. We'll only look at the commonly used ones
- $[i \in S \mid\rightarrow e]$ This creates a function whose domain is all the elements of **S** (i.e. the keys are the elements of **S**), and the corresponding values are created by evaluating the expression **e**
 $[i \in \{2,3,4\} \mid\rightarrow i^2] = (2 :> 4 @@ 3 :> 6 @@ 4 :> 8)$
- Read this as: "Take the set to the left of $\mid\rightarrow$ and use that as the keys. For each key, evaluate the expression to the right of $\mid\rightarrow$, and use that as the value"

226-5

Functions

- TLA+ provides lots of ways to create functions. We'll only look at the commonly used ones
- $[i \in S \mid\rightarrow e]$ This creates a function whose domain is all the elements of **S** (i.e. the keys are the elements of **S**), and the corresponding values are created by evaluating the expression **e**
 $[i \in \{2,3,4\} \mid\rightarrow i^2] = (2 :> 4 @@ 3 :> 6 @@ 4 :> 8)$
- Read this as: "Take the set to the left of $\mid\rightarrow$ and use that as the keys. For each key, evaluate the expression to the right of $\mid\rightarrow$, and use that as the value"
- That syntax on the right side of the $=$ is rarely used. But it's equivalent to the following Python dict: {2:4, 3:6, 4:8}

226-6

Functions

- TLA+ provides lots of ways to create functions. We'll only look at the commonly used ones
- $[i \in S \mid\rightarrow e]$ This creates a function whose domain is all the elements of **S** (i.e. the keys are the elements of **S**), and the corresponding values are created by evaluating the expression **e**
 $[i \in \{2,3,4\} \mid\rightarrow i^2] = (2 :> 4 @@ 3 :> 6 @@ 4 :> 8)$
- Read this as: "Take the set to the left of $\mid\rightarrow$ and use that as the keys. For each key, evaluate the expression to the right of $\mid\rightarrow$, and use that as the value"
- That syntax on the right side of the $=$ is rarely used. But it's equivalent to the following Python dict: {2:4, 3:6, 4:8}
- The syntax above is equivalent to Python's "dictionary comprehensions"

226-7

Functions

- TLA+ provides lots of ways to create functions. We'll only look at the commonly used ones
- $[i \in S \mid\rightarrow e]$ This creates a function whose domain is all the elements of **S** (i.e. the keys are the elements of **S**), and the corresponding values are created by evaluating the expression **e**
 $[i \in \{2,3,4\} \mid\rightarrow i^2] = (2 :> 4 @@ 3 :> 6 @@ 4 :> 8)$
- Read this as: "Take the set to the left of $\mid\rightarrow$ and use that as the keys. For each key, evaluate the expression to the right of $\mid\rightarrow$, and use that as the value"
- That syntax on the right side of the $=$ is rarely used. But it's equivalent to the following Python dict: {2:4, 3:6, 4:8}
- The syntax above is equivalent to Python's "dictionary comprehensions"

{i: i² for i in [2,3,4]}

226-8

Functions

- The value for some key can be accessed using the same syntax as array/dictionary access in Python

```
f == [ x \in {1,2,3} |-> x * 5 ]
```

```
f[1] = 5  
f[2] = 10  
f[3] = 15
```

227

Functions

228-1

Functions

- TLA+ has a special operator called **DOMAIN** which will return the domain of any function.

Functions

- TLA+ has a special operator called **DOMAIN** which will return the domain of any function.

```
DOMAIN [ i \in {2, 4, 6} |-> i*2 ] = {2,4,6}
```

228-2

228-3

Functions

- TLA+ has a special operator called **DOMAIN** which will return the domain of any function.

```
DOMAIN [ i ∈ {2, 4, 6} |-> i*2 ] = {2,4,6}
```

Exercise:

If we have the following function **f**

```
f == [x \in {1,2,3} |-> x*5]
```

How could you find the range (i.e. the set of values) of this function?

228-4

Functions

229-1

Functions

```
f == [x \in {1,2,3} |-> x*5]
```

Functions

```
f == [x \in {1,2,3} |-> x*5]
```

Answer:

229-2

229-3

Functions

```
f == [x \in {1,2,3} | -> x*5]
```

Answer:

```
{ f[i] : i \in DOMAIN f }
```

229-4

Sets of functions

230-1

Sets of functions

- It is very common in TLA+ to need to construct sets of functions.

230-2

Sets of functions

- It is very common in TLA+ to need to construct sets of functions.
- The general syntax is $[S \rightarrow T]$, where S and T are sets.

230-3

Sets of functions

- It is very common in TLA+ to need to construct sets of functions.
- The general syntax is $[S \rightarrow T]$, where **S** and **T** are sets.

NOTE: That's \rightarrow , not the $| \rightarrow$ we just looked at. This will screw you up at some point.

230-4

Sets of functions

- It is very common in TLA+ to need to construct sets of functions.
- The general syntax is $[S \rightarrow T]$, where **S** and **T** are sets.

NOTE: That's \rightarrow , not the $| \rightarrow$ we just looked at. This will screw you up at some point.

- $[S \rightarrow T]$ means

230-5

Sets of functions

- It is very common in TLA+ to need to construct sets of functions.
- The general syntax is $[S \rightarrow T]$, where **S** and **T** are sets.

NOTE: That's \rightarrow , not the $| \rightarrow$ we just looked at. This will screw you up at some point.

- $[S \rightarrow T]$ means
 - Create a set of functions.

230-6

Sets of functions

- It is very common in TLA+ to need to construct sets of functions.
- The general syntax is $[S \rightarrow T]$, where **S** and **T** are sets.

NOTE: That's \rightarrow , not the $| \rightarrow$ we just looked at. This will screw you up at some point.

- $[S \rightarrow T]$ means
 - Create a set of functions.
 - Each function in the set will have the domain **S**.

230-7

Sets of functions

- It is very common in TLA+ to need to construct sets of functions.
- The general syntax is $[S \rightarrow T]$, where **S** and **T** are sets.

NOTE: That's \rightarrow , not the $| \rightarrow$ we just looked at. This will screw you up at some point.

- $[S \rightarrow T]$ means
 - Create a set of functions.
 - Each function in the set will have the domain **S**.
 - The range of each set will be one of any combination of values of **T**.

230-8

Sets of functions

- It is very common in TLA+ to need to construct sets of functions.
- The general syntax is $[S \rightarrow T]$, where **S** and **T** are sets.

NOTE: That's \rightarrow , not the $| \rightarrow$ we just looked at. This will screw you up at some point.

- $[S \rightarrow T]$ means
 - Create a set of functions.
 - Each function in the set will have the domain **S**.
 - The range of each set will be one of any combination of values of **T**.
 - All possible combinations will be represented in the final set.
- Huh?

230-10

Sets of functions

- It is very common in TLA+ to need to construct sets of functions.
- The general syntax is $[S \rightarrow T]$, where **S** and **T** are sets.

NOTE: That's \rightarrow , not the $| \rightarrow$ we just looked at. This will screw you up at some point.

- $[S \rightarrow T]$ means
 - Create a set of functions.
 - Each function in the set will have the domain **S**.
 - The range of each set will be one of any combination of values of **T**.
 - All possible combinations will be represented in the final set.

230-9

Sets of functions

```
{ (2 :> "a" @@ 3 :> "a"),
  (2 :> "a" @@ 3 :> "b"),
  (2 :> "a" @@ 3 :> "c"),
  (2 :> "b" @@ 3 :> "a"),
  (2 :> "b" @@ 3 :> "b"),
  (2 :> "b" @@ 3 :> "c"),
  (2 :> "c" @@ 3 :> "a"),
  (2 :> "c" @@ 3 :> "b"),
  (2 :> "c" @@ 3 :> "c") }
```

That output is **roughly** equivalent to the following Python set of dictionaries:

```
{ {2:"a", 3:"a"}, 
  {2:"a", 3:"b"}, 
  {2:"a", 3:"c"}, 
  {2:"b", 3:"a"}, 
  {2:"b", 3:"b"}, 
  {2:"b", 3:"c"}, 
  {2:"c", 3:"a"}, 
  {2:"c", 3:"b"}, 
  {2:"c", 3:"c"} }
```

231

`:>` and `@@`

```
[{2,3} -> {"a", "b"}] = { (2 :> "a" @@ 3 :> "a"),
                           (2 :> "a" @@ 3 :> "b"),
                           (2 :> "b" @@ 3 :> "a"),
                           (2 :> "b" @@ 3 :> "b")}
```

`:>` is used as the separator between **keys** and **values**
`@@` is used to concatenate together two functions (remember, a function is a set of key/value pairs)

232

Tuples

233-1

Tuples

- A tuple is like an array in a regular programming language.
But instead of 0-based, it's 1-based

233-2

Tuples

- A tuple is like an array in a regular programming language.
But instead of 0-based, it's 1-based

`<< "a", "b", "c" >>` is a tuple of three elements

233-3

Tuples

- A tuple is like an array in a regular programming language. But instead of 0-based, it's 1-based

`<< "a", "b", "c" >>` is a tuple of three elements

`<< "a", "b", "c" >>[2] = "b"`

233-4

Tuples

- A tuple is like an array in a regular programming language. But instead of 0-based, it's 1-based

`<< "a", "b", "c" >>` is a tuple of three elements

`<< "a", "b", "c" >>[2] = "b"`

- Tuples are just syntactic sugar for functions! The keys are the integers from `1..N`, and the values are whatever you write inside the `<< >>`

233-5

Tuples

- A tuple is like an array in a regular programming language. But instead of 0-based, it's 1-based

`<< "a", "b", "c" >>` is a tuple of three elements

`<< "a", "b", "c" >>[2] = "b"`

- Tuples are just syntactic sugar for functions! The keys are the integers from `1..N`, and the values are whatever you write inside the `<< >>`
- Equivalent to the Python tuple `("a", "b", "c")`, except TLA+ is 1-index

233-6

EXCEPT

234-1

EXCEPT

- A final important construct is EXCEPT

234-2

EXCEPT

- A final important construct is EXCEPT
- Say we have the following function

234-3

EXCEPT

- A final important construct is EXCEPT
- Say we have the following function

```
f == [x \in {2,4,6} | -> x*4]
```

234-4

EXCEPT

- A final important construct is EXCEPT
 - Say we have the following function
- ```
f == [x \in {2,4,6} | -> x*4]
```
- And we want to “replace” the **value at key 4** with 30

234-5

# EXCEPT

- A final important construct is EXCEPT

- Say we have the following function

```
f == [x \in {2,4,6} | -> x*4]
```

- And we want to “replace” the **value** at **key** 4 with 30

```
t = [f EXCEPT !(4) = 30]
```

234-6

# EXCEPT

- A final important construct is EXCEPT

- Say we have the following function

```
f == [x \in {2,4,6} | -> x*4]
```

- And we want to “replace” the **value** at **key** 4 with 30

```
t = [f EXCEPT !(4) = 30]
```

- You can read that as:

234-7

# EXCEPT

- A final important construct is EXCEPT

- Say we have the following function

```
f == [x \in {2,4,6} | -> x*4]
```

- And we want to “replace” the **value** at **key** 4 with 30

```
t = [f EXCEPT !(4) = 30]
```

- You can read that as:

- **t** is the same as the function as **f**, except with the **value** for the **key 4** being **30**

234-8

# EXCEPT

```
f == [x \in {2,4,6} | -> x*4]
t == [f EXCEPT !(4) = 30]
```

## A Python equivalent

```
import copy
f = {2: 8, 4: 16, 6: 24}
t = copy.deepcopy(f)
t[4] = 30
```

235

# Summary

236-1

# Summary

- A **function** is a collection of **key:value** pairs.

236-2

# Summary

- A **function** is a collection of **key:value** pairs.
- The keys are called the **DOMAIN**, the values are the **range**.

236-3

# Summary

- A **function** is a collection of **key:value** pairs.
- The keys are called the **DOMAIN**, the values are the **range**.
- A **tuple** is a function whose keys are integers from 1..N, like an array in Python/C/etc.

236-4

# Summary

- A **function** is a collection of **key:value** pairs.
- The keys are called the **DOMAIN**, the values are the **range**.
- A **tuple** is a function whose keys are integers from 1..N, like an array in Python/C/etc.
- **Tuples** are syntactic sugar over functions.

236-5

```
CONSTANT RM /* The set of participating resource managers
VARIABLE rmState /* rmState[rm] is the state of resource manager r.

TCTypeOK ==
 rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
```

237-1

```
CONSTANT RM /* The set of participating resource managers
VARIABLE rmState /* rmState[rm] is the state of resource manager r.

TCTypeOK ==
 rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
```

237-2

```
CONSTANT RM /* The set of participating resource managers
VARIABLE rmState /* rmState[rm] is the state of resource manager r.

TCTypeOK ==
 rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
```

237-3

```
CONSTANT RM /* The set of participating resource managers
VARIABLE rmState /* rmState[rm] is the state of resource manager r.

TCTypeOK ==
 rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
```

237-4

```
CONSTANT RM /* The set of participating resource managers
VARIABLE rmState /* rmState[rm] is the state of resource manager r.

TCTypeOK ==
 rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
```

Let's assume that RM will be the set {"r1", "r2"}

237-5

```
TCTypeOK ==
 rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
```

```
TCTypeOK ==
 rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
```

[**S** -> **T**] creates a set of records, where each record in the set has all the elements of **S** as keys, with every possible assignment of values in **T** to those keys

238-1

238-2

```
TCTypeOK ==
rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
```

[**S** -> **T**] creates a set of records, where each record in the set has all the elements of **S** as keys, with every possible assignment of values in **T** to those keys

238-3

```
TCTypeOK ==
rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
```

[**S** -> **T**] creates a set of records, where each record in the set has all the elements of **S** as keys, with every possible assignment of values in **T** to those keys

238-4

```
TCTypeOK ==
rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
```

[**S** -> **T**] creates a set of records, where each record in the set has all the elements of **S** as keys, with every possible assignment of values in **T** to those keys

238-5

```
TCTypeOK ==
rmState \in [RM -> [{"working", "prepared", "committed", "aborted"}]]
```

[**S** -> **T**] creates a set of records, where each record in the set has all the elements of **S** as keys, with every possible assignment of values in **T** to those keys

238-6

```
TCTypeOK ==
rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
```

$[S \rightarrow T]$  creates a set of records, where each record in the set has all the elements of **S** as keys, with every possible assignment of values in **T** to those keys

If **RM** is defined to be  $\{\text{"r1"}, \text{"r2"}\}$  (i.e. we have two resource managers), then the resulting set of functions is:

238-7

```
TCTypeOK ==
rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
```

$[S \rightarrow T]$  creates a set of records, where each record in the set has all the elements of **S** as keys, with every possible assignment of values in **T** to those keys

If **RM** is defined to be  $\{\text{"r1"}, \text{"r2"}\}$  (i.e. we have two resource managers), then the resulting set of functions is:

```
{ [r1 |-> "waiting", r2 |-> "waiting"],
[r1 |-> "waiting", r2 |-> "prepared"],
[r1 |-> "waiting", r2 |-> "aborted"],
[r1 |-> "waiting", r2 |-> "committed"],
[r1 |-> "prepared", r2 |-> "waiting"],
[r1 |-> "prepared", r2 |-> "prepared"],
[r1 |-> "prepared", r2 |-> "aborted"],
[r1 |-> "prepared", r2 |-> "committed"],
[r1 |-> "aborted", r2 |-> "waiting"],
[r1 |-> "aborted", r2 |-> "prepared"],
[r1 |-> "aborted", r2 |-> "aborted"],
[r1 |-> "aborted", r2 |-> "committed"],
[r1 |-> "committed", r2 |-> "waiting"],
[r1 |-> "committed", r2 |-> "prepared"],
[r1 |-> "committed", r2 |-> "aborted"],
[r1 |-> "committed", r2 |-> "committed"] }
```

238-8

```
TCTypeOK ==
rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
```

```
TCTypeOK ==
rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
```

- **rmState** is the only variable we're using in this spec

239-1

239-2

```
TCTypeOK ==
rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
```

- **rmState** is the only variable we're using in this spec
- It's a record, indexed on resource manager names ("rm1", "rm2", etc.)

239-3

```
TCTypeOK ==
rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
```

- **rmState** is the only variable we're using in this spec
- It's a record, indexed on resource manager names ("rm1", "rm2", etc.)
- Thus we're saying with this invariant:

239-4

```
TCTypeOK ==
rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
```

- **rmState** is the only variable we're using in this spec
- It's a record, indexed on resource manager names ("rm1", "rm2", etc.)
- Thus we're saying with this invariant:
  - The "type" of **rmState** is one such that **rmState** will always be one of the records in this set

239-5

```
TCTypeOK ==
rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
```

- **rmState** is the only variable we're using in this spec
- It's a record, indexed on resource manager names ("rm1", "rm2", etc.)
- Thus we're saying with this invariant:
  - The "type" of **rmState** is one such that **rmState** will always be one of the records in **this set**

239-6

```
TCTypeOK ==
rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
```

- **rmState** is the only variable we're using in this spec
- It's a record, indexed on resource manager names ("rm1", "rm2", etc.)
- Thus we're saying with this invariant:
  - The "type" of **rmState** is one such that **rmState** will always be one of the records in this set

239-7

## Types

- TLA+ doesn't have an elaborate type system, on purpose

240-2

## Types

- TLA+ doesn't have an elaborate type system, on purpose
- The usual technique to check types is create an invariant, asserting that a particular variable is always a subset of something

240-3

# Types

- TLA+ doesn't have an elaborate type system, on purpose
- The usual technique to check types is create an invariant, asserting that a particular variable is always a subset of something
- That "something" is usually "every possible value the variable should be able to take"

240-4

# Types

- TLA+ doesn't have an elaborate type system, on purpose
- The usual technique to check types is create an invariant, asserting that a particular variable is always a subset of something
- That "something" is usually "every possible value the variable should be able to take"
- A C type declaration of `int i;` could be modelled in TLA+ as

240-5

# Types

- TLA+ doesn't have an elaborate type system, on purpose
- The usual technique to check types is create an invariant, asserting that a particular variable is always a subset of something
- That "something" is usually "every possible value the variable should be able to take"
- A C type declaration of `int i;` could be modelled in TLA+ as
  - `i \in Integers`

240-6

# Types

- TLA+ doesn't have an elaborate type system, on purpose
- The usual technique to check types is create an invariant, asserting that a particular variable is always a subset of something
- That "something" is usually "every possible value the variable should be able to take"
- A C type declaration of `int i;` could be modelled in TLA+ as
  - `i \in Integers`
  - Where `Integers` is the set of all integer values

240-7

```
TCInit == rmState = [r \in RM |-> "working"]
```

241-1

```
TCInit == rmState = [r \in RM |-> "working"]
```

- This is where we define the “initial” state. Normally it’s called “Init”, but here we’re calling it “**TCInit**.”

241-2

```
TCInit == rmState = [r \in RM |-> "working"]
```

- This is where we define the “initial” state. Normally it’s called “Init”, but here we’re calling it “**TCInit**.”
- **[r \in RM |-> "working"]** is the single record with “working” assigned to each resource manager.

241-3

```
TCInit == rmState = [r \in RM |-> "working"]
```

- This is where we define the “initial” state. Normally it’s called “Init”, but here we’re calling it “**TCInit**.”
- **[r \in RM |-> "working"]** is the single record with “working” assigned to each resource manager.

**[r1 |-> "working", r2 |-> "working"]**

241-4

```
TCInit == rmState = [r \in RM |-> "working"]
```

- This is where we define the “initial” state. Normally it’s called “Init”, but here we’re calling it “**TCInit**.”
- **[r \in RM |-> "working"]** is the single record with “working” assigned to each resource manager.  
  
[r1 |-> “working”, r2 |-> “working”]
- There are an infinite number of initial states. This formula is restricting them to the ones where **rmState** is equal to the above record

241-5

```
TCInit == rmState = [r \in RM |-> "working"]
```

- This is where we define the “initial” state. Normally it’s called “Init”, but here we’re calling it “**TCInit**.”
- **[r \in RM |-> "working"]** is the single record with “working” assigned to each resource manager.  
  
[r1 |-> “working”, r2 |-> “working”]
- There are an infinite number of initial states. This formula is restricting them to the ones where **rmState** is equal to the above record
- Thus only one possible initial state is permitted

241-6

## Initial State

242-1

## Initial State

```
rmState: [r1 |-> “working”, r2 |-> “working”]
```

242-2

## Next state formula

$\text{TCNext} == \exists r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$

243-1

## Next state formula

$\text{TCNext} == \exists r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$

- Let's skip a bunch and scroll down to  $\text{TCNext}$ , the “next state formula”

243-2

## Next state formula

$\text{TCNext} == \exists r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$

- Let's skip a bunch and scroll down to  $\text{TCNext}$ , the “next state formula”
- This says:

243-3

## Next state formula

$\text{TCNext} == \exists r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$

- Let's skip a bunch and scroll down to  $\text{TCNext}$ , the “next state formula”
- This says:
  - “There exists an  $r$  in  $\text{RM}$  such that either  $\text{Prepare}(r)$  is TRUE or  $\text{Decide}(r)$  is TRUE”

243-4

# Next state formula

TCNext ==  $\exists r \in RM : Prepare(r) \vee Decide(r)$

- Let's skip a bunch and scroll down to TCNext, the "next state formula"
- This says:
  - "There exists an  $r$  in  $RM$  such that either **Prepare( $r$ )** is TRUE or **Decide( $r$ )** is TRUE"
- Remember the fundamental concept of TLA+

243-5

# Next state formula

TCNext ==  $\exists r \in RM : Prepare(r) \vee Decide(r)$

- Let's skip a bunch and scroll down to TCNext, the "next state formula"
- This says:
  - "There exists an  $r$  in  $RM$  such that either **Prepare( $r$ )** is TRUE or **Decide( $r$ )** is TRUE"
- Remember the fundamental concept of TLA+
- Every single behaviour in the infinite universe of possibilities is generated, our formulas are used to specify a subset of behaviours

243-6

# Next state formula

TCNext ==  $\exists r \in RM : Prepare(r) \vee Decide(r)$

- Let's skip a bunch and scroll down to TCNext, the "next state formula"
- This says:
  - "There exists an  $r$  in  $RM$  such that either **Prepare( $r$ )** is TRUE or **Decide( $r$ )** is TRUE"
- Remember the fundamental concept of TLA+
- Every single behaviour in the infinite universe of possibilities is generated, our formulas are used to specify a subset of behaviours
- At every state, an infinite number of next states are theoretically possible. Our formula identifies which next states are **actually** possible, by applying the infinite set of next states to the formula and seeing which ones come out as TRUE

243-7

# Next state formula

TCNext ==  $\exists r \in RM : Prepare(r) \vee Decide(r)$

244-1

## Next state formula

```
TCNext == \E r \in RM : Prepare(r) \vee Decide(r)
```

- Let's look at `Prepare(r)` to get some clarity

244-2

## Next state formula

```
TCNext == \E r \in RM : Prepare(r) \vee Decide(r)
```

- Let's look at `Prepare(r)` to get some clarity

```
Prepare(r) == \wedge rmState[r] = "working"
 \wedge rmState' = [rmState EXCEPT ![r] = "prepared"]
```

244-3

## Next state formula

```
TCNext == \E r \in RM : Prepare(r) \vee Decide(r)
```

- Let's look at `Prepare(r)` to get some clarity

```
Prepare(r) == \wedge rmState[r] = "working"
 \wedge rmState' = [rmState EXCEPT ![r] = "prepared"]
```

- This says:

244-4

## Next state formula

```
TCNext == \E r \in RM : Prepare(r) \vee Decide(r)
```

- Let's look at `Prepare(r)` to get some clarity

```
Prepare(r) == \wedge rmState[r] = "working"
 \wedge rmState' = [rmState EXCEPT ![r] = "prepared"]
```

- This says:

- This formula is true for a pair of states if in the current state, resource manager `r` is "working"

244-5

# Next state formula

TCNext ==  $\forall r \in RM : Prepare(r) \vee Decide(r)$

- Let's look at `Prepare(r)` to get some clarity

`Prepare(r) ==  $\wedge rmState[r] = "working"$   
 $\wedge rmState' = [rmState EXCEPT !r = "prepared"]$`

- This says:

- This formula is true for a pair of states if in the `current state`, resource manager `r` is "working"
- And if in the `next state` the state of resource manager `r` is "prepared", and all the other resource managers have the same state as they do right now

244-6

# Next state formula

TCNext ==  $\forall r \in RM : Prepare(r) \vee Decide(r)$

`Prepare(r) ==  $\wedge rmState[r] = "working"$   
 $\wedge rmState' = [rmState EXCEPT !r = "prepared"]$`

`rmState: [r1 -> "prepared",  
r2 -> "working"]`

245-1

# Next state formula

TCNext ==  $\forall r \in RM : Prepare(r) \vee Decide(r)$

`Prepare(r) ==  $\wedge rmState[r] = "working"$   
 $\wedge rmState' = [rmState EXCEPT !r = "prepared"]$`

`rmState: [r1 -> "prepared",  
r2 -> "working"]` →

245-2

# Next state formula

TCNext ==  $\forall r \in RM : Prepare(r) \vee Decide(r)$

`Prepare(r) ==  $\wedge rmState[r] = "working"$   
 $\wedge rmState' = [rmState EXCEPT !r = "prepared"]$`

`rmState: [r1 -> "prepared",  
r2 -> "working"]` → ? `rmState: [r1 -> "committed",  
r2 -> "working"]`

245-3

# Next state formula

TCNext ==  $\exists r \in RM : Prepare(r) \vee Decide(r)$

Prepare(r) ==  $\wedge rmState[r] = "working"$   
 $\wedge rmState' = [rmState \text{ EXCEPT } !r] = "prepared"$



245-4

# Next state formula

TCNext ==  $\exists r \in RM : Prepare(r) \vee Decide(r)$

Prepare(r) ==  $\wedge rmState[r] = "working"$   
 $\wedge rmState' = [rmState \text{ EXCEPT } !r] = "prepared"$

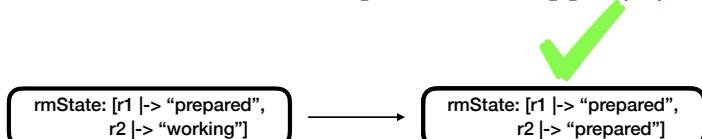


245-5

# Next state formula

TCNext ==  $\exists r \in RM : Prepare(r) \vee Decide(r)$

Prepare(r) ==  $\wedge rmState[r] = "working"$   
 $\wedge rmState' = [rmState \text{ EXCEPT } !r] = "prepared"$

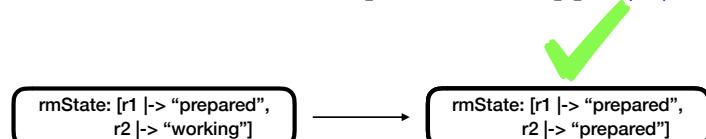


245-6

# Next state formula

TCNext ==  $\exists r \in RM : Prepare(r) \vee Decide(r)$

Prepare(r) ==  $\wedge rmState[r] = "working"$   
 $\wedge rmState' = [rmState \text{ EXCEPT } !r] = "prepared"$



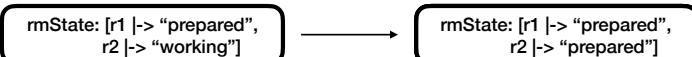
- The parts of the formula with no primed variables are called **enabling** conditions

245-7

# Next state formula

TCNext ==  $\forall r \in RM : Prepare(r) \vee Decide(r)$

Prepare(r) ==  $\wedge rmState[r] = "working" \leftarrow$   
 $\wedge rmState' = [rmState \text{ EXCEPT } !r] = "prepared"$



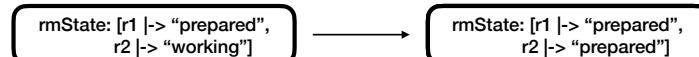
- The parts of the formula with no primed variables are called **enabling conditions**

245-8

# Next state formula

TCNext ==  $\forall r \in RM : Prepare(r) \vee Decide(r)$

Prepare(r) ==  $\wedge rmState[r] = "working" \leftarrow$   
 $\wedge rmState' = [rmState \text{ EXCEPT } !r] = "prepared"$



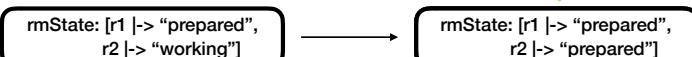
- The parts of the formula with no primed variables are called **enabling conditions**
- They describe what the **current state** must look like to be able to use this action

245-9

# Next state formula

TCNext ==  $\forall r \in RM : Prepare(r) \vee Decide(r)$

Prepare(r) ==  $\wedge rmState[r] = "working" \leftarrow$   
 $\wedge rmState' = [rmState \text{ EXCEPT } !r] = "prepared"$



- The parts of the formula with no primed variables are called **enabling conditions**
- They describe what the **current state** must look like to be able to use this action
- Any formula with **primed variables** is called an **action**

245-10

# Next state formula

TCNext ==  $\forall r \in RM : Prepare(r) \vee Decide(r)$

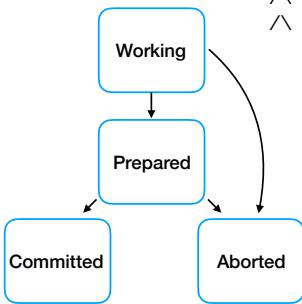
Decide(r) ==  $\vee \wedge rmState[r] = "prepared"$   
 $\wedge canCommit$   
 $\wedge rmState' = [rmState \text{ EXCEPT } !r] = "committed"$   
 $\vee \wedge rmState[r] \in \{"working", "prepared"\}$   
 $\wedge notCommitted$   
 $\wedge rmState' = [rmState \text{ EXCEPT } !r] = "aborted"$

246-1

# Next state formula

$\text{TCNext} == \forall r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$

$\text{Decide}(r) == \forall \wedge \text{rmState}[r] = \text{"prepared"} \wedge \text{canCommit} \wedge \text{rmState}' = [\text{rmState EXCEPT } !r = \text{"committed"}] \vee \forall \wedge \text{rmState}[r] \in \{\text{"working"}, \text{"prepared"}\} \wedge \text{notCommitted} \wedge \text{rmState}' = [\text{rmState EXCEPT } !r = \text{"aborted"}]$

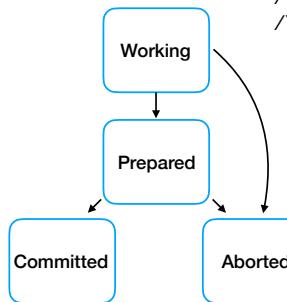


246-2

# Next state formula

$\text{TCNext} == \forall r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$

$\text{Decide}(r) == \forall \wedge \text{rmState}[r] = \text{"prepared"} \wedge \text{canCommit} \wedge \text{rmState}' = [\text{rmState EXCEPT } !r = \text{"committed"}] \vee \forall \wedge \text{rmState}[r] \in \{\text{"working"}, \text{"prepared"}\} \wedge \text{notCommitted} \wedge \text{rmState}' = [\text{rmState EXCEPT } !r = \text{"aborted"}]$

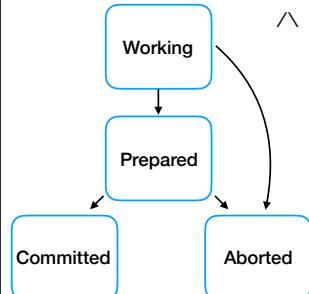


246-3

# Next state formula

$\text{TCNext} == \forall r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$

$\text{Decide}(r) == \forall \wedge \text{rmState}[r] = \text{"prepared"} \wedge \text{canCommit} \wedge \text{rmState}' = [\text{rmState EXCEPT } !r = \text{"committed"}] \vee \forall \wedge \text{rmState}[r] \in \{\text{"working"}, \text{"prepared"}\} \wedge \text{notCommitted} \wedge \text{rmState}' = [\text{rmState EXCEPT } !r = \text{"aborted"}]$



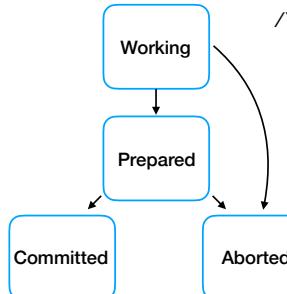
• This formula is true when either

246-4

# Next state formula

$\text{TCNext} == \forall r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$

$\text{Decide}(r) == \forall \wedge \text{rmState}[r] = \text{"prepared"} \wedge \text{canCommit} \wedge \text{rmState}' = [\text{rmState EXCEPT } !r = \text{"committed"}] \vee \forall \wedge \text{rmState}[r] \in \{\text{"working"}, \text{"prepared"}\} \wedge \text{notCommitted} \wedge \text{rmState}' = [\text{rmState EXCEPT } !r = \text{"aborted"}]$



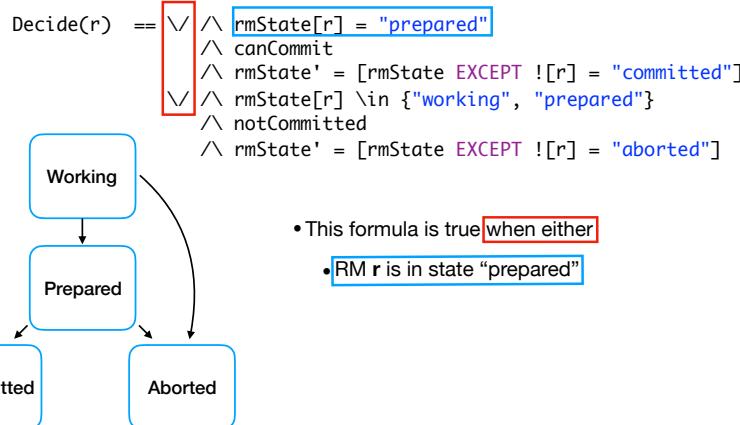
• This formula is true when either

• RM r is in state "prepared"

246-5

# Next state formula

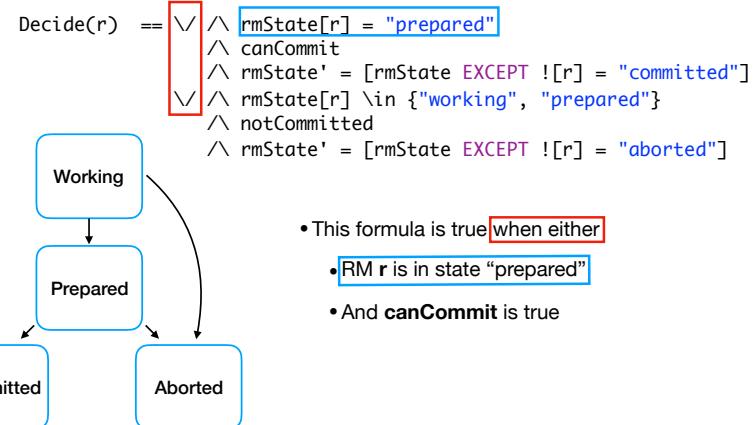
$\text{TCNext} == \exists r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$



246-6

# Next state formula

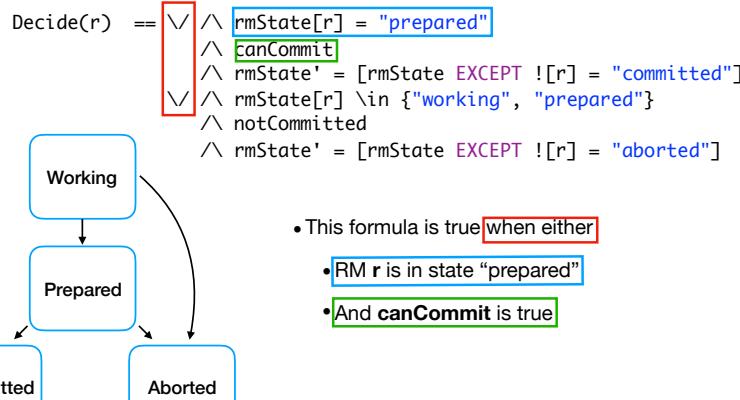
$\text{TCNext} == \exists r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$



246-7

# Next state formula

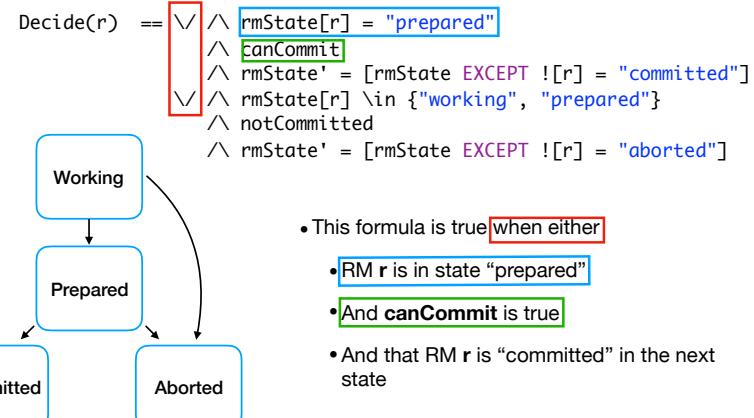
$\text{TCNext} == \exists r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$



246-8

# Next state formula

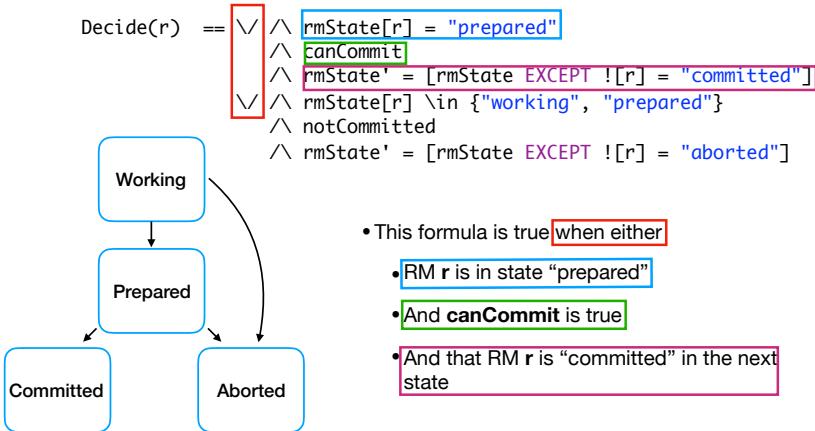
$\text{TCNext} == \exists r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$



246-9

# Next state formula

$\text{TCNext} == \forall r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$

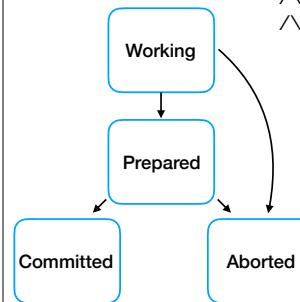


246-10

# Next state formula

$\text{TCNext} == \forall r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$

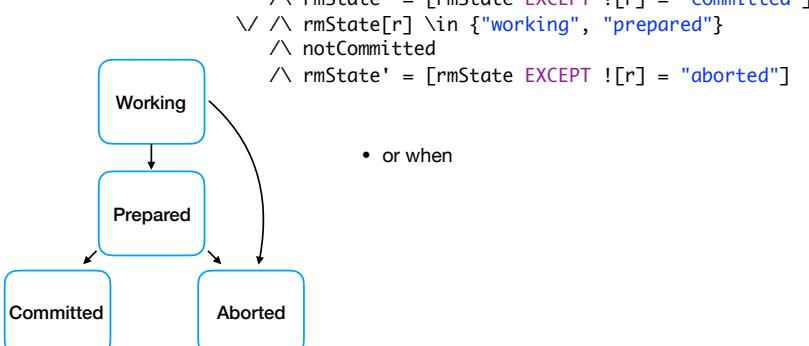
Decide( $r$ ) ==  $\vee \wedge \text{rmState}[r] = \text{"prepared"} \wedge \text{canCommit} \wedge \text{rmState}' = [\text{rmState EXCEPT } !r = \text{"committed"}] \vee \wedge \text{rmState}[r] \in \{\text{"working", "prepared"}\} \wedge \text{notCommitted} \wedge \text{rmState}' = [\text{rmState EXCEPT } !r = \text{"aborted"}]$



247-1

# Next state formula

$\text{TCNext} == \forall r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$

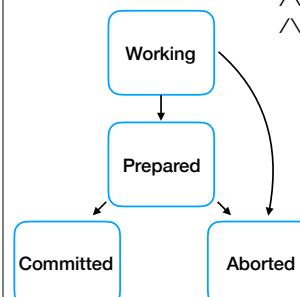


247-2

# Next state formula

$\text{TCNext} == \forall r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$

Decide( $r$ ) ==  $\vee \wedge \text{rmState}[r] = \text{"prepared"} \wedge \text{canCommit} \wedge \text{rmState}' = [\text{rmState EXCEPT } !r = \text{"committed"}] \vee \wedge \text{rmState}[r] \in \{\text{"working", "prepared"}\} \wedge \text{notCommitted} \wedge \text{rmState}' = [\text{rmState EXCEPT } !r = \text{"aborted"}]$

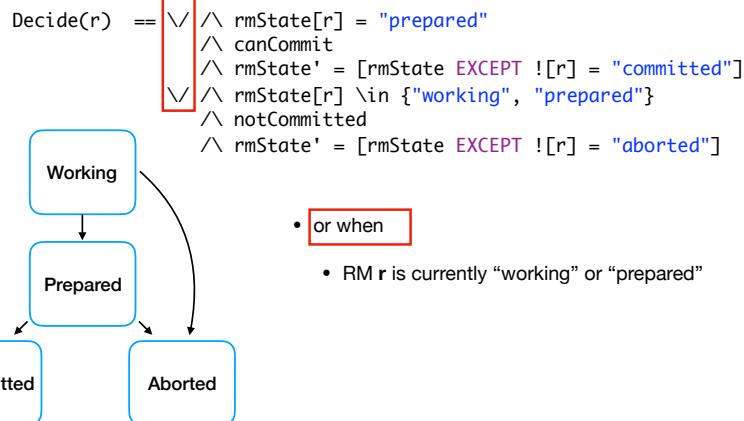


• or when

247-3

# Next state formula

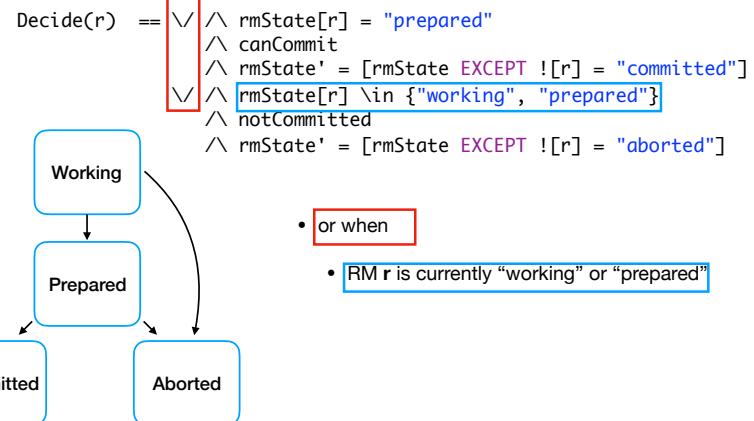
$TCNext == \exists r \in RM : Prepare(r) \vee Decide(r)$



247-4

# Next state formula

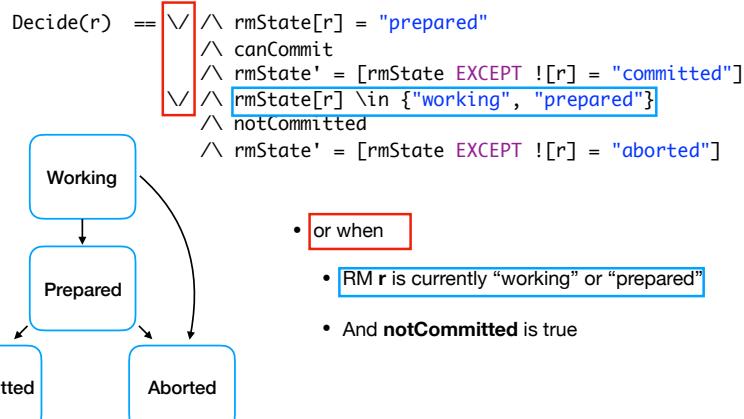
$TCNext == \exists r \in RM : Prepare(r) \vee Decide(r)$



247-5

# Next state formula

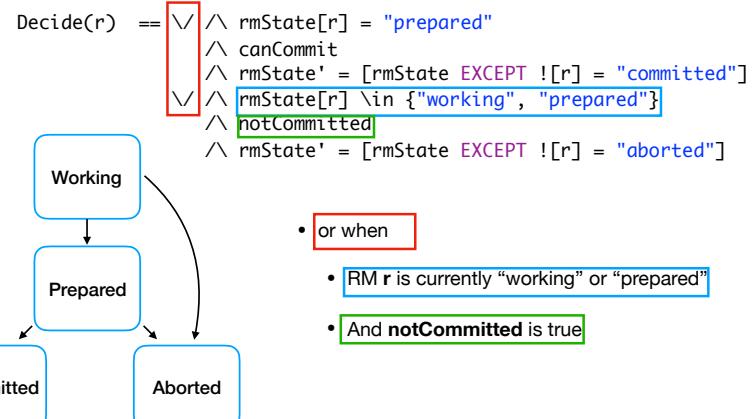
$TCNext == \exists r \in RM : Prepare(r) \vee Decide(r)$



247-6

# Next state formula

$TCNext == \exists r \in RM : Prepare(r) \vee Decide(r)$

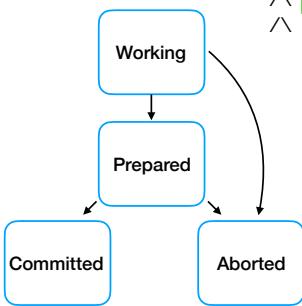


247-7

# Next state formula

$\text{TCNext} == \exists r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$

$\text{Decide}(r) == \begin{array}{l} \vee \wedge \text{rmState}[r] = \text{"prepared"} \\ \wedge \text{canCommit} \\ \wedge \text{rmState}' = [\text{rmState EXCEPT } !r = \text{"committed"}] \\ \vee \wedge \text{rmState}[r] \in \{\text{"working", "prepared"}\} \\ \wedge \text{notCommitted} \\ \wedge \text{rmState}' = [\text{rmState EXCEPT } !r = \text{"aborted"}] \end{array}$

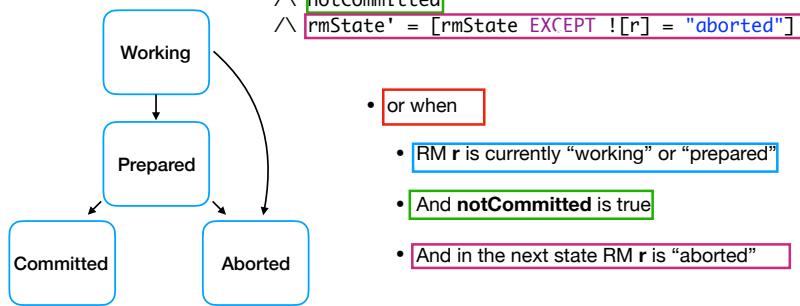


247-8

# Next state formula

$\text{TCNext} == \exists r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$

$\text{Decide}(r) == \begin{array}{l} \vee \wedge \text{rmState}[r] = \text{"prepared"} \\ \wedge \text{canCommit} \\ \wedge \text{rmState}' = [\text{rmState EXCEPT } !r = \text{"committed"}] \\ \vee \wedge \text{rmState}[r] \in \{\text{"working", "prepared"}\} \\ \wedge \text{notCommitted} \\ \wedge \text{rmState}' = [\text{rmState EXCEPT } !r = \text{"aborted"}] \end{array}$



247-9

# canCommit

$\text{canCommit} == \forall r \in \text{RM} : \text{rmState}[r] \in \{\text{"prepared", "committed"}\}$

248-1

# canCommit

$\text{canCommit} == \forall r \in \text{RM} : \text{rmState}[r] \in \{\text{"prepared", "committed"}\}$

## Python Equivalents

```

RM = ["rm1", "rm2"] # a constant
rmState = {"rm1": "working", "rm2": "working"}
canCommitStates = ["prepared", "committed"] # a constant
...
def canCommit():
 for r in RM:
 if rmState[r] not in canCommitStates:
 return False
 return True

def canCommit():
 return all(lambda r: rmState[r] in canCommitStates, RM))

```

248-2

# canCommit

```
canCommit == \A r \in RM : rmState[r] \in {"prepared", "committed"}
```

## Python Equivalents

```
RM = ["rm1", "rm2"] # a constant
rmState = {"rm1": "working", "rm2": "working"}
canCommitStates = ["prepared", "committed"] # a constant

...
def canCommit():
 for r in RM:
 if rmState[r] not in canCommitStates:
 return False
 return True

def canCommit():
 return all(map(lambda r: rmState[r] in canCommitStates, RM))
```

248-3

# canCommit

```
canCommit == \A r \in RM : rmState[r] \in {"prepared", "committed"}
```

## Python Equivalents

```
RM = ["rm1", "rm2"] # a constant
rmState = {"rm1": "working", "rm2": "working"}
canCommitStates = ["prepared", "committed"] # a constant

...
def canCommit():
 for r in RM:
 if rmState[r] not in canCommitStates:
 return False
 return True

def canCommit():
 return all(map(lambda r: rmState[r] in canCommitStates, RM))
```

248-4

# canCommit

```
canCommit == \A r \in RM : rmState[r] \in {"prepared", "committed"}
```

## Python Equivalents

```
RM = ["rm1", "rm2"] # a constant
rmState = {"rm1": "working", "rm2": "working"}
canCommitStates = ["prepared", "committed"] # a constant

...
def canCommit():
 for r in RM:
 if rmState[r] not in canCommitStates:
 return False
 return True

def canCommit():
 return all(map(lambda r: rmState[r] in canCommitStates, RM))
```

248-5

# notCommitted

```
notCommitted == \A r \in RM : rmState[r] # "committed"
```

"Not equal." You can also use /=

249-1

# notCommitted

```
notCommitted == \A r \in RM : rmState[r] # "committed"

Python Equivalents

RM = ["rm1", "rm2"]
rmState = {"rm1": "working", "rm2": "working"}

...
"Not equal." You can also use /=

def notCommitted():
 for r in RM:
 if rmState[r] == "committed":
 return False
 return True

def notCommitted():
 return all(map(lambda r: rmState[r] != "committed", RM))
```

249-2

# notCommitted

```
notCommitted == \A r \in RM : rmState[r] # "committed"

Python Equivalents

RM = ["rm1", "rm2"]
rmState = {"rm1": "working", "rm2": "working"}

...
"Not equal." You can also use /=

def notCommitted():
 for r in RM:
 if rmState[r] == "committed":
 return False
 return True

def notCommitted():
 return all(map(lambda r: rmState[r] != "committed", RM))
```

249-4

# notCommitted

```
notCommitted == \A r \in RM : rmState[r] # "committed"

Python Equivalents

RM = ["rm1", "rm2"]
rmState = {"rm1": "working", "rm2": "working"}

...
"Not equal." You can also use /=

def notCommitted():
 for r in RM:
 if rmState[r] == "committed":
 return False
 return True

def notCommitted():
 return all(map(lambda r: rmState[r] != "committed", RM))
```

249-3

# notCommitted

```
notCommitted == \A r \in RM : rmState[r] # "committed"

Python Equivalents

RM = ["rm1", "rm2"]
rmState = {"rm1": "working", "rm2": "working"}

...
"Not equal." You can also use /=

def notCommitted():
 for r in RM:
 if rmState[r] == "committed":
 return False
 return True

def notCommitted():
 return all(map(lambda r: rmState[r] != "committed", RM))
```

249-5

# Next state formula

$\text{TCNext} == \exists r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$

250-1

# Next state formula

$\text{TCNext} == \exists r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$

- Back to this, **TCNext** is true if there is some **r** in **RM** such that either **Prepare(r)** is TRUE, or **Decide(r)** is TRUE.

250-2

# Next state formula

$\text{TCNext} == \exists r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$

- Back to this, **TCNext** is true if there is some **r** in **RM** such that either **Prepare(r)** is TRUE, or **Decide(r)** is TRUE.
- If **multiple r** in RM satisfy that, TLC will create behaviours for all of them

250-3

# Next state formula

$\text{TCNext} == \exists r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$

- Back to this, **TCNext** is true if there is some **r** in **RM** such that either **Prepare(r)** is TRUE, or **Decide(r)** is TRUE.
- If **multiple r** in RM satisfy that, TLC will create behaviours for all of them
- TLC always explores every possible state

250-4

# TCConsistent

```
TCConsistent ==
 \A r1, r2 \in RM : ~ \wedge rmState[r1] = "aborted"
 \wedge rmState[r2] = "committed"
```

251-1

# TCConsistent

```
TCConsistent ==
 \A r1, r2 \in RM : ~ \wedge rmState[r1] = "aborted"
 \wedge rmState[r2] = "committed"
```

- For every possible pair of resource managers r1 and r2

251-2

# TCConsistent

```
TCConsistent ==
 \A r1, r2 \in RM : ~ \wedge rmState[r1] = "aborted"
 \wedge rmState[r2] = "committed"
```

- For every possible pair of resource managers r1 and r2
- It should not be the case that

251-3

# TCConsistent

```
TCConsistent ==
 \A r1, r2 \in RM : ~ \wedge rmState[r1] = "aborted"
 \wedge rmState[r2] = "committed"
```

- For every possible pair of resource managers r1 and r2
- It should not be the case that
- rmState[r1] = "aborted"

251-4

# TCConsistent

```
TCConsistent ==
 \A r1, r2 \in RM : ~ \wedge rmState[r1] = "aborted"
 \wedge rmState[r2] = "committed"
```

- For every possible pair of resource managers r1 and r2
- It should not be the case that
- `rmState[r1] = "aborted"`
- `And rmState[r2] = "committed"`

251-5

# TCConsistent

```
TCConsistent ==
 \A r1, r2 \in RM : ~ \wedge rmState[r1] = "aborted"
 \wedge rmState[r2] = "committed"
```

252-1

# TCConsistent

```
TCConsistent ==
 \A r1, r2 \in RM : ~ \wedge rmState[r1] = "aborted"
 \wedge rmState[r2] = "committed"
```

Python equivalent

```
def TCConsistent():
 for r1 in RM:
 for r2 in RM:
 if r1 == "aborted" and r2 == "committed"
 return False
 return True
```

252-2

# TCommit Recap

253-1

# TCommit Recap

- We have specified **what** a “transaction commit” system should do

253-2

# TCommit Recap

- We have specified **what** a “transaction commit” system should do
  - All RMs must agree on committing or aborting a transaction

253-3

# TCommit Recap

- We have specified **what** a “transaction commit” system should do
  - All RMs must agree on committing or aborting a transaction
  - It is never allowed that one RM is committed and another is aborted

253-4

# TCommit Recap

- We have specified **what** a “transaction commit” system should do
  - All RMs must agree on committing or aborting a transaction
  - It is never allowed that one RM is committed and another is aborted
- Our spec says **nothing** about how to actually implement such a thing, it only specifies the desired behaviours

253-5

# TCommit Recap

- We have specified **what** a “transaction commit” system should do
  - All RMs must agree on committing or aborting a transaction
  - It is never allowed that one RM is committed and another is aborted
- Our spec says **nothing** about how to actually implement such a thing, it only specifies the desired behaviours
- Now we can write a new spec which presents a solution to this problem

253-6

# TCommit Recap

- We have specified **what** a “transaction commit” system should do
  - All RMs must agree on committing or aborting a transaction
  - It is never allowed that one RM is committed and another is aborted
- Our spec says **nothing** about how to actually implement such a thing, it only specifies the desired behaviours
- Now we can write a new spec which presents a solution to this problem
  - (There are **LOTS** of algorithms that solve this problem, we'll just look at one)

253-7

# Two Phase Commit

# Two Phase Commit

- A simple solution of the Transaction Commit problem

254-1

254-2

## Two Phase Commit

- A simple solution of the Transaction Commit problem
- Introduces a “Transaction Manager” (TM) to coordinate with the RMs

254-3

## Two Phase Commit

- A simple solution of the Transaction Commit problem
- Introduces a “Transaction Manager” (TM) to coordinate with the RMs
- The TM becomes the “source of truth” on whether a transaction should commit or abort

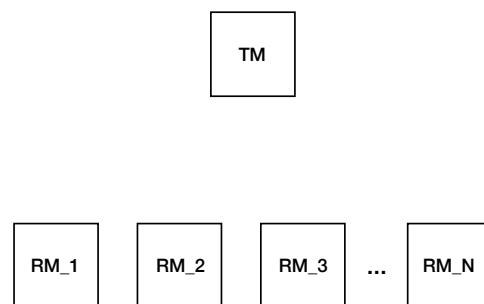
254-4

## Transaction Manager



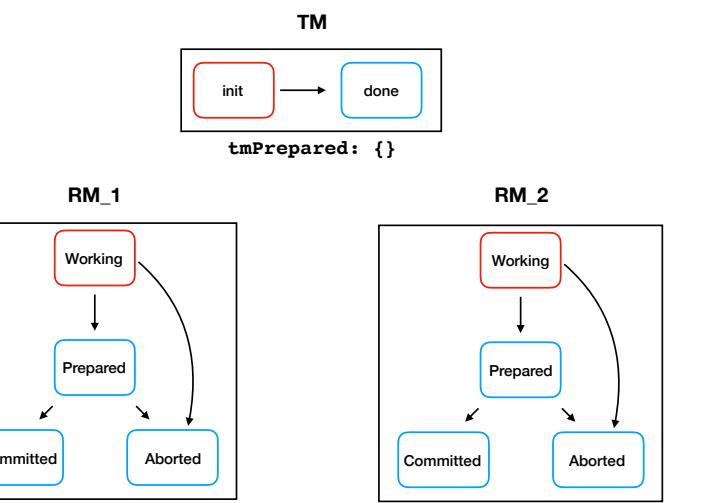
255-1

## Transaction Manager



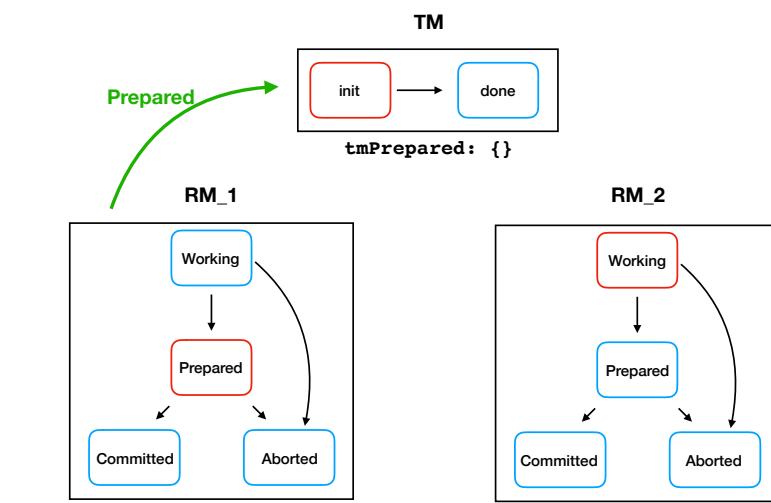
255-2

# Transaction Manager



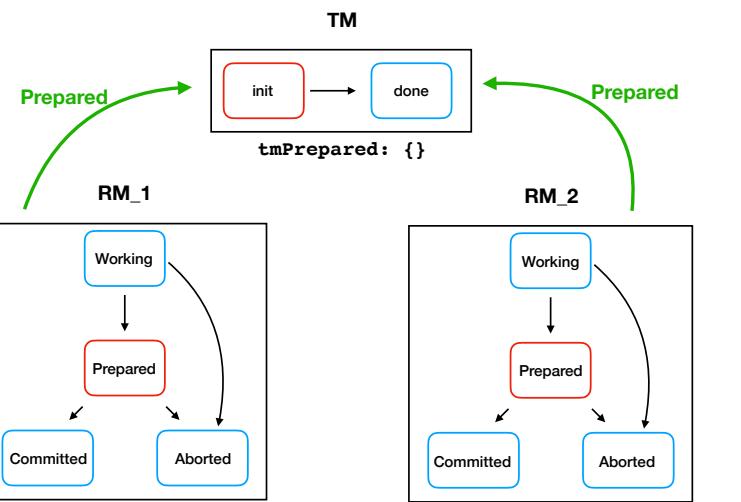
256

# Transaction Manager



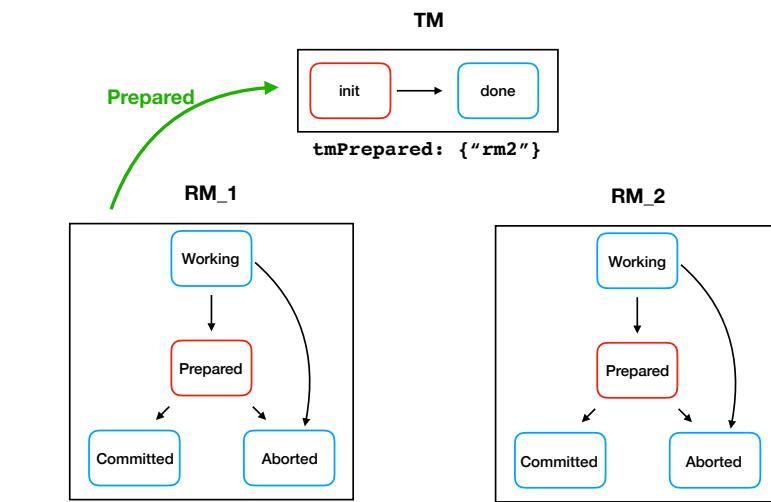
257

# Transaction Manager



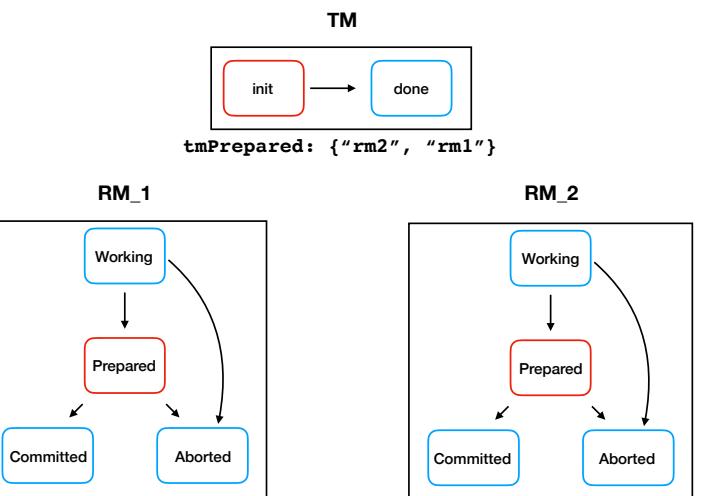
258

# Transaction Manager



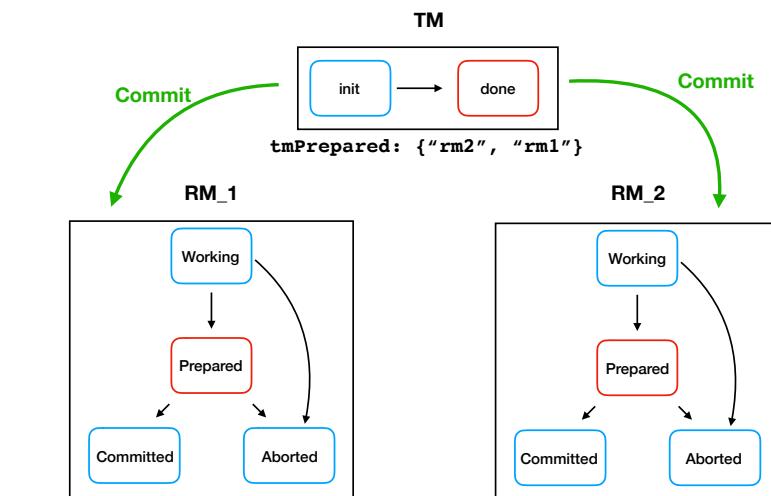
259

# Transaction Manager



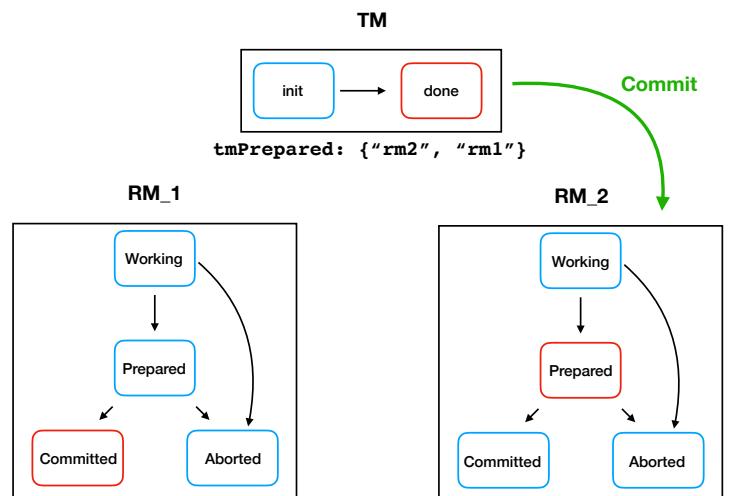
260

# Transaction Manager



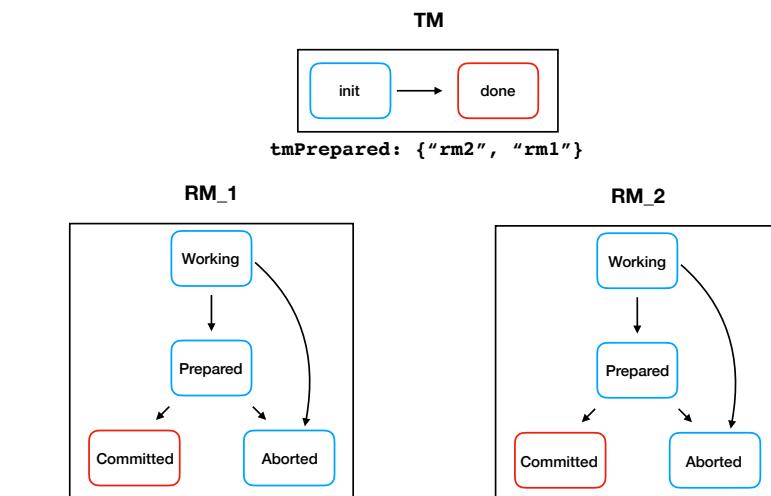
261

# Transaction Manager

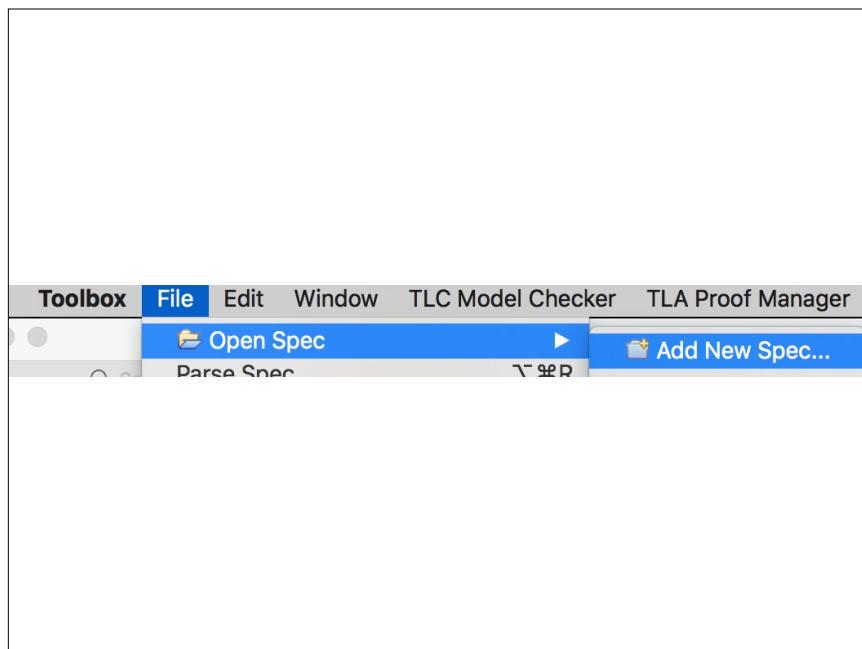


262

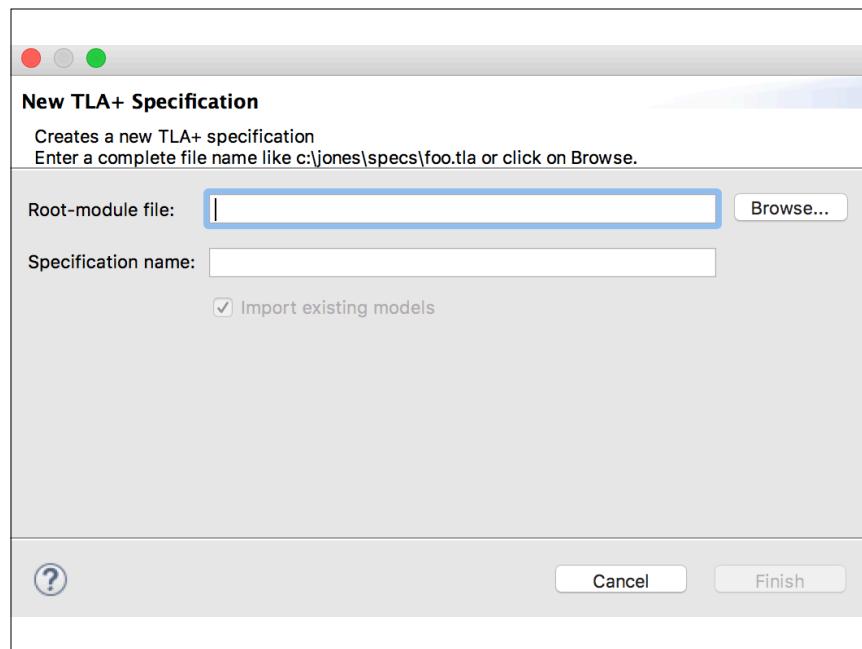
# Transaction Manager



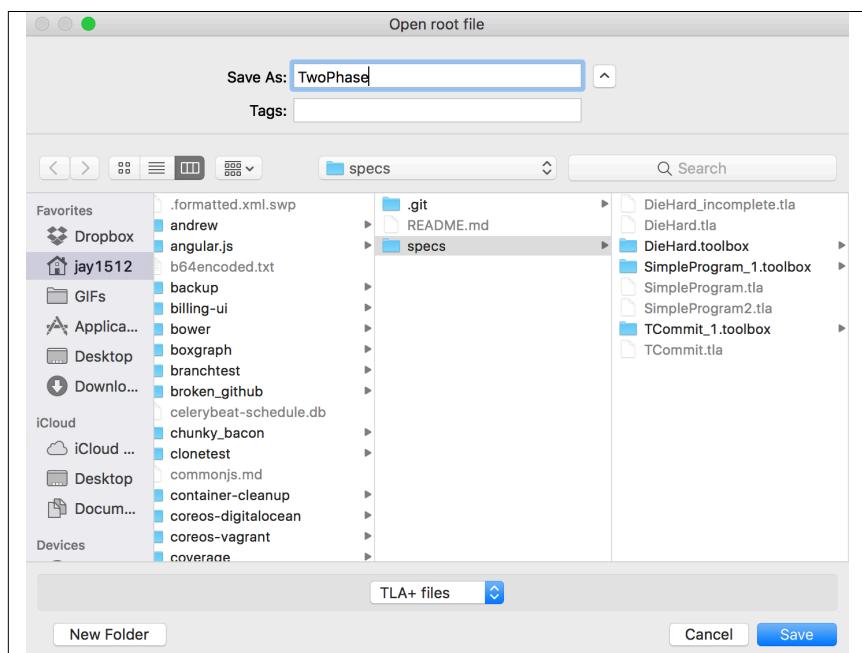
263



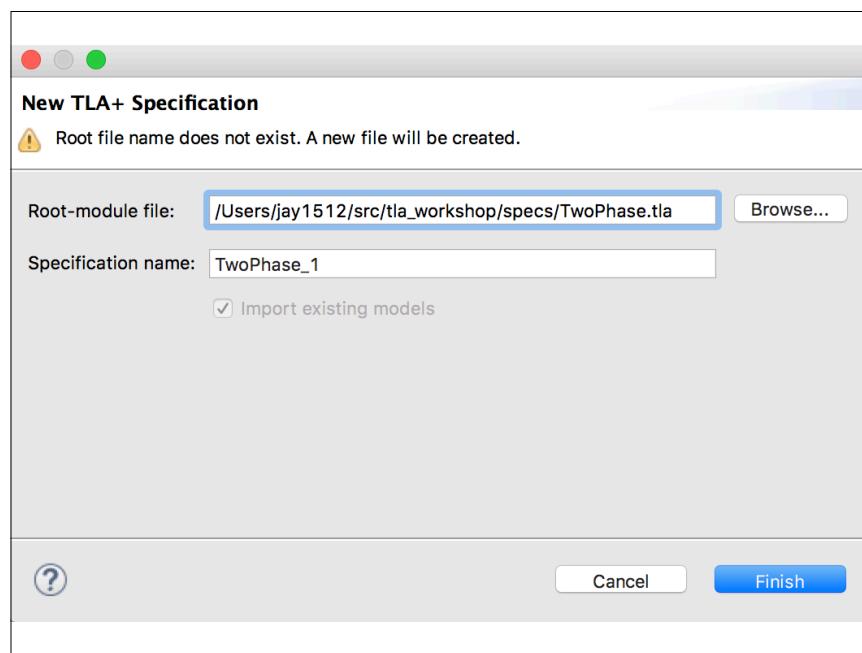
264



265



266



267

```

----- MODULE TwoPhase -----
(* This specification is discussed in "Two-Phase Commit", Lecture 6 of the *)
(* CSCI 680 course at the University of Washington. *)
(* which a transaction manager (TM) coordinates the resource managers *)
(* (RMs) to implement the Transaction Commit specification of module *)
(* TCommit. In this specification, RMs spontaneously issue Prepared *)
(* messages. We ignore the Prepare messages that the TM can send to the *)
(* RMs. *)
(*
(* For simplicity, we also eliminate Abort messages sent by an RM when it *)
(* decides to abort. Such a message would cause the TM to abort the *)
(* transaction, an event represented here by the TM spontaneously deciding *)
(* to abort. *)
----- CONSTANT RM ----- The set of resource managers
----- VARIABLES -----
rmState, /* rmState[r] is the state of resource manager r.
tmState, /* The state of the transaction manager.
tmPrepared, /* The set of RMs from which the TM has received "Prepared"
 /* messages.
----- msgs -----
(* In the protocol, processes communicate with one another by sending *)
(* messages. For simplicity, we represent message passing with the *)
(* variable msgs whose value is the set of all messages that have been *)
(* sent. A message is sent by adding it to the set msgs. An action *)
(* that, in an implementation, would be enabled by the receipt of a *)
(* certain message is here enabled by the presence of that message in *)
(* msgs. For simplicity, messages are never removed from msgs. This *)
(* allows a single message to be received by multiple receivers. *)
(* Receipt of the same message twice is therefore allowed; but in this *)
(* particular protocol, that's not a problem. *)
----- Messages ==
(* The set of all possible messages. Messages of type "Prepared" are *)
(* sent from the RM indicated by the message's rm field to the TM. *)
(* Messages of type "Commit" and "Abort" are broadcast by the TM, to be *)
(* received by all RMs. The set msgs contains just a single copy of *)
(* each a message. *)
----- [type : {"Prepared"}, rm : RM] \cup [type : {"Commit", "Abort"}]

```

specs/TwoPhase.tla

268

## Records

- Records are another syntactic sugar on top of functions, for cases where you want all the keys to be strings

## Records

- Records are another syntactic sugar on top of functions, for cases where you want all the keys to be strings

```
r == [nodes |-> {1,2}, edges |-> {"a", "b"}, cost |-> 5]
```

269-2

269-1

## Records

- Records are another syntactic sugar on top of functions, for cases where you want all the keys to be strings

```
r == [nodes |-> {1,2}, edges |-> {"a", "b"}, cost |-> 5]
r.nodes = {1,2}
```

269-3

# Records

- Records are another syntactic sugar on top of functions, for cases where you want all the keys to be strings

```
r == [nodes |-> {1,2}, edges |-> {"a", "b"}, cost |-> 5]
r.nodes = {1,2}
r.edges = {"a", "b"}
```

269-4

# Records

- Records are another syntactic sugar on top of functions, for cases where you want all the keys to be strings

```
r == [nodes |-> {1,2}, edges |-> {"a", "b"}, cost |-> 5]
r.nodes = {1,2}
r.edges = {"a", "b"}
r.cost = 5
```

269-5

# Records

- Records are another syntactic sugar on top of functions, for cases where you want all the keys to be strings

```
r == [nodes |-> {1,2}, edges |-> {"a", "b"}, cost |-> 5]
r.nodes = {1,2}
r.edges = {"a", "b"}
r.cost = 5
r["edges"] = {"a", "b"}
```

269-6

# Records

- Records are another syntactic sugar on top of functions, for cases where you want all the keys to be strings

```
r == [nodes |-> {1,2}, edges |-> {"a", "b"}, cost |-> 5]
r.nodes = {1,2}
r.edges = {"a", "b"}
r.cost = 5
r["edges"] = {"a", "b"}
```

- Equivalent to the following Python dictionary

269-7

# Records

- Records are another syntactic sugar on top of functions, for cases where you want all the keys to be strings

```
r == [nodes |-> {1,2}, edges |-> {"a", "b"}, cost |-> 5]
r.nodes = {1,2}
r.edges = {"a", "b"}
r.cost = 5
r["edges"] = {"a", "b"}
```

- Equivalent to the following Python dictionary

```
r = {"nodes": set([1,2]), "edges": set(["a", "b"]), "cost": 5}
```

269-8

# Records

- Records are another syntactic sugar on top of functions, for cases where you want all the keys to be strings

```
r == [nodes |-> {1,2}, edges |-> {"a", "b"}, cost |-> 5]
r.nodes = {1,2}
r.edges = {"a", "b"}
r.cost = 5
r["edges"] = {"a", "b"}
```

- Equivalent to the following Python dictionary

```
r = {"nodes": set([1,2]), "edges": set(["a", "b"]), "cost": 5}
```

269-9

# Sets of records

# Sets of records

- We also often need to easily create sets of records

270-1

270-2

# Sets of records

- We also often need to easily create sets of records
- The syntax for this is [key1: S, key2: T], where **S** and **T** are sets

270-3

# Sets of records

- We also often need to easily create sets of records
- The syntax for this is [key1: S, key2: T], where **S** and **T** are sets

```
[nodes: {1,2}, edges: {"a","b","c"}]
```

```
{ [nodes | -> 1, edges | -> "a"],
 [nodes | -> 1, edges | -> "b"],
 [nodes | -> 1, edges | -> "c"],
 [nodes | -> 2, edges | -> "a"],
 [nodes | -> 2, edges | -> "b"],
 [nodes | -> 2, edges | -> "c"] }
```

270-5

# Sets of records

- We also often need to easily create sets of records
- The syntax for this is [key1: S, key2: T], where **S** and **T** are sets

```
[nodes: {1,2}, edges: {"a","b","c"}]
```

270-4

# Sets of records

```
[nodes: {1,2}, edges: {"a","b","c"}]
```

- This is equivalent to the following Python:

271-1

# Sets of records

```
[nodes: {1,2}, edges: {"a","b","c"}]
```

- This is equivalent to the following Python:

```
set_of_records = set()
for node in [1,2]:
 for edge in ["a","b","c"]:
 set_of_records.add({"nodes": node, "edges", edge})
```

271-2

# Sets of records

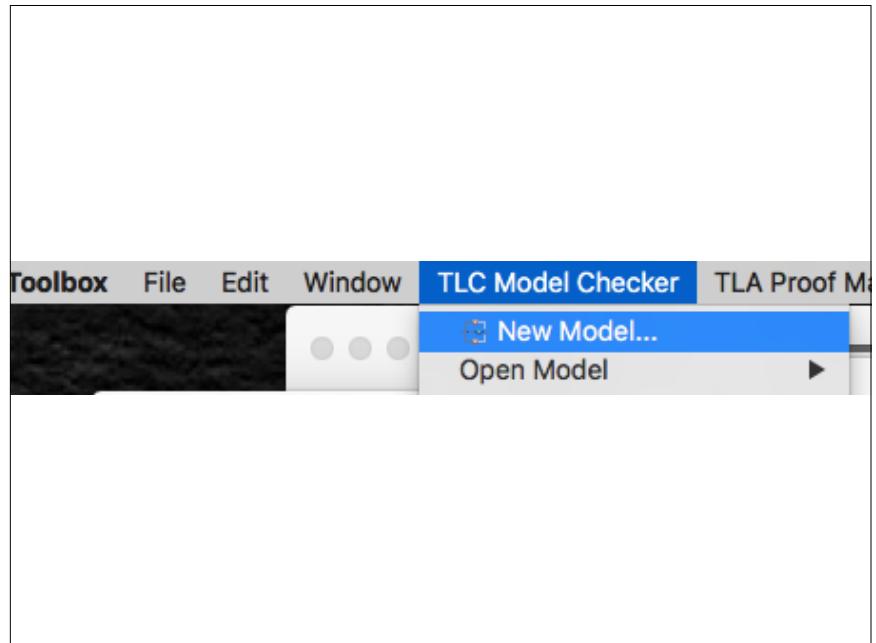
```
[nodes: {1,2}, edges: {"a","b","c"}]
```

- This is equivalent to the following Python:

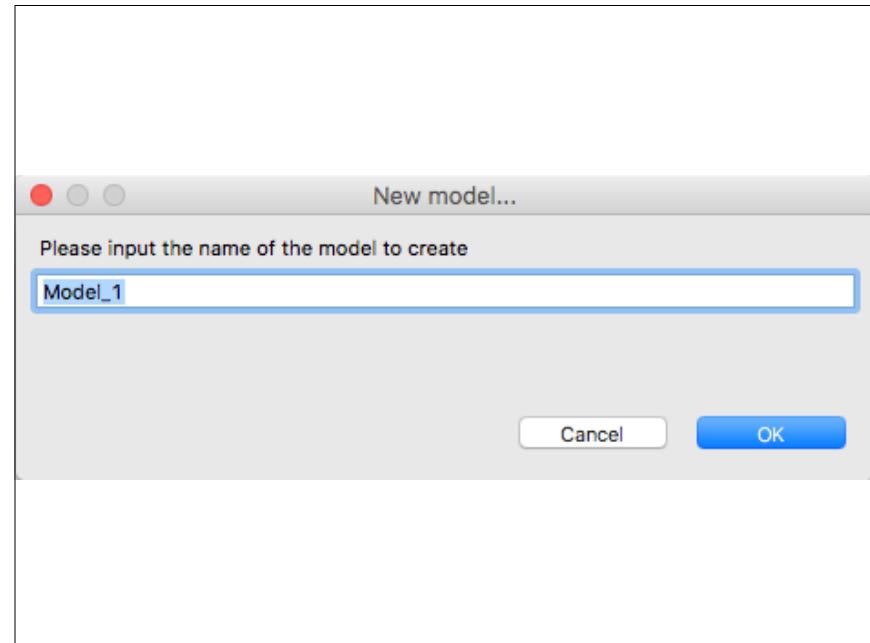
```
set_of_records = set()
for node in [1,2]:
 for edge in ["a","b","c"]:
 set_of_records.add({"nodes": node, "edges", edge})
```

```
{ {"nodes": 1, "edges": "a"},
 {"nodes": 1, "edges": "b"},
 {"nodes": 1, "edges": "c"},
 {"nodes": 2, "edges": "a"},
 {"nodes": 2, "edges": "b"},
 {"nodes": 2, "edges": "c"} }
```

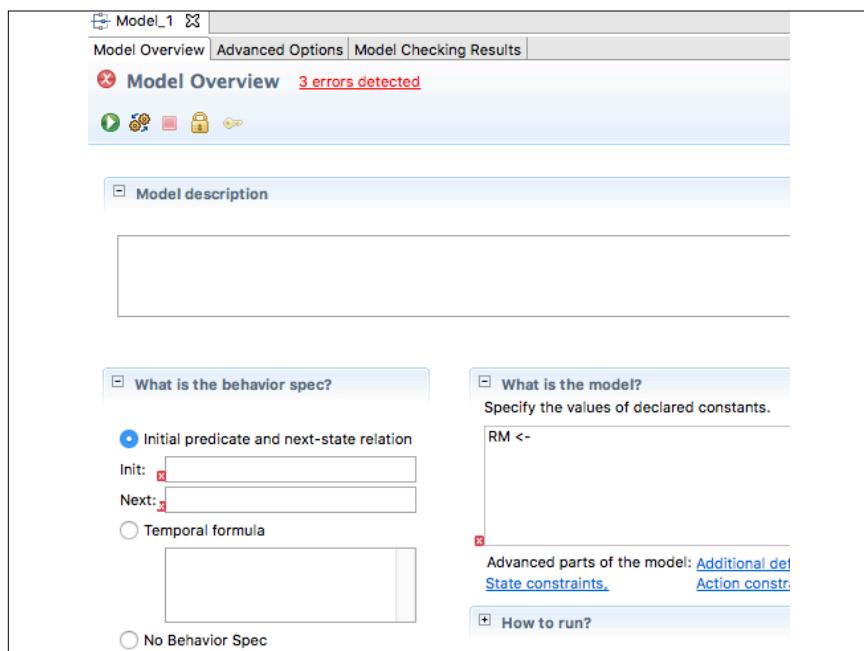
271-3



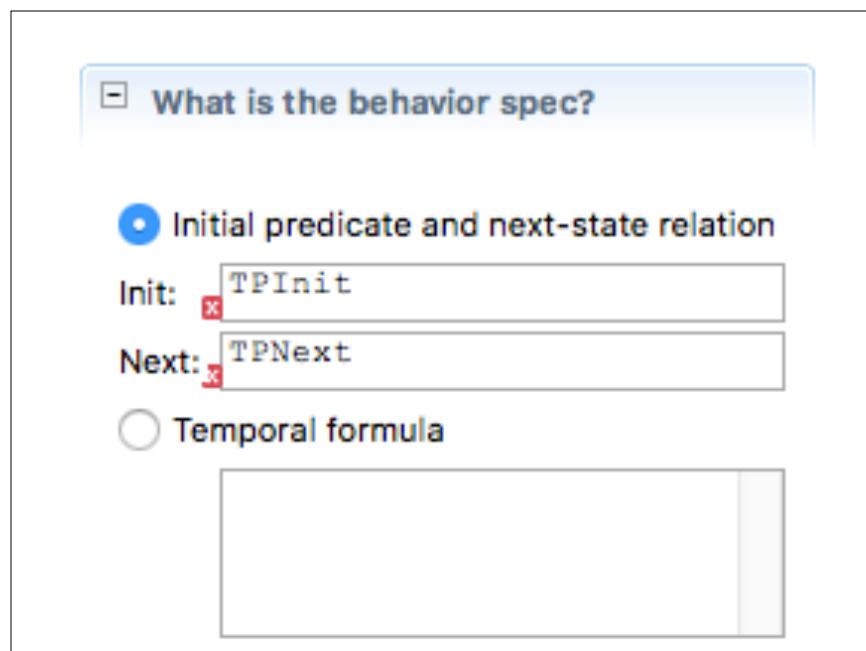
272



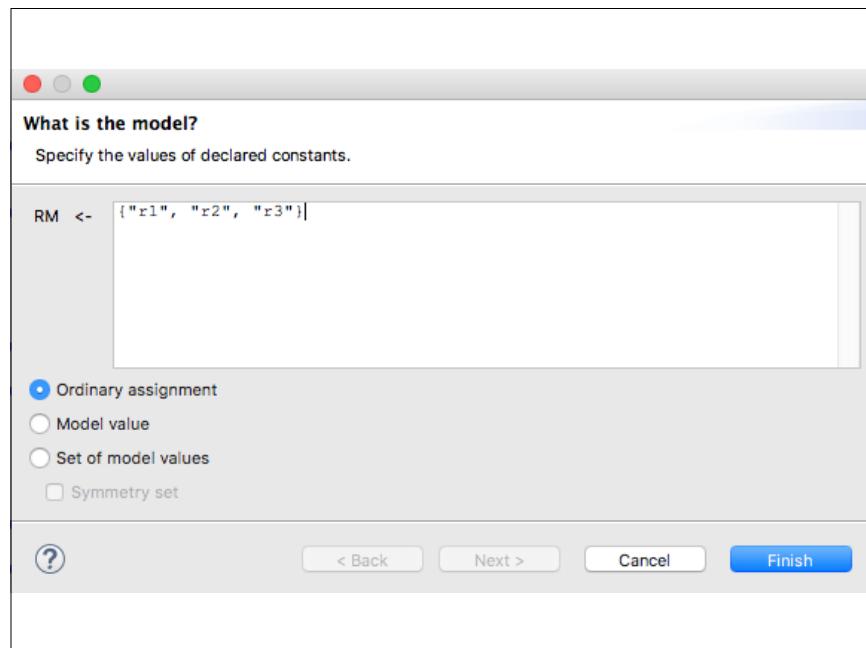
273



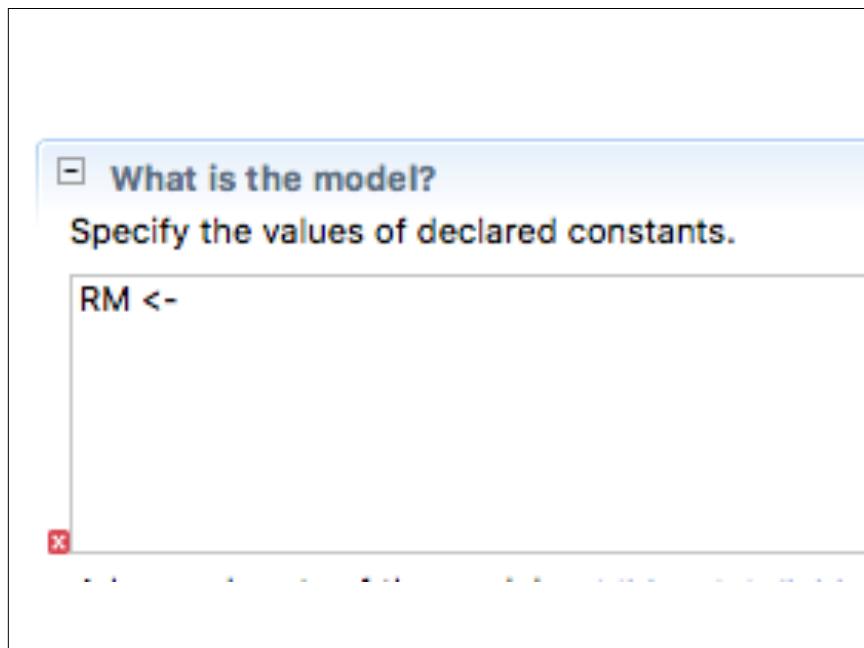
274



275



277



276

**X Model Overview** 2 errors detected

Checks the model for errors but does not run TLC on it.

Model description

What is the behavior spec?

Initial predicate and next-state relation

Init:

Next:

Temporal formula

No Behavior Spec

What is the model?

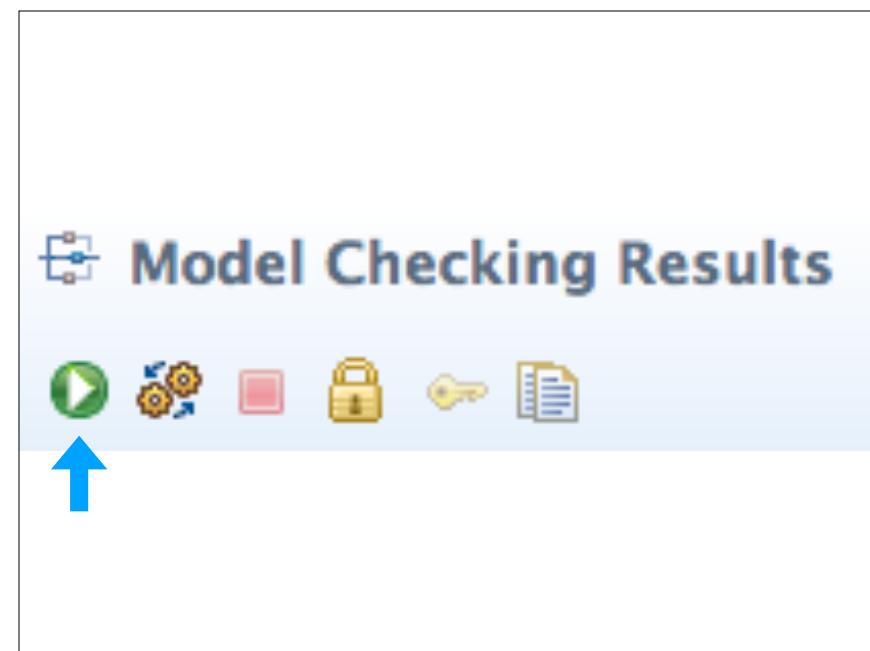
Specify the values of declared constants.

RM <- {"r1", "r2", "r3"}

Advanced parts of the model: [Additional di](#) [State constraints.](#) [Action const](#)

How to run?

278



279

[Model Overview](#) [Advanced Options](#) [Model Checking Results](#)

**Model Checking Results**

Stops the current TLC model checker run.

General

Start time:

End time:

Last checkpoint time:

Current status:

Errors detected:

Fingerprint collision probability:

Statistics

State space progress (click column header for graph)

| Time                | Diameter | States Found | Distinct States | Queue Size |
|---------------------|----------|--------------|-----------------|------------|
| 2017-09-19 12:08... | 11       | 1146         | 288             | 0          |

280

[Model Overview](#) [Advanced Options](#) [Model Checking Results](#)

**Model Checking Results**

General

Start time:

End time:

Last checkpoint time:

Current status:

Errors detected:

Fingerprint collision probability:

Statistics

State space progress (click column header for graph)

| Time                | Diameter | States Found | Distinct States | Queue Size | Coverage a                                                                                 |
|---------------------|----------|--------------|-----------------|------------|--------------------------------------------------------------------------------------------|
| 2017-09-19 12:08... | 11       | 1146         | 288             | 0          | Module<br>TwoPhase<br>TwoPhase<br>TwoPhase<br>TwoPhase<br>TwoPhase<br>TwoPhase<br>TwoPhase |

Evaluate Constant Expression

Expression:

281

## What to check?

### Deadlock

#### Invariants

Formulas true in every reachable state.

- TPTTypeOK
- TCConsistent

Add

Edit

Remove

#### Properties

282

## Model Checking Results



### General

Start time: Tue Sep 19 16:47:41 EDT 2017  
 End time: Tue Sep 19 16:47:41 EDT 2017  
 Last checkpoint time:  
 Current status: Not running  
 Errors detected: No errors  
 Fingerprint collision probability: calculated: 1.3E-14, observed: 2.6E

### Statistics

State space progress (click column header for graph)

| Time                | Diameter | States Found | Distinct States | Queue Size |
|---------------------|----------|--------------|-----------------|------------|
| 2017-09-19 16:47:41 | 11       | 1146         | 288             | 0          |

Coverage at 2017-09-19 16:47:41

| Module   | Location                           | Count |
|----------|------------------------------------|-------|
| TwoPhase | line 89, col 6 to line 89, col 22  | 1     |
| TwoPhase | line 90, col 6 to line 90, col 46  | 1     |
| TwoPhase | line 91, col 18 to line 91, col 24 | 1     |
| TwoPhase | line 91, col 27 to line 91, col 36 | 1     |
| TwoPhase | line 98, col 6 to line 98, col 22  | 64    |
| TwoPhase | line 99, col 6 to line 99, col 45  | 64    |

### Evaluate Constant Expression

283

## Evaluate Constant Expression

Expression:

```
[nodes: {1,2}, edges: {"a", "b", "c"}]
```

## Evaluate Constant Expression

Expression:

```
[nodes: {1,2}, edges: {"a", "b", "c"}]
```

Value:

```
{ [nodes :> 1, edges :> "a"],

 [nodes :> 1, edges :> "b"],

 [nodes :> 1, edges :> "c"],

 [nodes :> 2, edges :> "a"],

 [nodes :> 2, edges :> "b"],

 [nodes :> 2, edges :> "c"] }
```

284

285

# TwoPhase

```
CONSTANT RM /* The set of resource managers

VARIABLES
 rmState,
 tmState,
 tmPrepared,
 msgs
 /* rmState[r] is the state of resource manager r.
 * The state of the transaction manager.
 * The set of RMs from which the TM has received "Prepared"
 * messages.
```

286-1

# TwoPhase

```
CONSTANT RM /* The set of resource managers

VARIABLES
 rmState,
 tmState,
 tmPrepared,
 msgs
 /* rmState[r] is the state of resource manager r.
 * The state of the transaction manager.
 * The set of RMs from which the TM has received "Prepared"
 * messages.

 • RM and rmState are the same as before
```

286-2

# TwoPhase

```
CONSTANT RM /* The set of resource managers

VARIABLES
 rmState,
 tmState,
 tmPrepared,
 msgs
 /* rmState[r] is the state of resource manager r.
 * The state of the transaction manager.
 * The set of RMs from which the TM has received "Prepared"
 * messages.

 • RM and rmState are the same as before
```

- **tmState** is either “init” or “done”

286-3

# TwoPhase

```
CONSTANT RM /* The set of resource managers

VARIABLES
 rmState,
 tmState,
 tmPrepared,
 msgs
 /* rmState[r] is the state of resource manager r.
 * The state of the transaction manager.
 * The set of RMs from which the TM has received "Prepared"
 * messages.

 • RM and rmState are the same as before

 • tmState is either “init” or “done”

 • tmPrepared is either the empty set {}, or contains some
 combination of RMs (ex. {"rm1", "rm3"})
```

286-4

# TwoPhase

```
CONSTANT RM /* The set of resource managers
```

## VARIABLES

```
rmState,
tmState,
tmPrepared,
msgs
```

/\* rmState[r] is the state of resource manager r.  
 \* The state of the transaction manager.  
 \* The set of RMs from which the TM has received "Prepared"  
 \* messages.

- **RM** and **rmState** are the same as before
- **tmState** is either “init” or “done”
- **tmPrepared** is either the empty set {}, or contains some combination of RMs (ex. {"rm1", "rm3"})
- **msgs** starts as an empty set

286-5

# TwoPhase

```
Messages ==
(* *****)
(* The set of all possible messages. Messages of type "Prepared" are *)
(* sent from the RM indicated by the message's rm field to the TM. *)
(* Messages of type "Commit" and "Abort" are broadcast by the TM, to be *)
(* received by all RMs. The set msgs contains just a single copy of *)
(* such a message. *)
(* *****)
[type : {"Prepared"}, rm : RM] \union [type : {"Commit", "Abort"}]
```

287-1

# TwoPhase

```
Messages ==
(* *****)
(* The set of all possible messages. Messages of type "Prepared" are *)
(* sent from the RM indicated by the message's rm field to the TM. *)
(* Messages of type "Commit" and "Abort" are broadcast by the TM, to be *)
(* received by all RMs. The set msgs contains just a single copy of *)
(* such a message. *)
(* *****)
[type : {"Prepared"}, rm : RM] \union [type : {"Commit", "Abort"}]
```

287-2

# TwoPhase

```
Messages ==
(* *****)
(* The set of all possible messages. Messages of type "Prepared" are *)
(* sent from the RM indicated by the message's rm field to the TM. *)
(* Messages of type "Commit" and "Abort" are broadcast by the TM, to be *)
(* received by all RMs. The set msgs contains just a single copy of *)
(* such a message. *)
(* *****)
[type : {"Prepared"}, rm : RM] \union [type : {"Commit", "Abort"}]
```

287-3

# TwoPhase

```
Messages ==
(* *****)
(* The set of all possible messages. Messages of type "Prepared" are *)
(* sent from the RM indicated by the message's rm field to the TM. *)
(* Messages of type "Commit" and "Abort" are broadcast by the TM, to be *)
(* received by all RMs. The set msgs contains just a single copy of *)
(* such a message.
(* *****)
[type : {"Prepared"}, rm : RM] \union [type : {"Commit", "Abort"}]
```



287-4

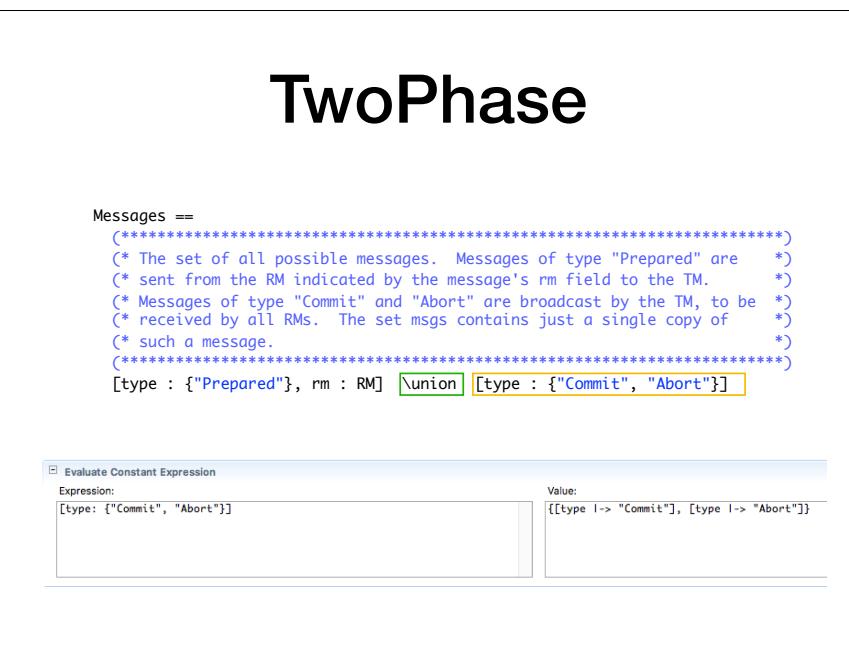
# TwoPhase

```
Messages ==
(* *****)
(* The set of all possible messages. Messages of type "Prepared" are *)
(* sent from the RM indicated by the message's rm field to the TM. *)
(* Messages of type "Commit" and "Abort" are broadcast by the TM, to be *)
(* received by all RMs. The set msgs contains just a single copy of *)
(* such a message.
(* *****)
[type : {"Prepared"}, rm : RM] \union [type : {"Commit", "Abort"}]
```

288-1

# TwoPhase

```
Messages ==
(* *****)
(* The set of all possible messages. Messages of type "Prepared" are *)
(* sent from the RM indicated by the message's rm field to the TM. *)
(* Messages of type "Commit" and "Abort" are broadcast by the TM, to be *)
(* received by all RMs. The set msgs contains just a single copy of *)
(* such a message.
(* *****)
[type : {"Prepared"}, rm : RM] \union [type : {"Commit", "Abort"}]
```



288-3

# TwoPhase

```
Messages ==
 (* *****)
 (* The set of all possible messages. Messages of type "Prepared" are *)
 (* sent from the RM indicated by the message's rm field to the TM. *)
 (* Messages of type "Commit" and "Abort" are broadcast by the TM, to be *)
 (* received by all RMs. The set msgs contains just a single copy of *)
 (* such a message. *)
 (* *****)
 [type : {"Prepared"}, rm : RM] \union [type : {"Commit", "Abort"}]
```

289-1

# TwoPhase

```
Messages ==
 (* *****)
 (* The set of all possible messages. Messages of type "Prepared" are *)
 (* sent from the RM indicated by the message's rm field to the TM. *)
 (* Messages of type "Commit" and "Abort" are broadcast by the TM, to be *)
 (* received by all RMs. The set msgs contains just a single copy of *)
 (* such a message. *)
 (* *****)
 [type : {"Prepared"}, rm : RM] \union [type : {"Commit", "Abort"}]
```

289-2

# TwoPhase

```
Messages ==
 (* *****)
 (* The set of all possible messages. Messages of type "Prepared" are *)
 (* sent from the RM indicated by the message's rm field to the TM. *)
 (* Messages of type "Commit" and "Abort" are broadcast by the TM, to be *)
 (* received by all RMs. The set msgs contains just a single copy of *)
 (* such a message. *)
 (* *****)
 [type : {"Prepared"}, rm : RM] \union [type : {"Commit", "Abort"}]
```

Evaluate Constant Expression

Expression:  
[type: {"Prepared"}, rm:RM ] \union [type: {"Commit", "Abort"}]

Value:  
[ { type : "Commit",  
 rm : "r1" },  
{ type : "Abort",  
 rm : "r1" },  
{ type : "Prepared",  
 rm : "r1" },  
{ type : "Prepared",  
 rm : "r2" },  
{ type : "Prepared",  
 rm : "r3" } ]

289-3

# TPTypeOK

```
TPTypeOK ==
 (* *****)
 (* The type-correctness invariant *)
 (* *****)
 /\ rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
 /\ tmState \in {"init", "done"}
 /\ tmPrepared \subseteqq RM
 /\ msgs \subseteqq Messages
```

290-1

# TPTTypeOK

```
TPTTypeOK ==
 (* *****)
 (* The type-correctness invariant *)
 (* *****)
 \wedge rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
 \wedge tmState \in {"init", "done"}
 \wedge tmPrepared \subseteqq RM
 \wedge msgs \subseteqq Messages
```

290-2

# TPTTypeOK

```
TPTTypeOK ==
 (* *****)
 (* The type-correctness invariant *)
 (* *****)
 \wedge rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
 \wedge tmState \in {"init", "done"}
 \wedge tmPrepared \subseteqq RM
 \wedge msgs \subseteqq Messages
```

Evaluate Constant Expression

Expression:

```
[RM -> {"working", "prepared", "committed", "aborted"}]
```

290-3

# TPTTypeOK

```
TPTTypeOK ==
 (* *****)
 (* The type-correctness invariant *)
 (* *****)
 \wedge rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
 \wedge tmState \in {"init", "done"}
 \wedge tmPrepared \subseteqq RM
 \wedge msgs \subseteqq Messages

 { [r1 |-> "working", r2 |-> "working", r3 |-> "working"],
 [r1 |-> "working", r2 |-> "working", r3 |-> "prepared"],
 [r1 |-> "working", r2 |-> "working", r3 |-> "committed"],
 [r1 |-> "working", r2 |-> "working", r3 |-> "aborted"],
 [r1 |-> "working", r2 |-> "prepared", r3 |-> "working"],
 [r1 |-> "working", r2 |-> "prepared", r3 |-> "prepared"],
 [r1 |-> "working", r2 |-> "prepared", r3 |-> "committed"],
 [r1 |-> "working", r2 |-> "prepared", r3 |-> "aborted"],
 [r1 |-> "working", r2 |-> "committed", r3 |-> "working"],
 [r1 |-> "working", r2 |-> "committed", r3 |-> "prepared"],
 [r1 |-> "working", r2 |-> "committed", r3 |-> "committed"],
 [r1 |-> "working", r2 |-> "committed", r3 |-> "aborted"],
 [r1 |-> "aborted", r2 |-> "prepared", r3 |-> "aborted"],
 [r1 |-> "aborted", r2 |-> "committed", r3 |-> "working"],
 [r1 |-> "aborted", r2 |-> "committed", r3 |-> "aborted"] }
```

290-4

# TPTTypeOK

```
TPTTypeOK ==
 (* *****)
 (* The type-correctness invariant *)
 (* *****)
 \wedge rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
 \wedge tmState \in {"init", "done"}
 \wedge tmPrepared \subseteqq RM
 \wedge msgs \subseteqq Messages
```

291-1

# TPTypeOK

```
TPTypeOK ==
 (* The type-correctness invariant *)
 (^ rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
 ^ tmState \in {"init", "done"}
 ^ tmPrepared \subseteqq RM
 ^ msgs \subseteqq Messages
```

291-2

# TPTypeOK

```
TPTypeOK ==
 (* The type-correctness invariant *)
 (^ rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
 ^ tmState \in {"init", "done"}
 ^ tmPrepared \subseteqq RM
 ^ msgs \subseteqq Messages
```

\subseteqq means “subset or equal”

291-3

# TPTypeOK

```
TPTypeOK ==
 (* The type-correctness invariant *)
 (^ rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
 ^ tmState \in {"init", "done"}
 ^ tmPrepared \subseteqq RM
 ^ msgs \subseteqq Messages
```

\subseteqq means “subset or equal”

291-4

# msgs

292-1

# msgs

- The way we will model message transfer is by keeping a global set of all messages ever sent, **msgs**

292-2

# msgs

- The way we will model message transfer is by keeping a global set of all messages ever sent, **msgs**
- Messages will be added to this set, but never removed

292-3

# msgs

- The way we will model message transfer is by keeping a global set of all messages ever sent, **msgs**
- Messages will be added to this set, but never removed
- The protocol behaves correctly in the face of messages never being removed

292-4

# msgs

- The way we will model message transfer is by keeping a global set of all messages ever sent, **msgs**
- Messages will be added to this set, but never removed
- The protocol behaves correctly in the face of messages never being removed
- This will make more sense after reading through the spec!

292-5

# TPIinit

```
TPIinit ==
 (*****
 (* The initial predicate. *)
 (*****)
 \wedge rmState = [r \in RM |-> "working"]
 \wedge tmState = "init"
 \wedge tmPrepared = {}
 \wedge msgs = {}
)
```

293-1

# TPIinit

```
TPIinit ==
 (*****
 (* The initial predicate. *)
 (*****)
 \wedge rmState = [r \in RM |-> "working"]
 \wedge tmState = "init"
 \wedge tmPrepared = {}
 \wedge msgs = {}
)
```

293-2

# TPIinit

```
TPIinit ==
 (*****
 (* The initial predicate. *)
 (*****)
 \wedge rmState = [r \in RM |-> "working"]
 \wedge tmState = "init"
 \wedge tmPrepared = {}
 \wedge msgs = {}
)
```

Evaluate this expression. What do you get?

293-3

# TPIinit

```
TPIinit ==
 (*****
 (* The initial predicate. *)
 (*****)
 \wedge rmState = [r \in RM |-> "working"]
 \wedge tmState = "init"
 \wedge tmPrepared = {}
 \wedge msgs = {}
)
```

Evaluate this expression. What do you get?

Value:  
[r1 |-> "working", r2 |-> "working", r3 |-> "working"]

293-4

## TPInit

```
TPInit ==
(******)
(* The initial predicate. *)
(******)
\wedge rmState = [r \in RM |> "working"]
\wedge tmState = "init"
\wedge tmPrepared = {}
\wedge msgs = {}
```

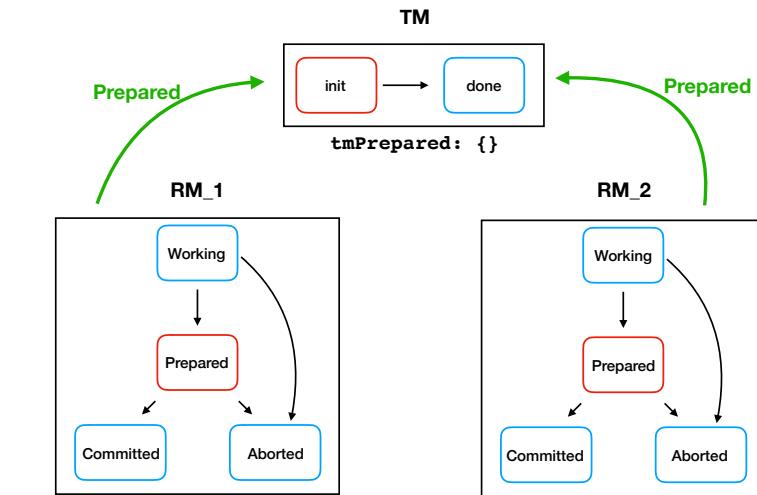
Evaluate this expression. What do you get?

Value:  
[r1 |> "working", r2 |> "working", r3 |> "working"]

This is a single function, where the DOMAIN is made up of resource managers, and the value associated to each is “working”

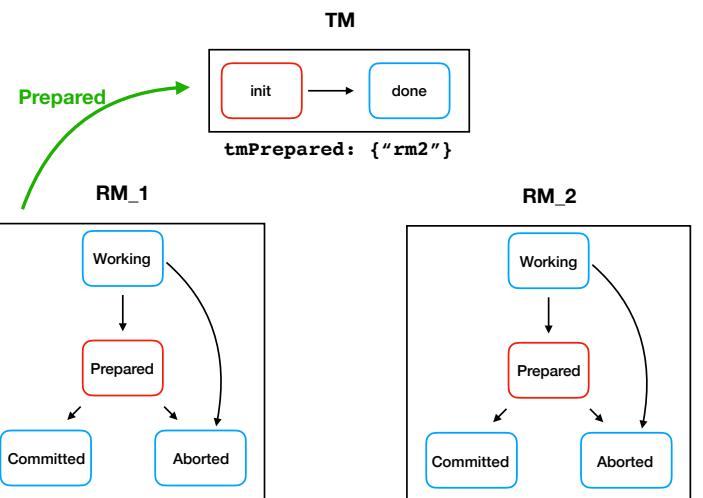
293-5

## TMRcvPrepared(r)



294

## TMRcvPrepared(r)



295

## TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
\wedge tmState = "init"
\wedge [type |> "Prepared", rm |> r] \in msgs
\wedge tmPrepared' = tmPrepared \union {r}
\wedge UNCHANGED <<rmState, tmState, msgs>>
```

This formula describes the TM receiving a “Prepared” message from a RM

296-1

# TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \union {r}
 \wedge UNCHANGED <<rmState, tmState, msgs>>
```

This formula describes the TM receiving a “Prepared” message from a RM

**This formula has primed and unprimed variables, and is therefore an “action”**

296-2

# TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \union {r}
 \wedge UNCHANGED <<rmState, tmState, msgs>>
```

This formula describes the TM receiving a “Prepared” message from a RM

**This formula has primed and unprimed variables, and is therefore an “action”**

**What are the enabling conditions?**

296-4

# TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \union {r}
 \wedge UNCHANGED <<rmState, tmState, msgs>>
```

This formula describes the TM receiving a “Prepared” message from a RM

**This formula has primed and unprimed variables, and is therefore an “action”**

**What are the enabling conditions?**

296-3

# TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \union {r}
 \wedge UNCHANGED <<rmState, tmState, msgs>>
```

This formula describes the TM receiving a “Prepared” message from a RM

**This formula has primed and unprimed variables, and is therefore an “action”**

**What are the enabling conditions?**

tmState = “init” means that the TM must be in the “init” state

296-5

# TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \union {r}
 \wedge UNCHANGED <<rmState, tmState, msgs>>
```

This formula describes the TM receiving a “Prepared” message from a RM

**This formula has primed and unprimed variables, and is therefore an “action”**

**What are the enabling conditions?**

tmState = “init” means that the TM must be in the “init” state

[type |-> “Prepared”, rm |-> r] \in msgs means that a “Prepared” message from RM r must be present in the msgs set

296-6

# TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \union {r}
 \wedge UNCHANGED <<rmState, tmState, msgs>>
```

297-1

# TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \union {r}
 \wedge UNCHANGED <<rmState, tmState, msgs>>
```

# TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \union {r}
 \wedge UNCHANGED <<rmState, tmState, msgs>>
```

This says that “in the next state, **tmPrepared** will be the result of taking the UNION of the current **tmPrepared** and r”

ex. If r is “rm3”, and tmPrepared = {"rm1"}, then in the next state, tmPrepared will be {"rm1", “rm3”}

297-2

297-3

## TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \cup {r}
 \wedge UNCHANGED <<rmState, tmState, msgs>>
```

298-1

## TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \cup {r}
 \wedge UNCHANGED <<rmState, tmState, msgs>>
```

298-2

## TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \cup {r}
 \wedge UNCHANGED <<rmState, tmState, msgs>>
```

This is new

298-3

## TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \cup {r}
 \wedge UNCHANGED <<rmState, tmState, msgs>>
```

This is new

It says: "In the next state, **rmState** and **tmState** and **msgs** will be unchanged"

298-4

# TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \cup {r}
 \wedge UNCHANGED <>rmState, tmState, msgs>>
```

This is new

It says: "In the next state, **rmState** and **tmState** and **msgs** will be unchanged"

It is syntactic sugar for:

```
\wedge rmState' = rmState
\wedge tmState' = tmState
\wedge msgs' = msgs
```

298-5

# TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \cup {r}
 \wedge UNCHANGED <>rmState, tmState, msgs>>
```

299-1

# TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \cup {r}
 \wedge UNCHANGED <>rmState, tmState, msgs>>
```

- Remember, a formula describes the valid pairs of states, i.e., the valid next state given the current state

299-2

# TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \cup {r}
 \wedge UNCHANGED <>rmState, tmState, msgs>>
```

- Remember, a formula describes the valid pairs of states, i.e., the valid next state given the current state
- A formula **must** explicitly mention all the variables in the next state

299-3

# TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \cup {r}
 \wedge UNCHANGED <<rmState, tmState, msgs>>
```

- Remember, a formula describes the valid pairs of states, i.e., the valid next state given the current state
- A formula **must** explicitly mention all the variables in the next state
- Without `UNCHANGED <<rmState, tmState, msgs>>` then a valid **next state** would be:

299-4

# TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \cup {r}
 \wedge UNCHANGED <<rmState, tmState, msgs>>
```

- Remember, a formula describes the valid pairs of states, i.e., the valid next state given the current state
- A formula **must** explicitly mention all the variables in the next state
- Without `UNCHANGED <<rmState, tmState, msgs>>` then a valid **next state** would be:

```
rmState: [rm1 |-> "working", rm2 |-> "prepared", rm3 |-> "working"]
msgs: {[type |-> "Prepared", rm |-> "rm2"]}
tmPrepared: {"rm2"}
tmState: "what is going on here"
```

299-6

# TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \cup {r}
 \wedge UNCHANGED <<rmState, tmState, msgs>>
```

- Remember, a formula describes the valid pairs of states, i.e., the valid next state given the current state
- A formula **must** explicitly mention all the variables in the next state
- Without `UNCHANGED <<rmState, tmState, msgs>>` then a valid **next state** would be:

```
rmState: [rm1 |-> "working", rm2 |-> "prepared", rm3 |-> "working"]
msgs: {[type |-> "Prepared", rm |-> "rm2"]}
tmPrepared: {"rm2"}
tmState: "what is going on here"
```

299-5

# TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \cup {r}
 \wedge UNCHANGED <<rmState, tmState, msgs>>
```

```
rmState: [rm1 |-> "working", rm2 |-> "prepared", rm3 |-> "working"]
msgs: {[type |-> "Prepared", rm |-> "rm2"]}
tmPrepared: {"rm2"}
tmState: "what is going on here"
```

300-1

## TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \cup {r}
 \wedge UNCHANGED <> rmState, tmState, msgs>>
```

```
rmState: [rm1 |-> "working", rm2 |-> "prepared", rm3 |-> "working"]
msgs: {[type |-> "Prepared", rm |-> "rm2"]}
tmPrepared: {"rm2"}
tmState: "what is going on here"
```

- Why would that be a valid next state if we didn't have the **UNCHANGED**?

300-2

## TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \cup {r}
 \wedge UNCHANGED <> rmState, tmState, msgs>>
```

```
rmState: [rm1 |-> "working", rm2 |-> "prepared", rm3 |-> "working"]
msgs: {[type |-> "Prepared", rm |-> "rm2"]}
tmPrepared: {"rm2"}
tmState: "what is going on here"
```

- Why would that be a valid next state if we didn't have the **UNCHANGED**?
- Because the possible set of next states is infinite, and the purpose of our formula is to identify the “valid” ones

300-3

## RMPrepare(r)

```
RMPrepare(r) ==
 \wedge rmState[r] = "working"
 \wedge rmState' = [rmState EXCEPT ![r] = "prepared"]
 \wedge msgs' = msgs \cup {[type |-> "Prepared", rm |-> r]}
 \wedge UNCHANGED <> rmState, tmPrepared>>
```

301-1

## RMPrepare(r)

```
RMPrepare(r) ==
 \wedge rmState[r] = "working"
 \wedge rmState' = [rmState EXCEPT ![r] = "prepared"]
 \wedge msgs' = msgs \cup {[type |-> "Prepared", rm |-> r]}
 \wedge UNCHANGED <> rmState, tmPrepared>>
```

- This action defines a RM r transitioning from “working” to “prepared”

301-2

# RMPrepare(r)

```
RMPrepare(r) ==
 \rmState[r] = "working"
 \rmState' = [rmState EXCEPT ![r] = "prepared"]
 \msgs' = msgs \union {[type I-> "Prepared", rm I-> r]}
 \UNCHANGED <<tmState, tmPrepared>>
```

- This action defines a RM r transitioning from “working” to “prepared”
- It is valid for an RM r whenever that r is in a “working” state

301-3

# RMPrepare(r)

```
RMPrepare(r) ==
 \rmState[r] = "working"
 \rmState' = [rmState EXCEPT ![r] = "prepared"]
 \msgs' = msgs \union {[type I-> "Prepared", rm I-> r]}
 \UNCHANGED <<tmState, tmPrepared>>
```

- This action defines a RM r transitioning from “working” to “prepared”
- It is valid for an RM r whenever that r is in a “working” state
- In the new state:
  - rmState has that RM set to “prepared”

301-5

# RMPrepare(r)

```
RMPrepare(r) ==
 \rmState[r] = "working"
 \rmState' = [rmState EXCEPT ![r] = "prepared"]
 \msgs' = msgs \union {[type I-> "Prepared", rm I-> r]}
 \UNCHANGED <<tmState, tmPrepared>>
```

- This action defines a RM r transitioning from “working” to “prepared”
- It is valid for an RM r whenever that r is in a “working” state
- In the new state:

301-4

# RMPrepare(r)

```
RMPrepare(r) ==
 \rmState[r] = "working"
 \rmState' = [rmState EXCEPT ![r] = "prepared"]
 \msgs' = msgs \union {[type I-> "Prepared", rm I-> r]}
 \UNCHANGED <<tmState, tmPrepared>>
```

- This action defines a RM r transitioning from “working” to “prepared”
- It is valid for an RM r whenever that r is in a “working” state
- In the new state:
  - rmState has that RM set to “prepared”
  - And a new message has been added to msgs, specifically, the message from RM r to the TM, telling it that it is prepared

301-6

# TPNext

```
TPNext ==
 \vee TMCommit \vee TMAbort
 \vee \E r \in RM :
 TMRcvPrepared(r) \vee RMPREPARE(r) \vee RMChooseToAbort(r)
 \vee RMRCVCOMMITMSG(r) \vee RMRCVABORTMSG(r)
```

302-1

# TPNext

```
TPNext ==
 \vee TMCommit \vee TMAbort
 \vee \E r \in RM :
 TMRcvPrepared(r) \vee RMPREPARE(r) \vee RMChooseToAbort(r)
 \vee RMRCVCOMMITMSG(r) \vee RMRCVABORTMSG(r)
```

- Our next state action. This determines all the possible next states

302-2

# TPNext

```
TPNext ==
 \vee TMCommit \vee TMAbort
 \vee \E r \in RM :
 TMRcvPrepared(r) \vee RMPREPARE(r) \vee RMChooseToAbort(r)
 \vee RMRCVCOMMITMSG(r) \vee RMRCVABORTMSG(r)
```

- Our next state action. This determines all the possible next states

**At any time, the TM can potentially commit or abort**

302-3

# TPNext

```
TPNext ==
 \vee TMCommit \vee TMAbort
 \vee \E r \in RM :
 TMRcvPrepared(r) \vee RMPREPARE(r) \vee RMChooseToAbort(r)
 \vee RMRCVCOMMITMSG(r) \vee RMRCVABORTMSG(r)
```

- Our next state action. This determines all the possible next states

**At any time, the TM can potentially commit or abort**

**For any of the RMs, a valid next state is potentially**

302-4

# TPNext

```
TPNext ==
 \vee TMCommit \vee TMAbort
 \vee \E r \in RM :
 TMRcvPrepared(r) \vee RMPrepare(r) \vee RMChooseToAbort(r)
 \vee RMRcvCommitMsg(r) \vee RMRcvAbortMsg(r)
```

- Our next state action. This determines all the possible next states
  - At any time, the TM can *potentially* commit or abort
  - For any of the RMs, a valid next state is *potentially*
    - The TM receiving the “prepared” message from the RM  $r$

302-5

# TPNext

```
TPNext ==
 \vee TMCommit \vee TMAbort
 \vee \E r \in RM :
 TMRcvPrepared(r) \vee RMPrepare(r) \vee RMChooseToAbort(r)
 \vee RMRcvCommitMsg(r) \vee RMRcvAbortMsg(r)
```

- Our next state action. This determines all the possible next states
  - At any time, the TM can *potentially* commit or abort
  - For any of the RMs, a valid next state is *potentially*
    - The TM receiving the “prepared” message from the RM  $r$
    - The RM moving into a “prepared” state

302-6

# TPNext

```
TPNext ==
 \vee TMCommit \vee TMAbort
 \vee \E r \in RM :
 TMRcvPrepared(r) \vee RMPrepare(r) \vee RMChooseToAbort(r)
 \vee RMRcvCommitMsg(r) \vee RMRcvAbortMsg(r)
```

- Our next state action. This determines all the possible next states
  - At any time, the TM can *potentially* commit or abort
  - For any of the RMs, a valid next state is *potentially*
    - The TM receiving the “prepared” message from the RM  $r$
    - The RM moving into a “prepared” state
    - The RM choosing to abort

302-7

# TPNext

```
TPNext ==
 \vee TMCommit \vee TMAbort
 \vee \E r \in RM :
 TMRcvPrepared(r) \vee RMPrepare(r) \vee RMChooseToAbort(r)
 \vee RMRcvCommitMsg(r) \vee RMRcvAbortMsg(r)
```

- Our next state action. This determines all the possible next states
  - At any time, the TM can *potentially* commit or abort
  - For any of the RMs, a valid next state is *potentially*
    - The TM receiving the “prepared” message from the RM  $r$
    - The RM moving into a “prepared” state
    - The RM choosing to abort
    - The RM receiving a commit message

302-8

## TPNext

```
TPNext ==
 \vee TMCommit \vee TMAbort
 \vee \E r \in RM :
 TMRcvPrepared(r) \vee RMPREPARE(r) \vee RMChooseToAbort(r)
 \vee RMRCVCOMMITMSG(r) \vee RMRCVABORTMSG(r)
```

- Our next state action. This determines all the possible next states

- At any time, the TM can *potentially* commit or abort
- For any of the RMs, a valid next state is *potentially*
  - The TM receiving the “prepared” message from the RM  $r$
  - The RM moving into a “prepared” state
  - The RM choosing to abort
  - The RM receiving a commit message
  - The RM receiving an abort message

302-9

## INSTANCE TCommit

### INSTANCE TCommit

- This “imports” all variable declarations, constants, and definitions from the **TCommit** module

### INSTANCE TCommit

- This “imports” all variable declarations, constants, and definitions from the **TCommit** module
- This is how we got access to the **TCConsistent** invariant

303-2

303-3

## INSTANCE TCommit

- This “imports” all variable declarations, constants, and definitions from the **TCommit** module
- This is how we got access to the **TCConsistent** invariant
- Because the model checker still passes with the **TCConsistent** invariant enabled, it means that the **TwoPhase** specification “implements” the **TCommit** specification

303-4

Model Checking Results

General

|                                    |                                     |
|------------------------------------|-------------------------------------|
| Start time:                        | Tue Sep 19 16:47:41 EDT 2017        |
| End time:                          | Tue Sep 19 16:47:41 EDT 2017        |
| Last checkpoint time:              |                                     |
| Current status:                    | Not running                         |
| Errors detected:                   | No errors.                          |
| Fingerprint collision probability: | calculated: 1.3E-14, observed: 2.6E |

Statistics

State space progress (click column header for graph)

| Time                | Diameter | States Found | Distinct States | Queue Size |
|---------------------|----------|--------------|-----------------|------------|
| 2017-09-19 16:47... | 11       | 1146         | 288             | 0          |

Coverage at 2017-09-19 16:47:41

| Module   | Location                           | Count |
|----------|------------------------------------|-------|
| TwoPhase | line 41, col 3 to line 80, col 39  | 50    |
| TwoPhase | line 89, col 6 to line 89, col 22  | 1     |
| TwoPhase | line 90, col 6 to line 90, col 46  | 1     |
| TwoPhase | line 91, col 18 to line 91, col 24 | 1     |
| TwoPhase | line 91, col 27 to line 91, col 36 | 1     |
| TwoPhase | line 98, col 6 to line 98, col 22  | 64    |
| TwoPhase | line 99, col 6 to line 99, col 45  | 64    |

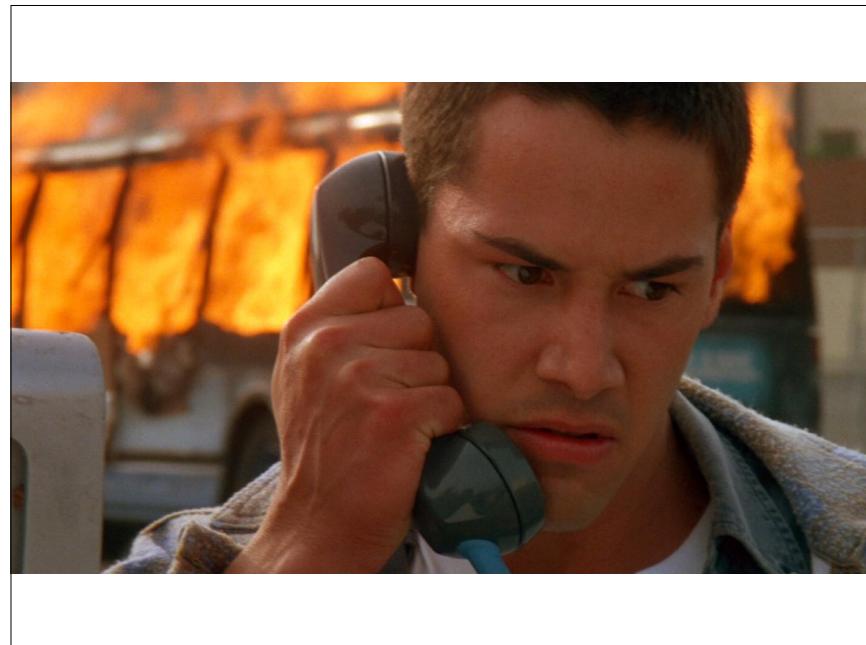
Evaluate Constant Expression

305

## Exercise

- Replace the contents of your `TwoPhase.tla` with the contents of `TwoPhase_incomplete.tla`
- Try to write the correct definitions for
  - TMCommit
  - TMAbort
  - RMChooseToAbort
  - RMRcvCommitMsg
  - RMRcvAbortMsg

304



306

# The End?

- This brings us to the end of Video 6 in Lamport's series
  - TLA Video Series: [https://rax.io/tla\\_videos](https://rax.io/tla_videos)
  - <http://lamport.azurewebsites.net/video/videos.html>
- You now know **most** of the important TLA+ syntax, but not all
  - LET/IN is an important one, so is CHOOSE
- I recommend watching Lamport's videos, as they'll offer a slightly different perspective and will solidify your understanding
- Thanks!