

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

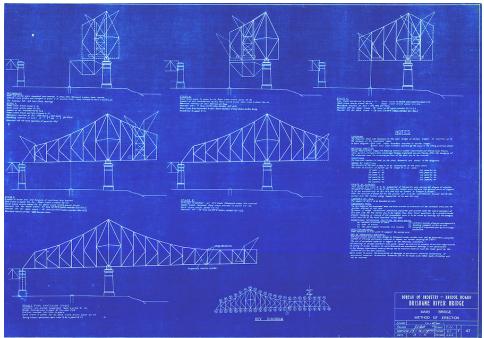
- Introduce the concepts of TLA+
- Introduce a good amount of the syntax of TLA+
- Introduce the model checker, TLC
- Provide a foundation on which you can continue learning
- Convince everyone that thinking **early** is a good thing

6-6

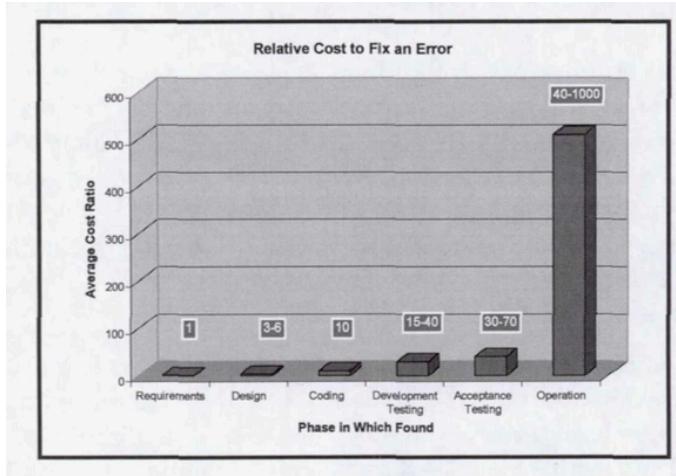
Goals

- 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

"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

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/>

11

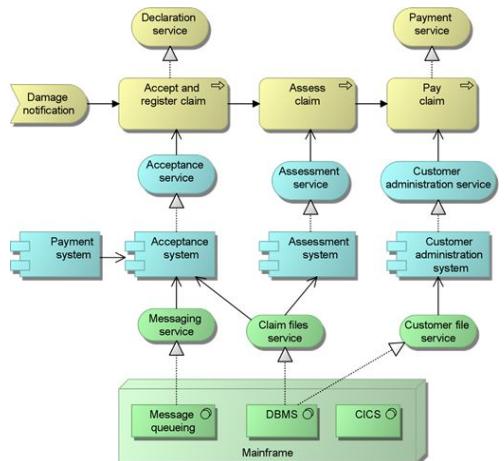
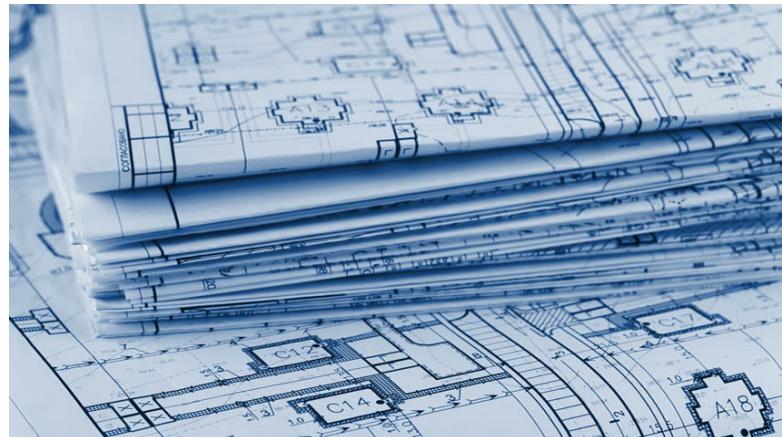


Image via <https://en.wikipedia.org/wiki/ArchiMate>

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)

17-2

"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)

17-3

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

Raft

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

19-2

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

20-1

Raft

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

20-2

Raft

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

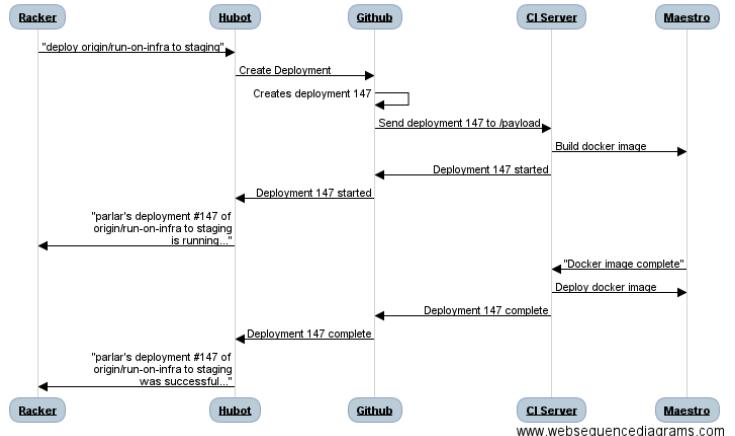
20-3

Raft

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

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 \in {"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 \in 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)

```

22

Deployment Pipeline

Deployment Pipeline

- “Only deploy if an image exists”

23-1

Deployment Pipeline

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

23-2

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

```
def has_access(self, *allowed_dept_list, **kwargs):  
    ignore_core_admin = kwargs.get("ignore_core_admin", False)  
    special_permissions = set(("HK_RACKER", "CORE_PERMISSION_ADMIN", "PER_EMPLOYEE_TERMINATE"))  
    users_departments = set(self.departments)  
    allowed_dept_list = set(allowed_dept_list)  
  
    if special_permissions.union(allowed_dept_list):  
        return True if allowed_dept_list.union(users_departments) else False  
  
    elif ignore_core_admin is False and "CORE_ADMIN" in users_departments:  
        return True  
  
    else:  
        return True if allowed_dept_list.union(users_departments) else False
```

24

```
def has_access(self, *allowed_dept_list, **kwargs):
    ignore_core_admin = kwargs.get("ignore_core_admin", False)
    special_permissions = set(("HK_RACKER", "CORE_PERMISSION_ADMIN", "PER_EMPLOYEE_TERMINATE"))
    users_departments = set(self.departments)
    allowed_dept_list = set(allowed_dept_list)

    ignore_core_admin = True if special_permissions.union(allowed_dept_list) else ignore_core_admin
    if not ignore_core_admin and "CORE_ADMIN" in users_departments:
        return True
    else:
        return allowed_dept_list.union(users_departments)
```

25

PlusCal

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

26-2

PlusCal

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

26-3

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+

26-4

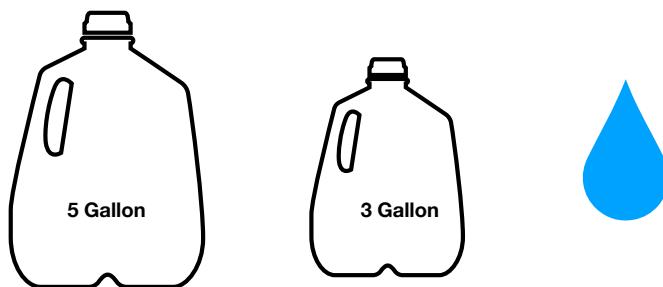
PlusCal

- 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

26-5

How do we measure out exactly 4 gallons?

i.e. "The Die Hard Problem"



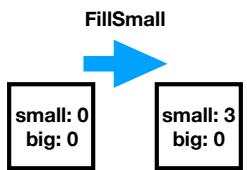
27

One Possible Execution

small: 0
big: 0

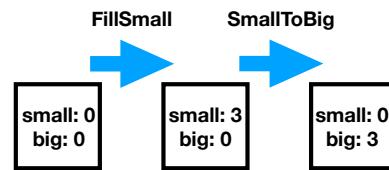
28-1

One Possible Execution



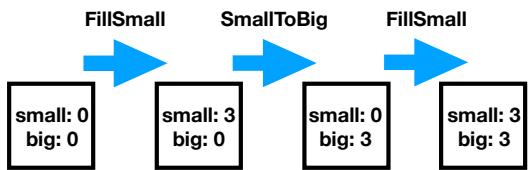
28-2

One Possible Execution



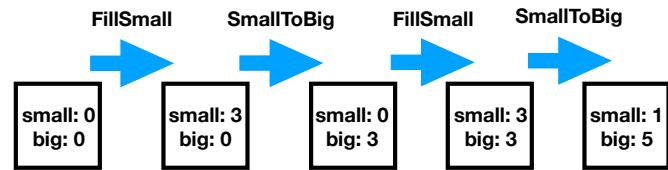
28-3

One Possible Execution



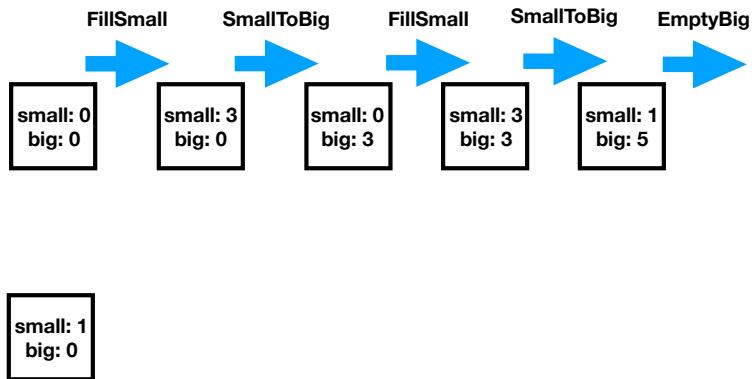
28-4

One Possible Execution



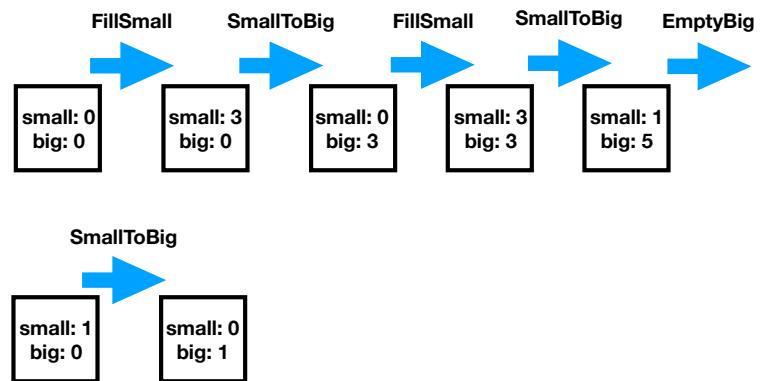
28-5

One Possible Execution



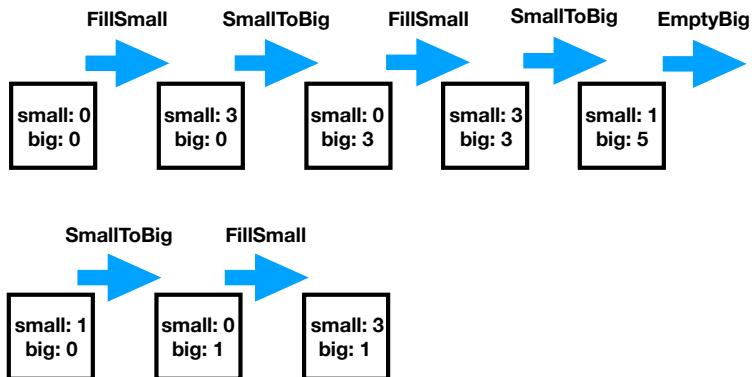
28-6

One Possible Execution



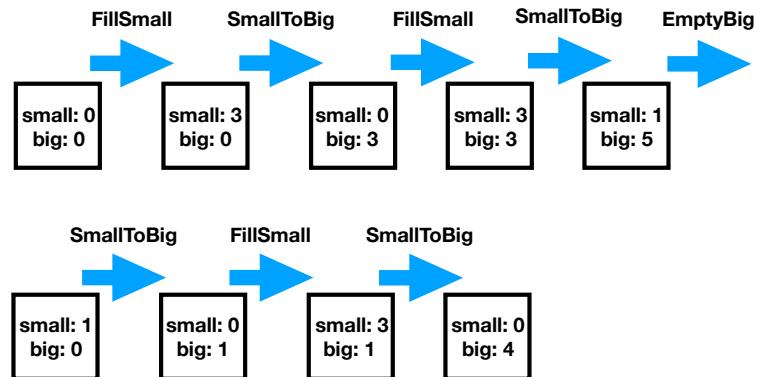
28-7

One Possible Execution



28-8

One Possible Execution



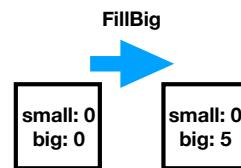
28-9

Another Possible Execution



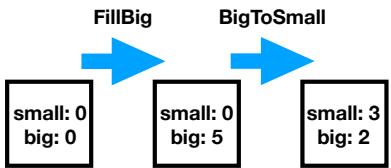
29-1

Another Possible Execution



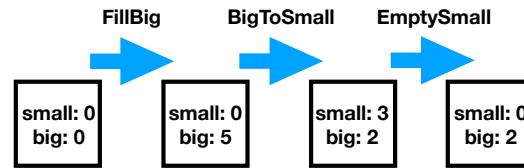
29-2

Another Possible Execution



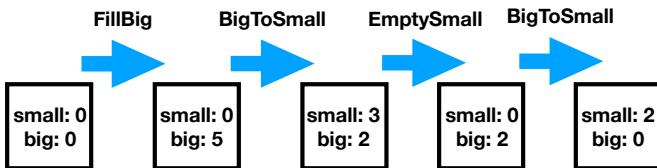
29-3

Another Possible Execution



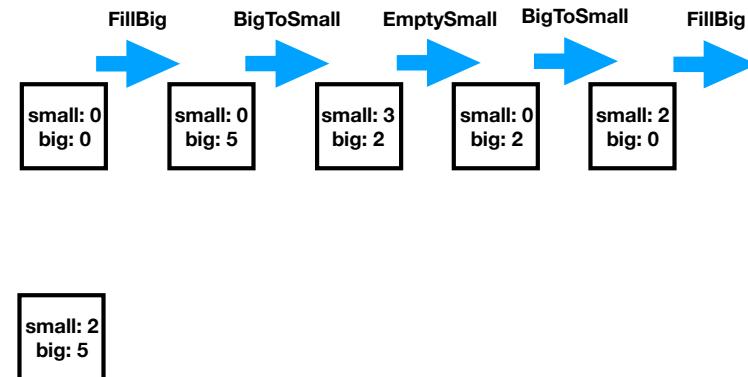
29-4

Another Possible Execution



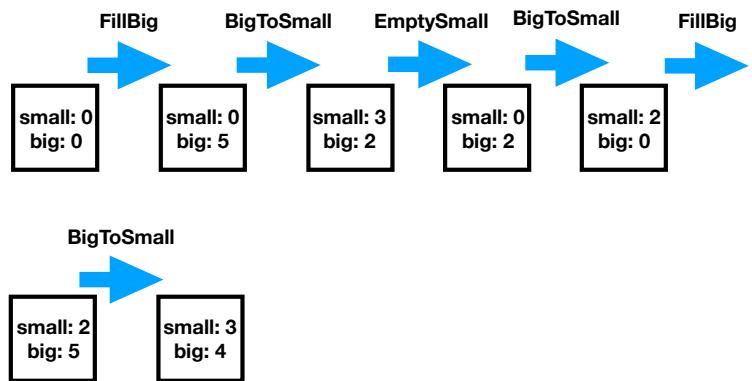
29-5

Another Possible Execution



29-6

Another Possible Execution



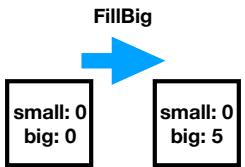
29-7

Yet Another Possible Execution



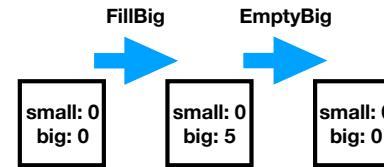
30-1

Yet Another Possible Execution



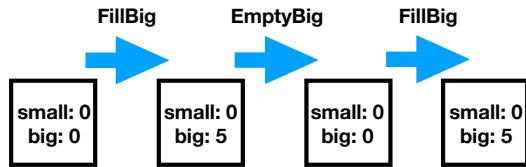
30-2

Yet Another Possible Execution



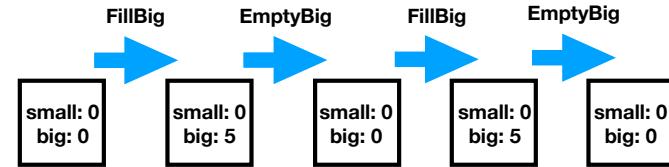
30-3

Yet Another Possible Execution



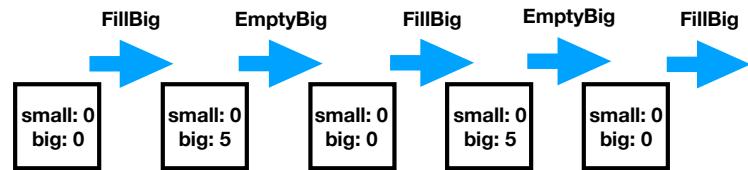
30-4

Yet Another Possible Execution

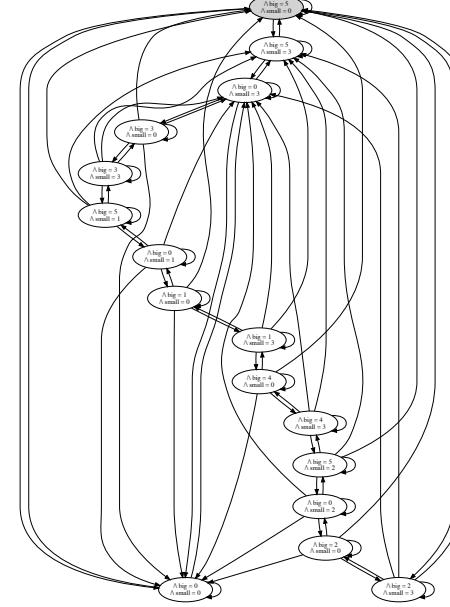


30-5

Yet Another Possible Execution



30-6



31

TLA+ Terms

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

32-1

32-2

TLA+ Terms

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

32-3

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.

32-4

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**.

32-5

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.

32-6

How could we describe digital systems (behaviours)?

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

33

We'll instead use State Machines!

A state machine is defined by three things:

We'll instead use State Machines!

A state machine is defined by three things:

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

34-2

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**

34-3

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**

34-4

Die Hard State Machine

Die Hard State Machine

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

35-2

Die Hard State Machine

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

35-3

Die Hard State Machine

1. Variables: **big** and **small**
2. Possible initial values: **big=0, small=0**
3. Possible next states: ?

35-4

Die Hard Next States

Die Hard Next States

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

36-2

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**

36-3

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**

36-4

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**

36-5

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:

36-6

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:
$$\text{if } (\text{big} + \text{small}) \leq 5 \text{ THEN } \text{big}=\text{big}+\text{small}; \text{small}=0$$

36-7

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)
```

36-8

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)
```

36-9

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.

36-10

Possible next states

small: 3
big: 2

37-1

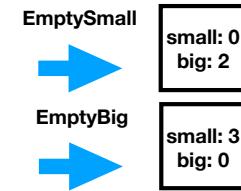
Possible next states



small: 3
big: 2

37-2

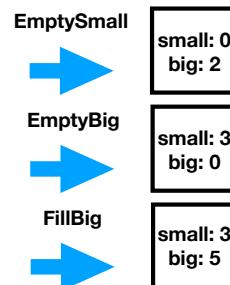
Possible next states



small: 3
big: 2

37-3

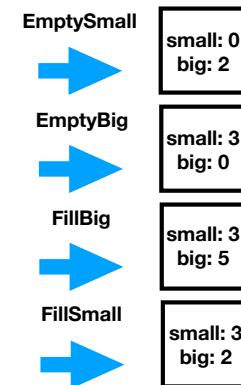
Possible next states



small: 3
big: 2

37-4

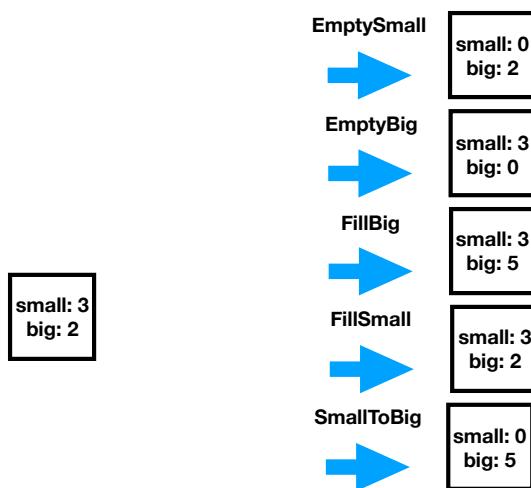
Possible next states



small: 3
big: 2

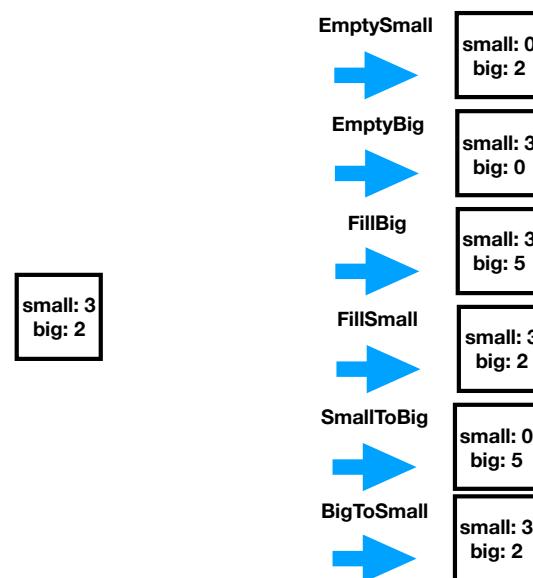
37-5

Possible next states



37-6

Possible next states



37-7

```

VARIABLES small, big

TypeOK ==  $\wedge$  small  $\in$  0..3
           $\wedge$  big  $\in$  0..5

Init ==  $\wedge$  big = 5
         $\wedge$  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
  
```

38

Why use state machines?

39-1

Why use state machines?

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

39-2

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:

39-3

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

39-4

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

39-5

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

39-6

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

39-7

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.

39-8

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.

39-9

Math Time!!!

40

“and” / “or”

41-1

“and” / “or”



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
```

“and” / “or”



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
```

41-2

41-3

= 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 create named definitions of things.

= 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 ==.

42-1

42-2

= 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 create named definitions of things.

= 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 create named definitions of things.

`MyDef == A /\ 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.

42-3

42-4

Exercise

TRUE \/ (TRUE /\ FALSE)

TRUE /\ (FALSE /\ (TRUE \/ FALSE))

FALSE /\ (TRUE /\ ((FALSE \/ TRUE) /\ (TRUE \/ FALSE)))

43-1

Exercise

TRUE \/ (TRUE /\ FALSE)

TRUE

TRUE /\ (FALSE /\ (TRUE \/ FALSE))

FALSE /\ (TRUE /\ ((FALSE \/ TRUE) /\ (TRUE \/ FALSE)))

43-2

Exercise

TRUE \/ (TRUE /\ FALSE)

TRUE

TRUE /\ (FALSE /\ (TRUE \/ FALSE))

TRUE

FALSE /\ (TRUE /\ ((FALSE \/ TRUE) /\ (TRUE \/ FALSE)))

43-3

Exercise

TRUE \/ (TRUE /\ FALSE)

TRUE

TRUE /\ (FALSE /\ (TRUE \/ FALSE))

TRUE

FALSE /\ (TRUE /\ ((FALSE \/ TRUE) /\ (TRUE \/ FALSE)))

FALSE

43-4

TLA + Syntax

TLA+ specs make heavy use of formulas like this

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

44-1

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.

44-2

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)
```

44-3

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

44-4

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)
```

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

44-5

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

44-6

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

44-7

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

44-8

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

44-9

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

44-11

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

44-10

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

44-12

TLA+ Syntax

In general:

45-1

TLA+ Syntax

In general:

$$\begin{array}{ccc} \wedge & & \wedge \\ \wedge & \wedge & \wedge \\ \wedge & \wedge & \wedge \\ A & \wedge & B & \wedge & C & \wedge & \dots & \wedge & N & = & \dots \\ & & & & & & & & & & & \wedge \\ & & & & & & & & & & & N \end{array}$$

45-2

TLA+ Syntax

In general:

$$\begin{array}{ccc} \wedge & & \wedge \\ \wedge & \wedge & \wedge \\ \wedge & \wedge & \wedge \\ A & \wedge & B & \wedge & C & \wedge & \dots & \wedge & N & = & \dots \\ & & & & & & & & & & & \wedge \\ & & & & & & & & & & & N \end{array}$$

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

45-3

TLA+ Syntax

In general:

$$\begin{array}{ccc} \wedge & & \wedge \\ \wedge & \wedge & \wedge \\ \wedge & \wedge & \wedge \\ A & \wedge & B & \wedge & C & \wedge & \dots & \wedge & N & = & \dots \\ & & & & & & & & & & & \wedge \\ & & & & & & & & & & & N \end{array}$$

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

45-4

TLA+ Syntax

In general:

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

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

45-5

TLA+ Syntax

In general:

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

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

45-6

TLA+ Syntax

In general:

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

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

45-7

TLA+ Syntax

TLA+ does not allow “ambiguous” formulas:

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

46-1

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.

46-2

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)

46-3

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.)

46-4

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)

47-1

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.

47-2

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.

47-3

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.

47-4

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.

47-5

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.

47-6

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.

47-7

TLA+ Syntax

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

48-1

TLA+ Syntax

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

$/\ A$

48-2

TLA+ Syntax

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

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

48-3

TLA+ Syntax

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

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

48-4

TLA+ Syntax

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

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

48-5

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

48-6

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

48-7

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

48-8

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

48-9

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

48-10

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.

48-11

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

48-12

TLA+ Syntax

We’ll create indent levels for each of these

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

49-1

TLA+ Syntax

We’ll create indent levels for each of these

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

```
/\ A
```

49-2

TLA+ Syntax

We'll create indent levels for each of these

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

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

49-3

TLA+ Syntax

We'll create indent levels for each of these

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

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

49-4

TLA+ Syntax

We'll create indent levels for each of these

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

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

49-5

TLA+ Syntax

We'll create indent levels for each of these

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

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

49-6

TLA+ Syntax

We'll create indent levels for each of these

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

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

49-7

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
```

49-8

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
```

49-9

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
```

49-10

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 .

49-11

TLA+ Syntax

What about this? How would this look?

A /\ B \/\ C /\ D

50-1

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.

50-2

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)

50-3

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

50-4

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

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

50-5

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

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

50-6

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

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

50-7

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
 \vee
 \wedge
 \vee

50-8

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
 \vee
 \wedge
 \vee

50-9

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
 \vee
 \wedge
 \vee

50-10

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
 \vee
 \wedge
 \vee

50-11

Exercise

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

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

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

51

Solutions

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

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

52-1

Solutions

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

$\vee \boxed{A}$
 $\vee \wedge B$
 $\wedge C \vee D$
 $\wedge E$

52-2

Solutions

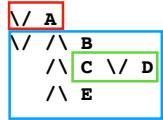
1. $\boxed{A} \vee (\boxed{B} \wedge (C \vee D) \wedge E)$

$\vee \boxed{A}$
 $\vee \wedge \boxed{B}$
 $\wedge C \vee D$
 $\wedge E$

52-3

Solutions

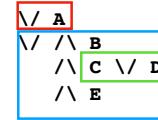
1. $\boxed{A} \vee (B \wedge (\textcolor{blue}{C \vee D}) \wedge E)$



52-4

Solutions

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



Alternate Solution

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

52-5

Solutions

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

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

53-1

Solutions

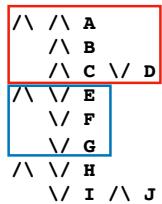
2. $(\boxed{A \wedge B \wedge (C \vee D)}) \wedge (E \vee F \vee G) \wedge (H \vee (I \wedge J))$

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

53-2

Solutions

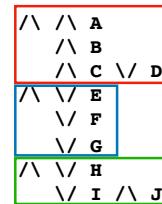
$$2. (\text{A} \wedge \text{B} \wedge (\text{C} \vee \text{D})) \wedge (\text{E} \vee \text{F} \vee \text{G}) \wedge (\text{H} \vee (\text{I} \wedge \text{J}))$$



53-3

Solutions

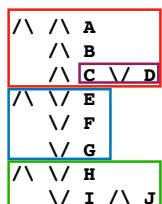
$$2. (\text{A} \wedge \text{B} \wedge (\text{C} \vee \text{D})) \wedge (\text{E} \vee \text{F} \vee \text{G}) \wedge (\text{H} \vee (\text{I} \wedge \text{J}))$$



53-4

Solutions

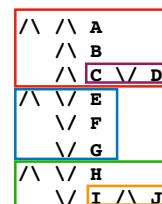
$$2. (\text{A} \wedge \text{B} \wedge (\text{C} \vee \text{D})) \wedge (\text{E} \vee \text{F} \vee \text{G}) \wedge (\text{H} \vee (\text{I} \wedge \text{J}))$$



53-5

Solutions

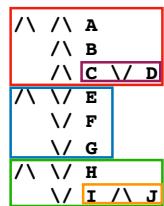
$$2. (\text{A} \wedge \text{B} \wedge (\text{C} \vee \text{D})) \wedge (\text{E} \vee \text{F} \vee \text{G}) \wedge (\text{H} \vee (\text{I} \wedge \text{J}))$$



53-6

Solutions

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



Alternate Solution

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

53-7

Solutions

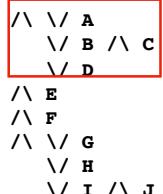
$$3. (\text{A} \vee (\text{B} \wedge \text{C}) \vee \text{D}) \wedge \text{E} \wedge \text{F} \wedge (\text{G} \vee \text{H} \vee (\text{I} \wedge \text{J}))$$

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

54-1

Solutions

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



Solutions

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

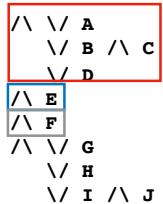
$$\begin{array}{l} \wedge \vee \text{A} \\ \wedge \text{B} \wedge \text{C} \\ \wedge \text{D} \\ \wedge \text{E} \\ \wedge \text{F} \\ \wedge \vee \text{G} \\ \wedge \text{H} \\ \wedge \text{I} \vee \text{J} \end{array}$$

54-2

54-3

Solutions

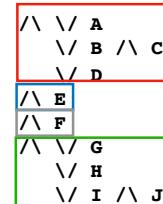
$$3. (\text{A} \vee \neg (\text{B} \wedge \text{C}) \vee \text{D}) \wedge \text{E} \wedge \neg \text{F} \wedge (\text{G} \vee \text{H} \vee \neg (\text{I} \wedge \text{J}))$$



54-4

Solutions

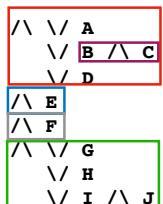
$$3. (\text{A} \vee \neg (\text{B} \wedge \text{C}) \vee \text{D}) \wedge \text{E} \wedge \neg \text{F} \wedge (\text{G} \vee \text{H} \vee \neg (\text{I} \wedge \text{J}))$$



54-5

Solutions

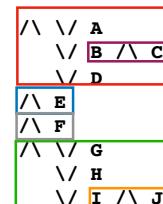
$$3. (\text{A} \vee \neg (\text{B} \wedge \text{C}) \vee \text{D}) \wedge \text{E} \wedge \neg \text{F} \wedge (\text{G} \vee \text{H} \vee \neg (\text{I} \wedge \text{J}))$$



54-6

Solutions

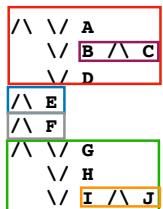
$$3. (\text{A} \vee \neg (\text{B} \wedge \text{C}) \vee \text{D}) \wedge \text{E} \wedge \neg \text{F} \wedge (\text{G} \vee \text{H} \vee \neg (\text{I} \wedge \text{J}))$$



54-7

Solutions

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



Alternate Solution

$$\begin{aligned} &\vee \vee A \\ &\vee \vee B \\ &\quad \vee C \\ &\quad \vee D \\ &\vee \vee E \\ &\vee \vee F \\ &\vee \vee G \\ &\quad \vee H \\ &\quad \vee I \\ &\quad \vee J \end{aligned}$$

54-8

Sets

55-1

Sets

- An unordered collection of unique things

Sets

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

55-2

55-3

Sets

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

55-4

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"}

55-5

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)

55-6

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()

55-7

Sets

56-1

Sets

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

56-2

Sets

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

56-3

Sets

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

56-4

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

56-5

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”])`

56-6

Sets

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

57-1

57-2

Sets

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

57-3

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"]))`

57-4

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

57-5

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"]))`

57-6

Sets

58-1

Sets

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

58-2

Sets

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

58-3

Sets

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

58-4

Sets

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

58-5

Sets

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

58-6

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"]))`

58-7

Sets

59-1

Sets

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

59-2

Sets

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

59-3

Sets

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

59-4

Sets

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

59-5

Sets

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

59-6

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:

59-7

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”]))`

59-8

Sets

60-1

Sets

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

60-2

Sets

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

60-3

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))`

60-4

Exercises

(TRUE or FALSE for each)

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

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

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

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

61-1

Exercises

(TRUE or FALSE for each)

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

TRUE

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

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

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

61-2

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

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

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

61-3

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

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

61-4

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

61-5

“There exists”

62-1

“There exists”

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

62-2

“There exists”

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

62-3

“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”

62-4

“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”

62-5

“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”

62-6

“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’

62-7

“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’

62-8

“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

62-9

“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

62-10

“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
- $\exists x \in \{1,2,3,4\} : x > 3$ is TRUE
- $\exists x \in \{1,2,3,4\} : x > 5$ is FALSE

62-11

“There exists”

63-1

“There exists”

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

is roughly equivalent to the following Python expressions

63-2

“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
```

63-3

“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
```

63-4

“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
```

63-5

“For all”

64-1

“For all”

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

64-2

“For all”

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

64-3

“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”

64-4

“For all”

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

64-5

“For all”

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

64-6

“For all”

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

64-7

“For all”

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

64-8

“For all”

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

64-9

“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

64-10

“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

64-11

“For all”

65-1

“For all”

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

is roughly equivalent to the following Python expressions

65-2

“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
```

65-3

“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
```

65-4

“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
```

65-5

Exercises

$$\exists x \in \{1,2,3,4\} : (x > 3) \wedge (x < 4)$$
$$\exists x \in \{1,2,3,4\} : \exists y \in \{5, 6, 7\} : x < y$$
$$\forall x \in \{1,2,3\} : \{x\} \subseteq \{1, 2\}$$
$$\exists x \in \{1,2,3\} : \{x\} \subseteq \{1, 2\}$$
$$\forall x \in \{3, 4, 5\} : \wedge x > 1 \\ \wedge x < 6$$

66-1

Exercises

$$\exists x \in \{1,2,3,4\} : (x > 3) \wedge (x < 4)$$
 FALSE
$$\exists x \in \{1,2,3,4\} : \exists y \in \{5, 6, 7\} : x < y$$
$$\forall x \in \{1,2,3\} : \{x\} \subseteq \{1, 2\}$$
$$\exists x \in \{1,2,3\} : \{x\} \subseteq \{1, 2\}$$
$$\forall x \in \{3, 4, 5\} : \wedge x > 1 \\ \wedge x < 6$$

66-2

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
$$\forall x \in \{1,2,3\} : \{x\} \subseteq \{1, 2\}$$
$$\exists x \in \{1,2,3\} : \{x\} \subseteq \{1, 2\}$$
$$\forall x \in \{3, 4, 5\} : \wedge x > 1 \\ \wedge x < 6$$

66-3

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
$$\exists x \in \{1,2,3\} : \{x\} \subseteq \{1, 2\}$$
$$\forall x \in \{3, 4, 5\} : \wedge x > 1 \\ \wedge x < 6$$

66-4

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

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

66-5

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

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

66-6

Set constructors

Set constructors

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

67-1

67-2

Set constructors

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

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

67-3

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:

67-4

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])
```

67-5

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]))
```

67-6

Set constructors

68-1

Set constructors

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

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

68-2

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:

68-3

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]])`

68-4

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]))
```

68-5

Solutions

70-1

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 $\{1, 3, 5, 7, 9\}$).

69

Solutions

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

70-2

Solutions

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

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

70-3

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

70-4

Back to State Machines



71

72-1

Back to State Machines

Recall that a state machine is defined by three things:

72-2

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

72-4

Back to State Machines

Recall that a state machine is defined by three things:

1. The **variables**

72-3

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
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**)

72-5

A small C program

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

73-1

A small C program

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

73-2

A small C program

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

73-3

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

73-4

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

73-5

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
- Next states for some given state... let's see!

73-7

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

73-6

A small C program - example behaviour

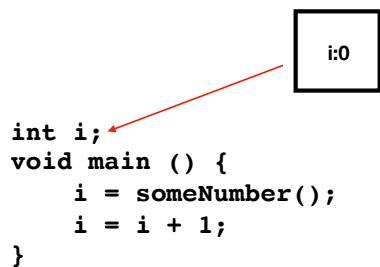
Assume that for this particular behaviour, `someNumber()` returns 42.

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

74-1

A small C program - example behaviour

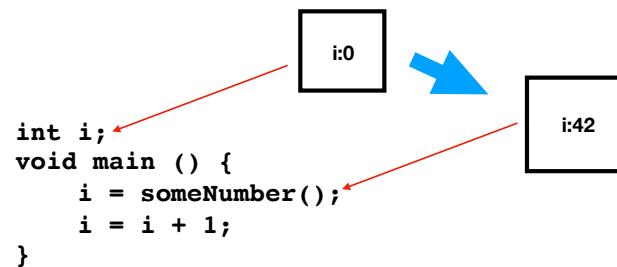
Assume that for this particular behaviour, `someNumber()` returns 42.



74-2

A small C program - example behaviour

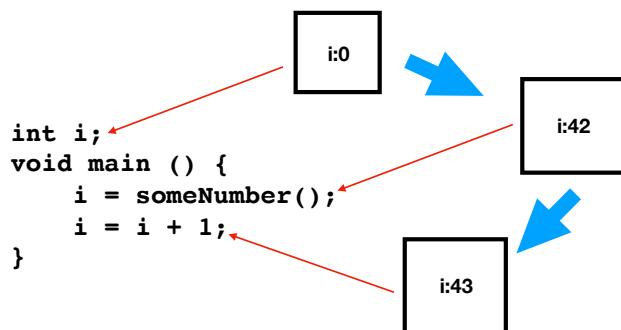
Assume that for this particular behaviour, `someNumber()` returns 42.



74-3

A small C program - example behaviour

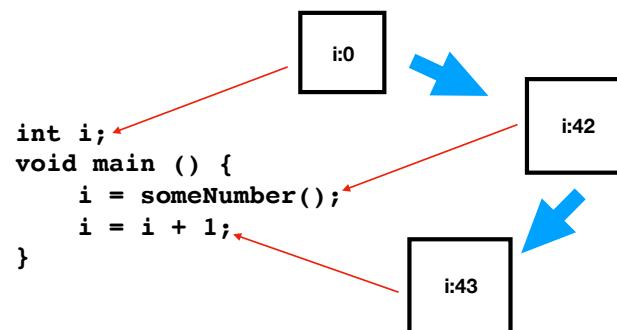
Assume that for this particular behaviour, `someNumber()` returns 42.



74-4

A small C program - example behaviour

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

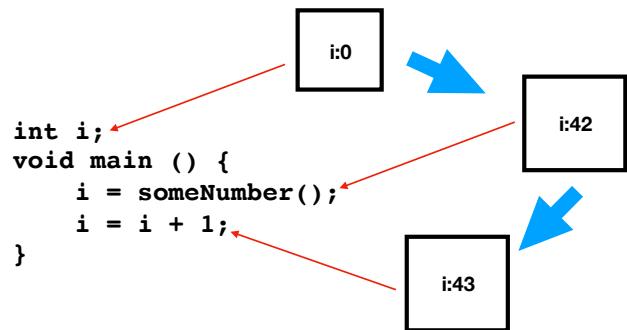


So we know that for the state [i:42], the only next possible state is [i:43].

74-5

A small C program - example behaviour

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

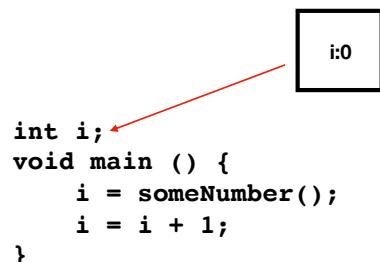


So we know that for the state `[i:42]`, the only next possible state is `[i:43]`.
And for the state `[i:43]`, the only next state is “program termination”.

74-6

A small C program - example behaviour

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



75-2

A small C program - example behaviour

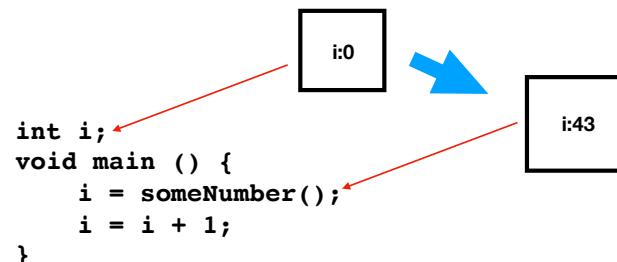
Now let's look at a behaviour where `someNumber()` returned 43.

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

75-1

A small C program - example behaviour

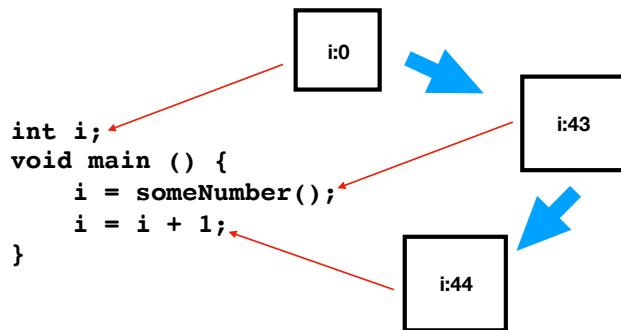
Now let's look at a behaviour where `someNumber()` returned 43.



75-3

A small C program - example behaviour

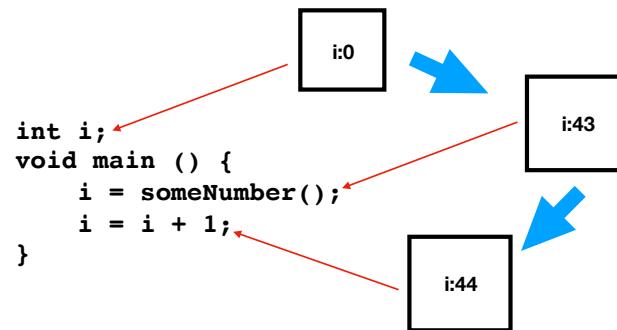
Now let's look at a behaviour where `someNumber()` returned 43.



75-4

A small C program - example behaviour

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

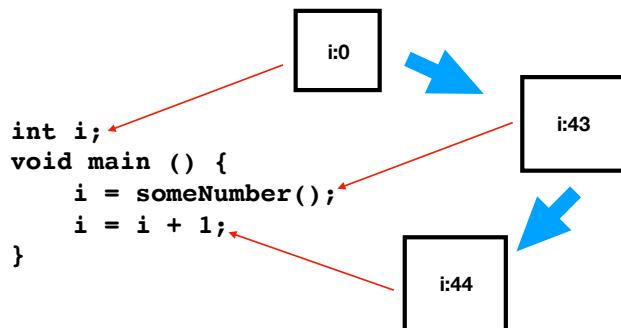


In this case, a state of `[i:43]` has a next state of `[i:44]`.

75-5

A small C program - example behaviour

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

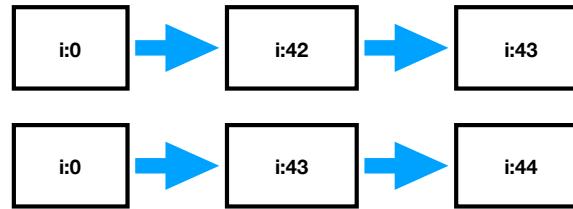


In this case, a state of `[i:43]` has a next state of `[i:44]`.

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

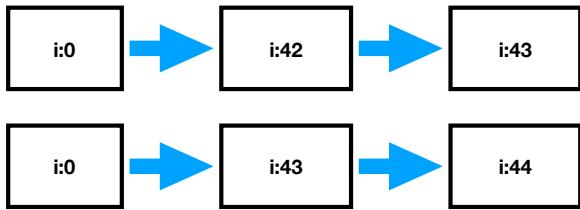
75-6

43



76-1

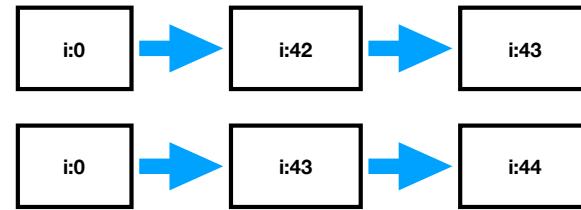
43



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

76-2

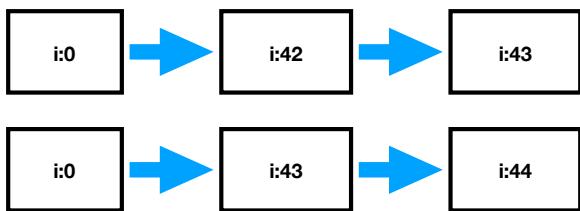
43



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

76-3

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.

76-4

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

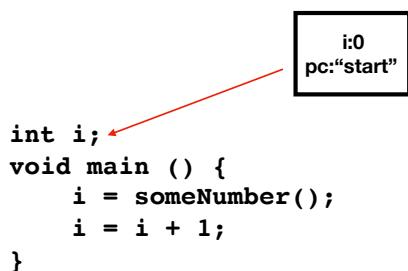
```
int i;
void main () {
    i = someNumber();
    i = i + 1;
}
```

77-1

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

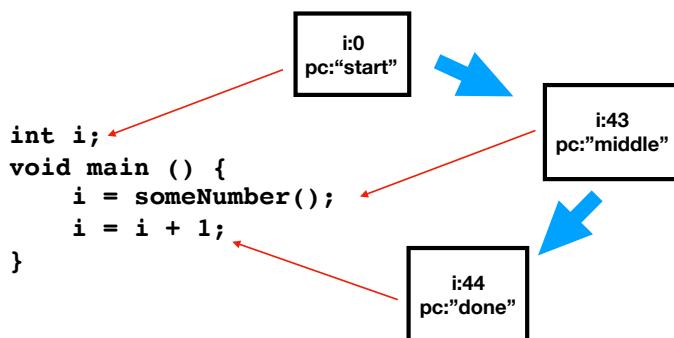


77-2

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

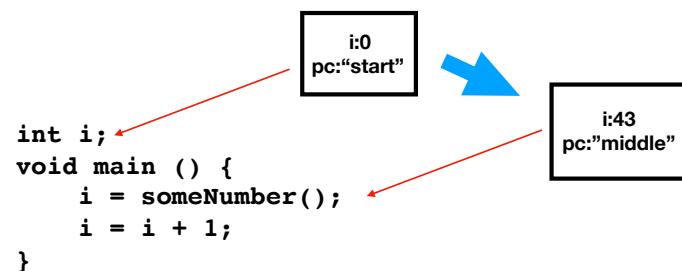


77-4

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



77-3

A small C program

Let's make this a bit more formal

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

78-1

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;
}
```

78-2

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;
}
```

78-4

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;
}
```

78-3

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
```

78-5

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;
}
```

```
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

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
```

78-7

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
```

78-8

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
```

78-9

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
```

78-10

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
```

79-1

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"
```

79-2

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"
```



79-3

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
```



79-4

A small C program

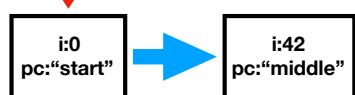
```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```



79-5

A small C program

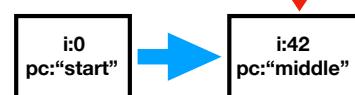
```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```



79-6

A small C program

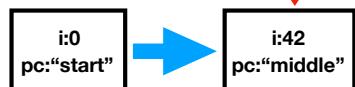
```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```



79-7

A small C program

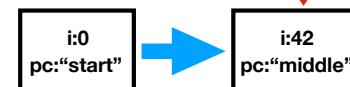
```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```



79-8

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
```



79-9

A small C program

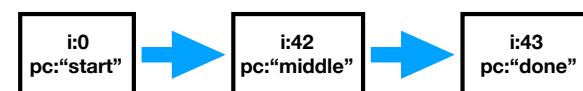
```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```



79-10

A small C program

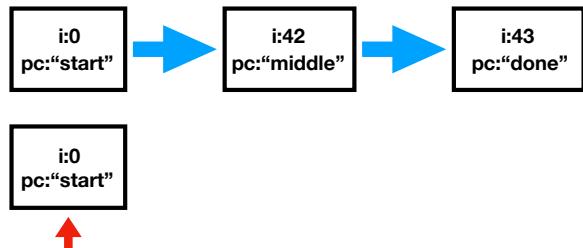
```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```



80-1

A small C program

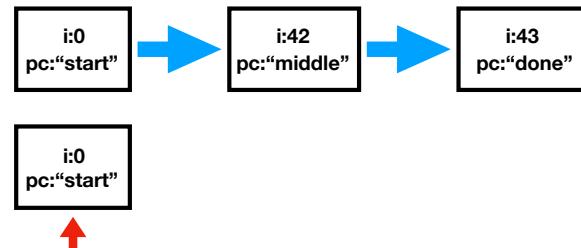
```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```



80-2

A small C program

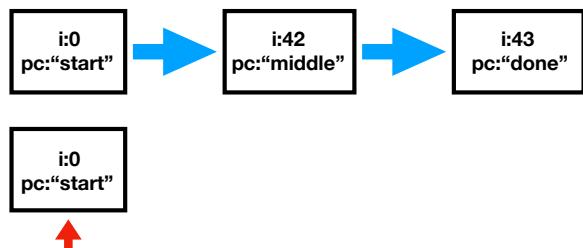
```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```



80-3

A small C program

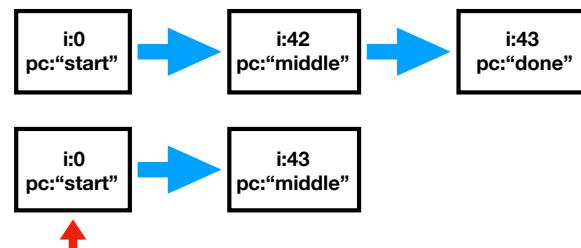
```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```



80-4

A small C program

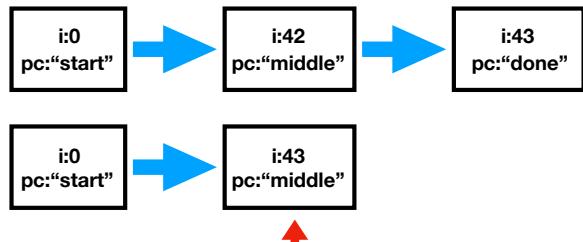
```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```



80-5

A small C program

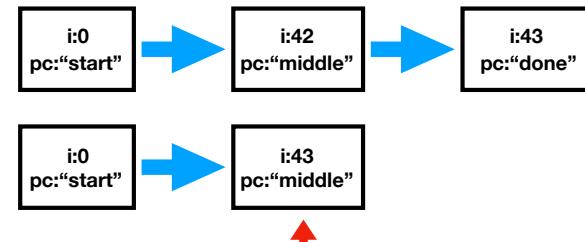
```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```



80-6

A small C program

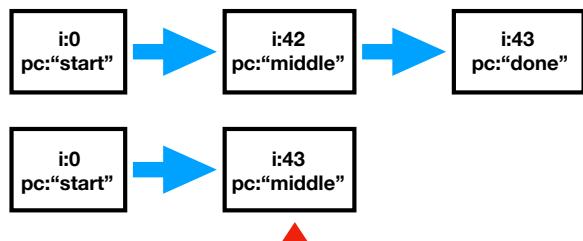
```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```



80-7

A small C program

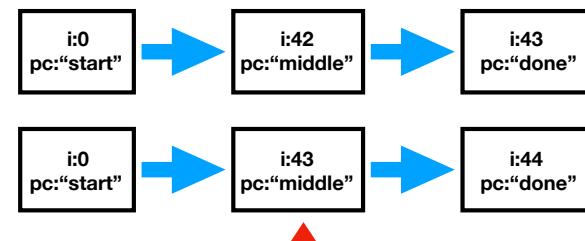
```
if current value of pc = "start"
then
    next value of i ∈ 1..1000
    next value of pc = "middle"
else if current value of pc = "middle"
then
    next value of i = current value of i + 1
    next value of pc = "done"
else
    no next values
```



80-8

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
```



80-9

Improving the next state formula

81-1

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
```

81-2

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
```

82-1

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

82-2

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

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
```

84-1

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
```

83

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

84-2

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

84-3

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
```

85

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

84-4

= doesn't mean assignment!!!

86-1

= 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"

86-2

= 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
```

86-4

= 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
```

86-3

= 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
```

86-5

= 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"

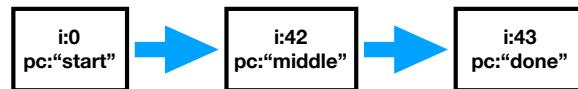
```
if pc = "start"
then
    i' ∈ 1..1000
    pc' = "middle"
else if pc = "middle"
then
    i' = i + 1
    pc' = "done"
else
    no next values
```

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

86-6

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
```



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
```



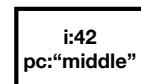
Current state

Next state

88

State pairs

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



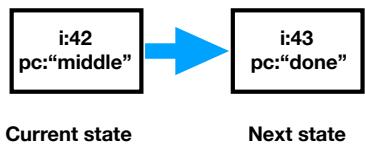
Current state

Next state

89

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

90

State pairs

91-1

State pairs

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

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:

91-2

91-3

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.**

91-4

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.

91-5

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**.

91-6

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).

91-7

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!

91-8

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
```

92-1

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

92-2

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

92-3

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
```

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

93

A little bit more formal

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

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

95

A little bit more formal

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

96

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

97

Looking pretty good...

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

99

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

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
```

100-1

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?

100-2

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

100-3

Case 1

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

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

101-1

101-2

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

101-3

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

101-4

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

101-5

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:

101-6

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:

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

101-7

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:

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

101-8

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:

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

101-9

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:

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

101-10

Case 2

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

102-1

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

102-2

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

102-3

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

102-4

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

102-5

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:

102-6

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:

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

102-7

Rewrite!

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

103-1

Rewrite!

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

becomes

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")))

103-2

103-3

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")))

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")))

103-4

103-5

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")))
```

103-6

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")))
```

103-7

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")))
```

103-8

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")))
```

103-9

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.)

103-10

TLA+ Syntax Replacement

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

TLA+ Syntax Replacement

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

becomes

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"
```

104-2

104-3

Comparison

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

105-1

Comparison

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

105-2

Notes

106-1

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.

106-2

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.
- `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.

106-3

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.

106-4

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”.

106-5

An aside

107-1

An aside

- Specifying at this level is incredibly rare.

107-2

An aside

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

107-3

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.

107-4

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.

107-5

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.

107-6

Next State

Next State

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

108-2

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.

108-3

Next State

```
\/ /\ pc = "start"
 /\ i' ∈ 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’ ”.

108-4

Next State

```
\/ /\ pc = "start"
 /\ i' ∈ 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’ ”.
- So... What's actually going on? What is this really saying?

108-5

Behaviours

- Think back to our Die Hard example

109-1

Behaviours

109-2

Behaviours

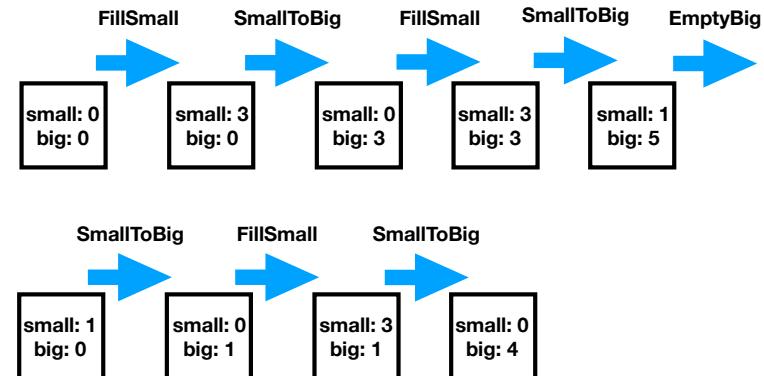
- Think back to our Die Hard example
- We showed three possible behaviours

109-3

Behaviours

- Think back to our Die Hard example

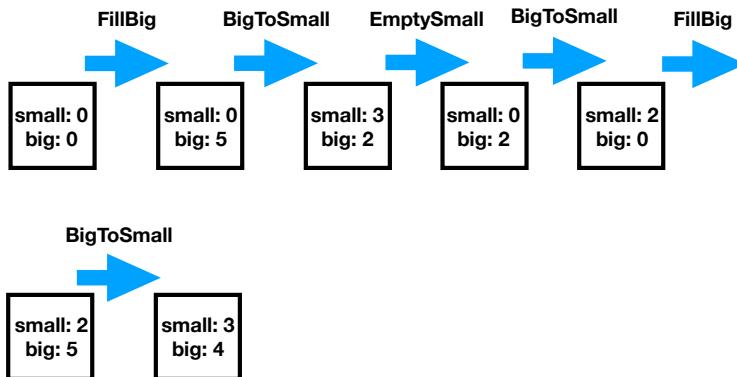
- We showed three possible behaviours



109-4

Behaviours

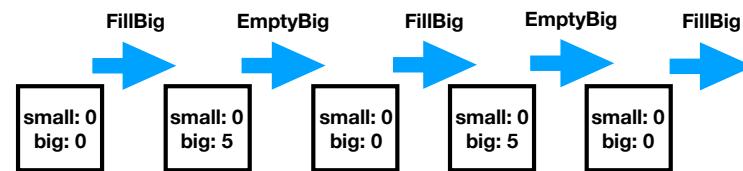
- Think back to our Die Hard example
- We showed three possible behaviours



109-5

Behaviours

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



109-6

Behaviours

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

109-7

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**

109-8

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**



109-9

Behaviours

110-1

Behaviours

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

110-2

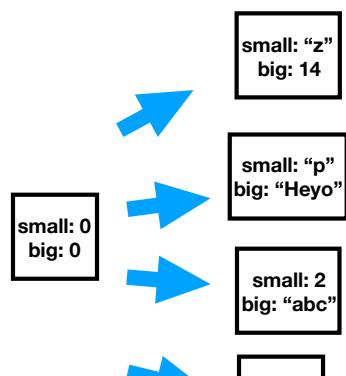
Behaviours

- 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.

110-3

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.



110-4

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.

110-5

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.
- 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+.

110-6

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.

110-7

Behaviours

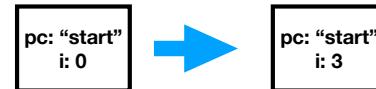
```
\forall \exists pc = "start"  
      \exists i' \in 1..1000  
      \exists pc' = "middle"  
\forall \exists pc = "middle"  
      \exists i' = i + 1  
      \exists pc' = "done"
```



111-1

Behaviours

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



111-2

Behaviours

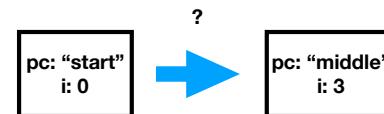
```
\/ /\ pc = "start"
 /\ i' ∈ 1..1000
 /\ pc' = "middle"
/\ /\ pc = "middle"
 /\ i' = i + 1
 /\ pc' = "done"
```



111-3

Behaviours

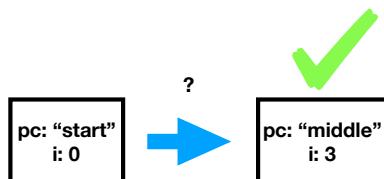
```
\/ /\ pc = "start"
 /\ i' ∈ 1..1000
 /\ pc' = "middle"
/\ /\ pc = "middle"
 /\ i' = i + 1
 /\ pc' = "done"
```



111-4

Behaviours

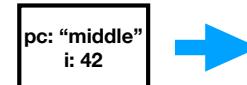
```
\/ /\ pc = "start"
 /\ i' ∈ 1..1000
 /\ pc' = "middle"
/\ /\ pc = "middle"
 /\ i' = i + 1
 /\ pc' = "done"
```



111-5

Behaviours

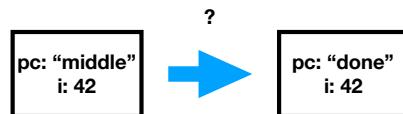
```
\/ /\ pc = "start"
 /\ i' ∈ 1..1000
 /\ pc' = "middle"
/\ /\ pc = "middle"
 /\ i' = i + 1
 /\ pc' = "done"
```



112-1

Behaviours

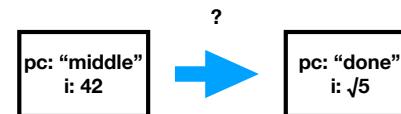
```
\ / \ pc = "start"
  \ i' ∈ 1..1000
    \ pc' = "middle"
\ / \ pc = "middle"
  \ i' = i + 1
  \ pc' = "done"
```



112-2

Behaviours

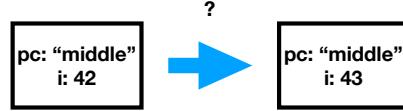
```
\ / \ pc = "start"
  \ i' ∈ 1..1000
    \ pc' = "middle"
\ / \ pc = "middle"
  \ i' = i + 1
  \ pc' = "done"
```



112-3

Behaviours

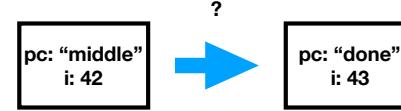
```
\ / \ pc = "start"
  \ i' ∈ 1..1000
    \ pc' = "middle"
\ / \ pc = "middle"
  \ i' = i + 1
  \ pc' = "done"
```



112-4

Behaviours

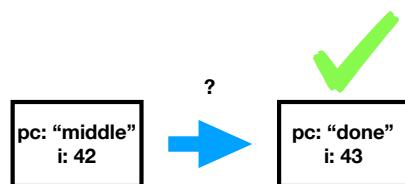
```
\ / \ pc = "start"
  \ i' ∈ 1..1000
    \ pc' = "middle"
\ / \ pc = "middle"
  \ i' = i + 1
  \ pc' = "done"
```



112-5

Behaviours

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

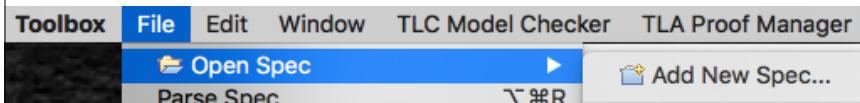


112-6

Final Spec for our C Program

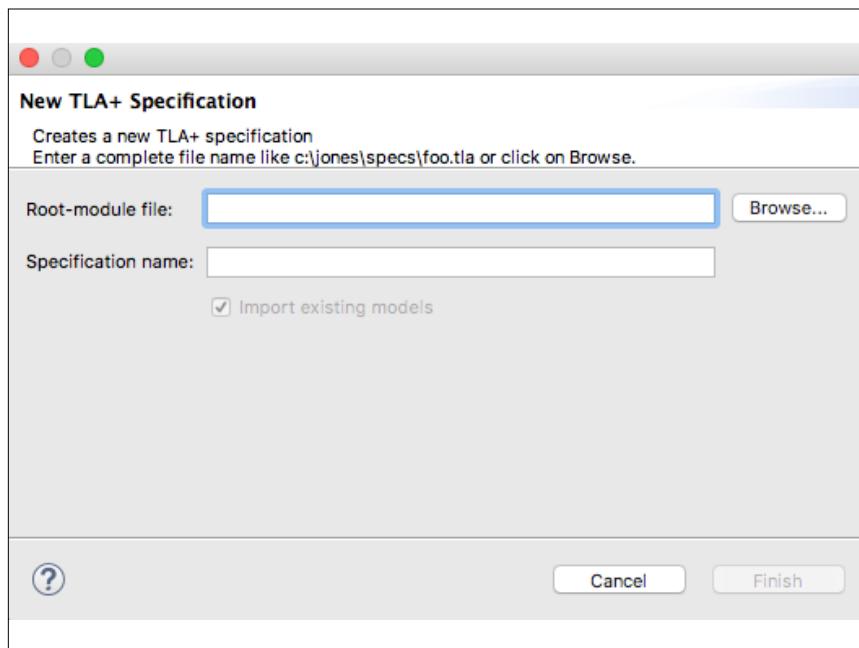
Final Spec for our C Program

```
— 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"
```

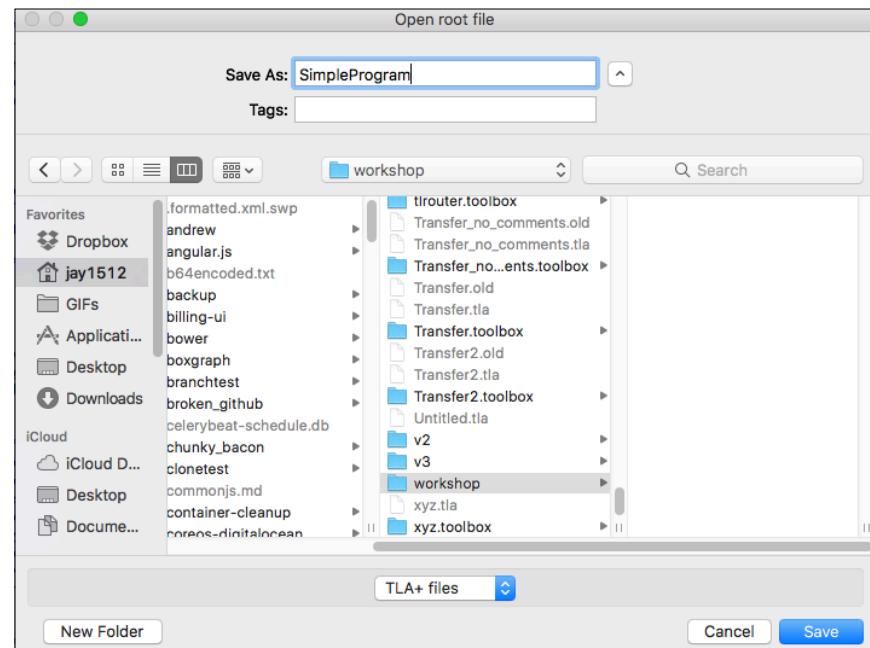


113-2

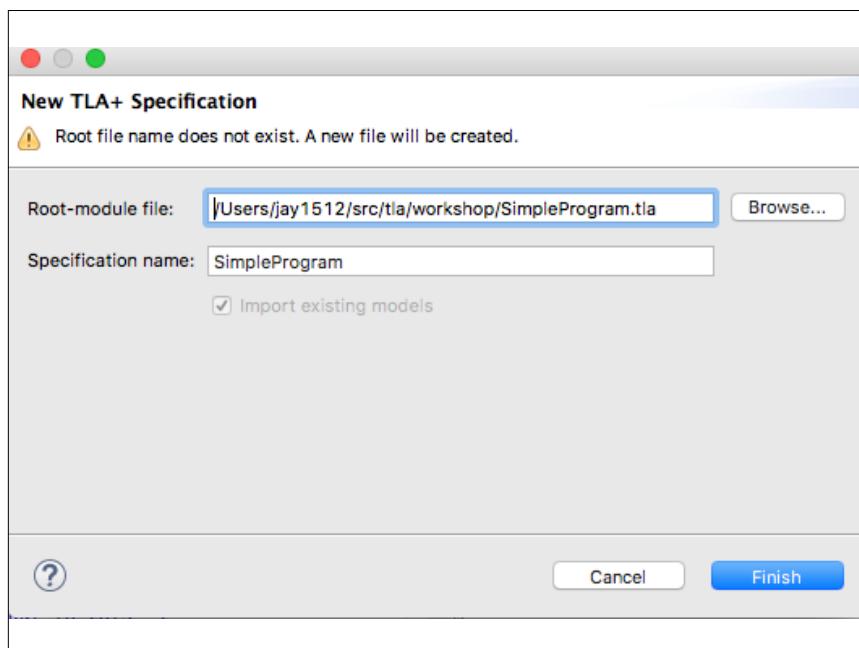
114



115



116



117

The editor window title is "SimpleProgram.tla" and the path is "/Users/jay1512/src/tla/workshop/SimpleProgram.tla". The code area is titled "TLA Module" and contains the following text:

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

118

The screenshot shows a TLA+ editor window with the file `SimpleProgram.tla` open. The code defines a module `SimpleProgram` that extends `Integers` and declares variables `i` and `pc`. It includes an initial state `Init` where `pc = "start"` and `i = 0`, and a next state `Next` that transitions through states `"start"`, `"middle"`, and `"done"`, with `i` incrementing by 1 each time. The code is annotated with modification history information.

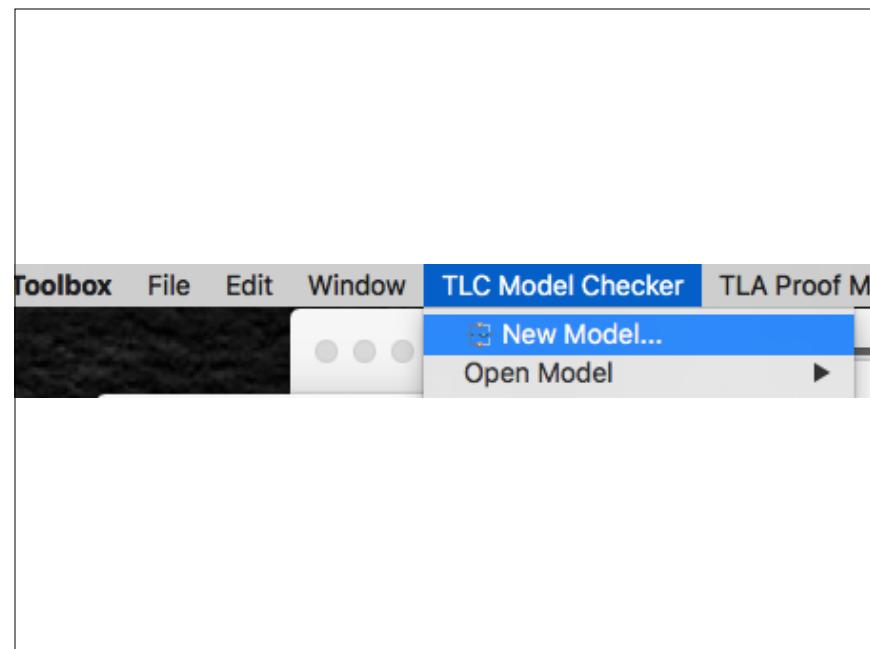
```

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

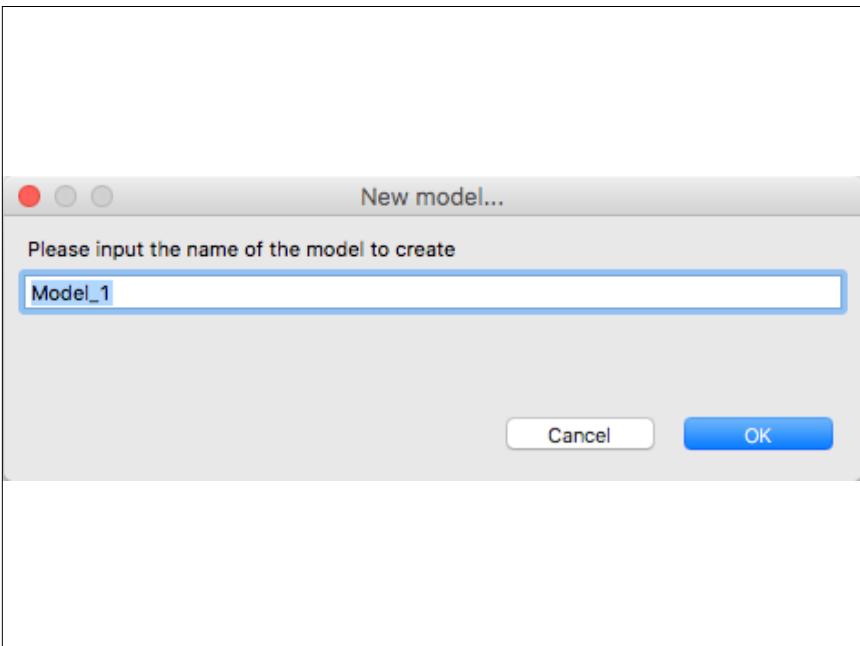
```

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

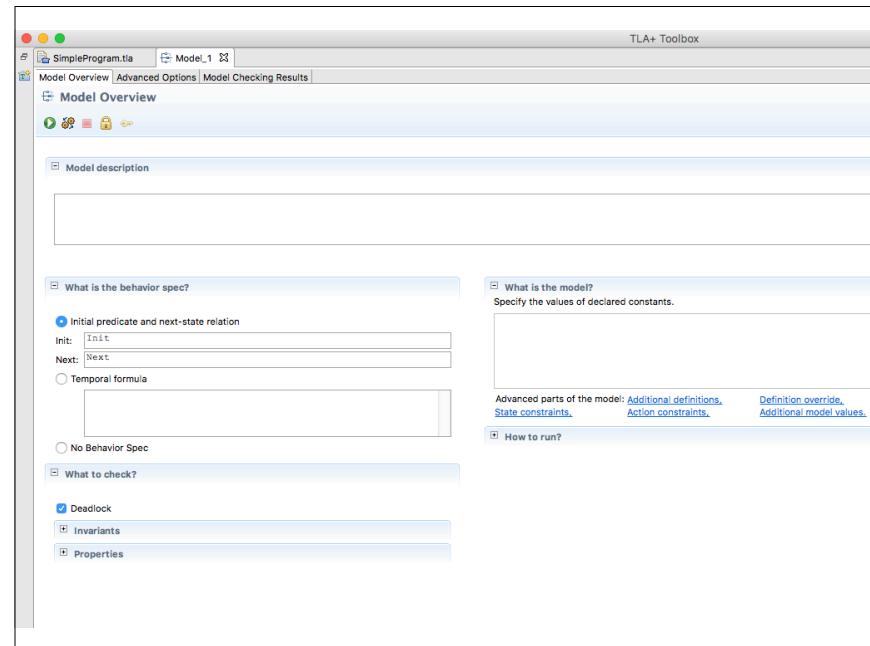
119



120



121



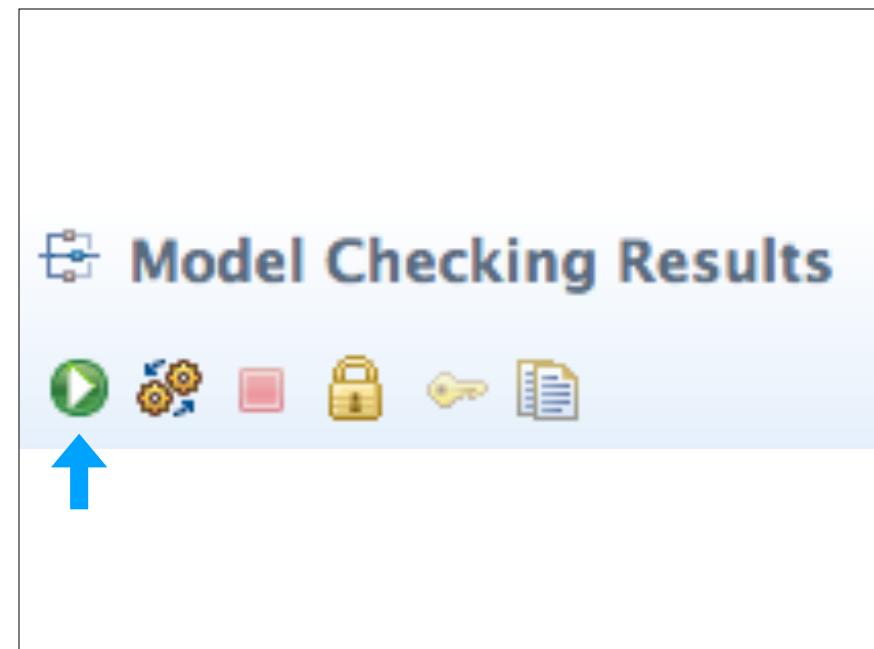
122

```

SimpleProgram.tla
Users/jay1512/src/tla/workshop/SimpleProgram.tla
TLA Module
1 EXTENDS Integers
2 VARIABLES i, pc
3
4 Init == (pc="start") \wedge (i=0)
5
6 Next == 
7   \vee \wedge pc = "start"
8   \wedge i' \in 1..1000
9   \wedge i' = i + 1
10  \vee \wedge pc = "middle"
11  \wedge i' = i + 1
12  \wedge \wedge pc = "done"
13
14
15 =====
16 ▾ Modification History
17 ▾ Last modified Thu Sep 14 13:04:45 EDT 2017 by jay1512
18 ▾ Created Thu Sep 14 13:03:35 EDT 2017 by jay1512
19

```

123



124

Model Overview

Model Checking Results State space exploration incomplete

TLC Errors

Deadlock reached.

Name	Value
<Initial predicate>	State (num = 1)
i	0
pc	"start"
i	3
pc	"middle"
i	4
pc	"done"

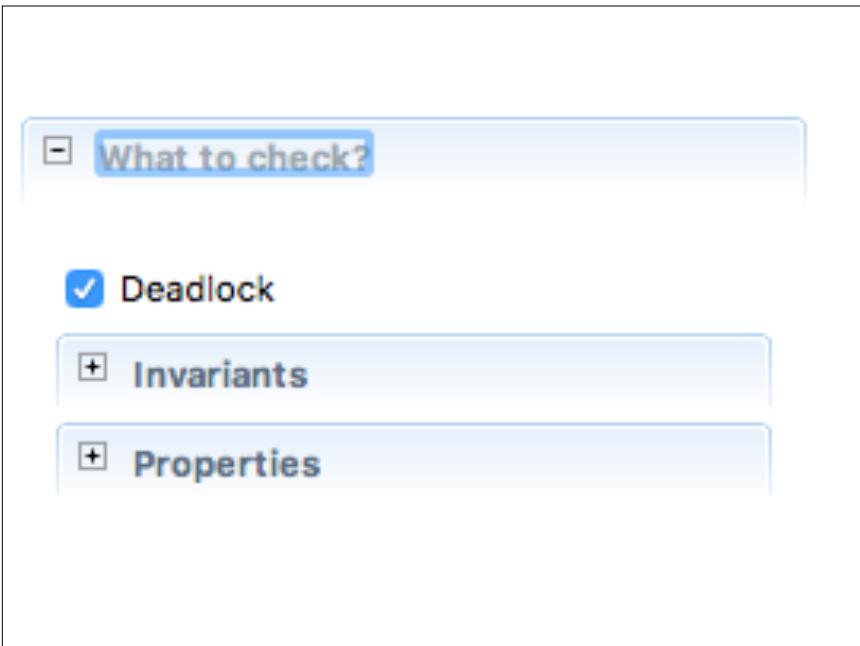
125

Model Overview

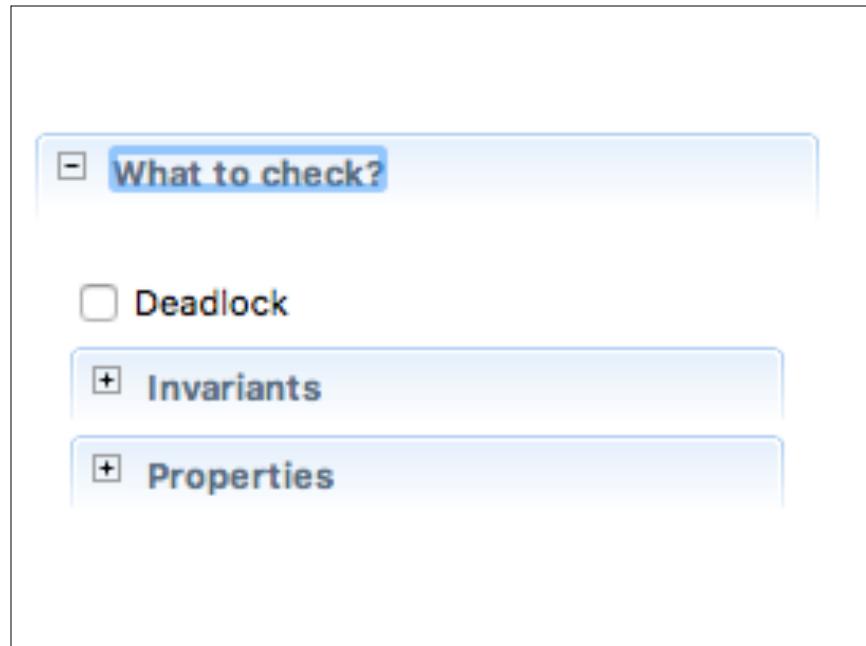
Advanced Options

Model Checking Results

126



127



128

The screenshot shows the "Model Checking Results" window for "Model_1". It includes tabs for "Model Overview", "Advanced Options", and "Model Checking Results". Under "Model Checking Results", there are sections for "General" and "Statistics".

General:

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

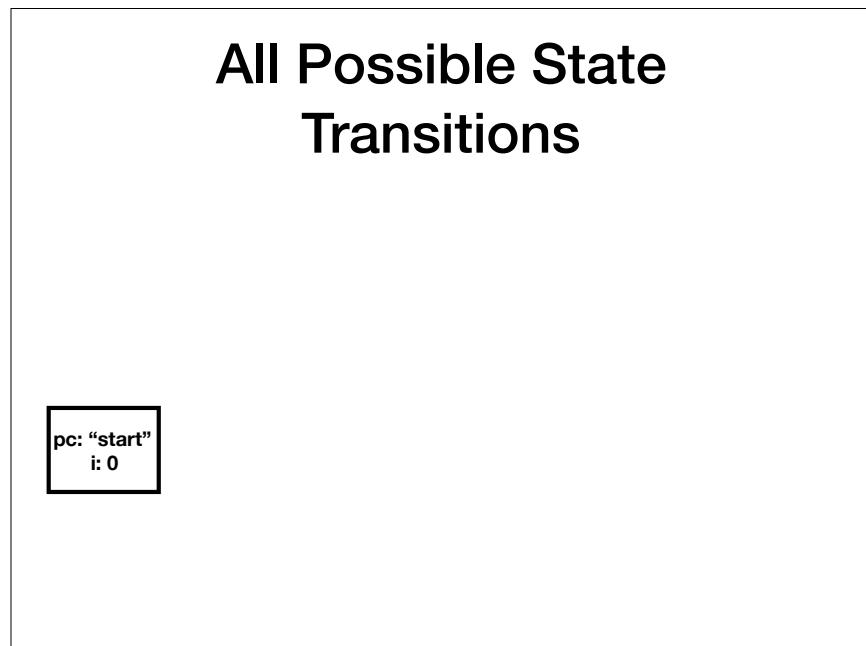
Statistics:

State space progress (click column header for graph)					Coverage at	2017-09-14 13:06:41
Time	Diameter	States Found	Distinct States	Queue Size	Module	Location
2017-09-14 13:06...	3	2001	2001	0	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:

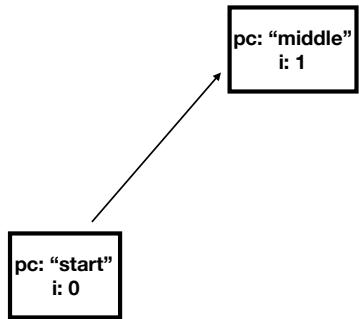
Expression: Value:

129



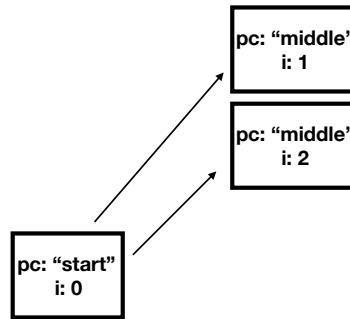
130-1

All Possible State Transitions



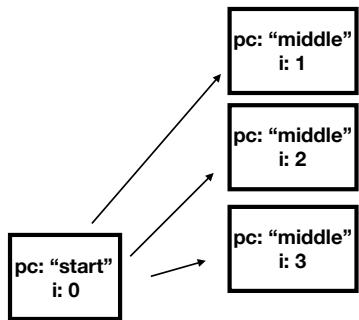
130-2

All Possible State Transitions



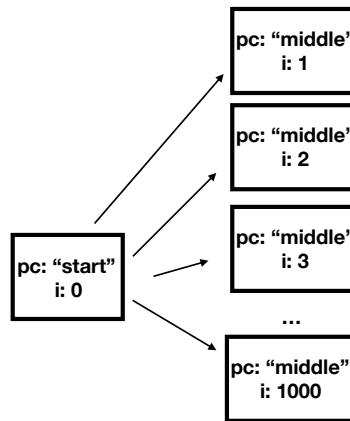
130-3

All Possible State Transitions



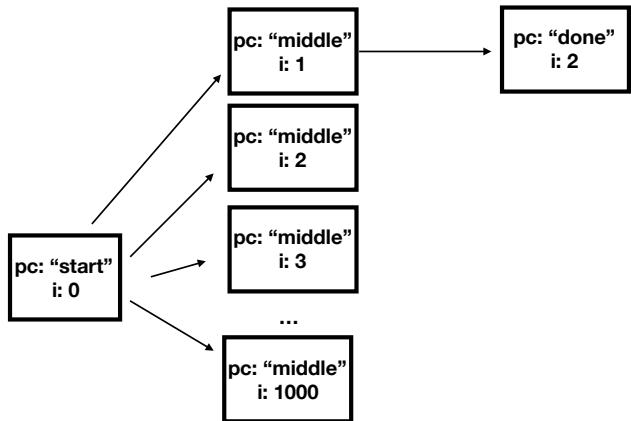
130-4

All Possible State Transitions



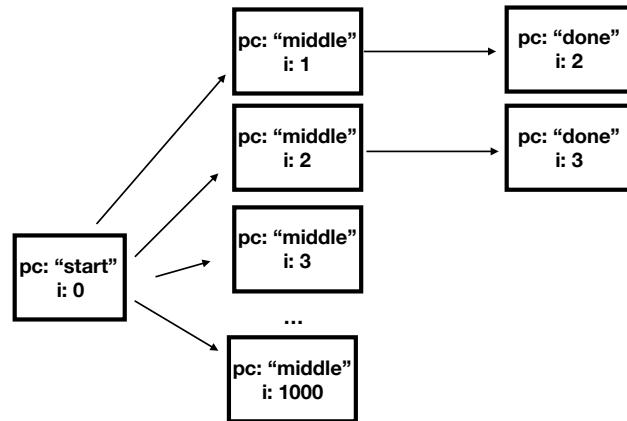
130-5

All Possible State Transitions



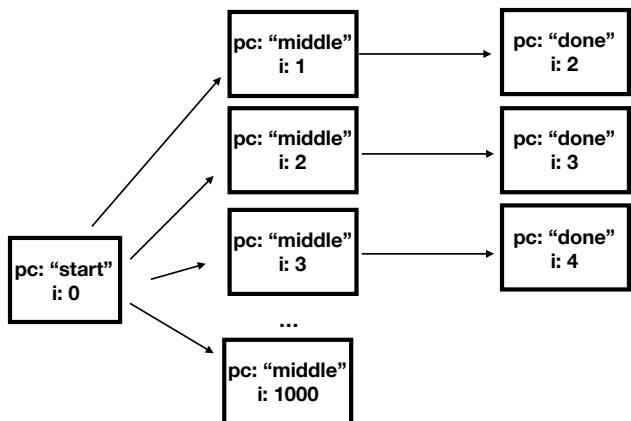
130-6

All Possible State Transitions



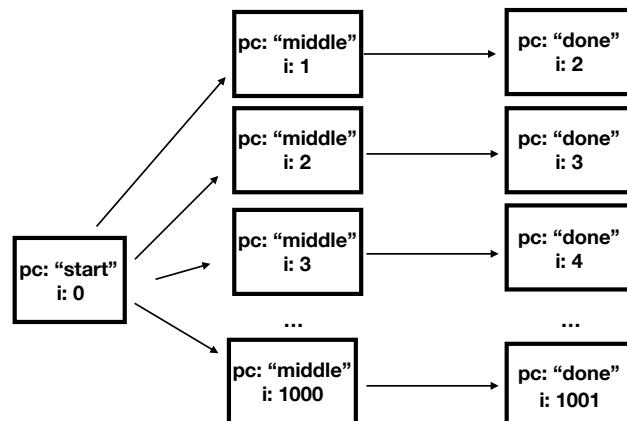
130-7

All Possible State Transitions



130-8

All Possible State Transitions



130-9

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"
```

131-1

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"
```

131-2

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"
```

131-3

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"
```

131-4

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"
```

131-5

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
```

131-6

```
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

132

What to check?

Deadlock

Invariants

Formulas true in every **reachable** state.

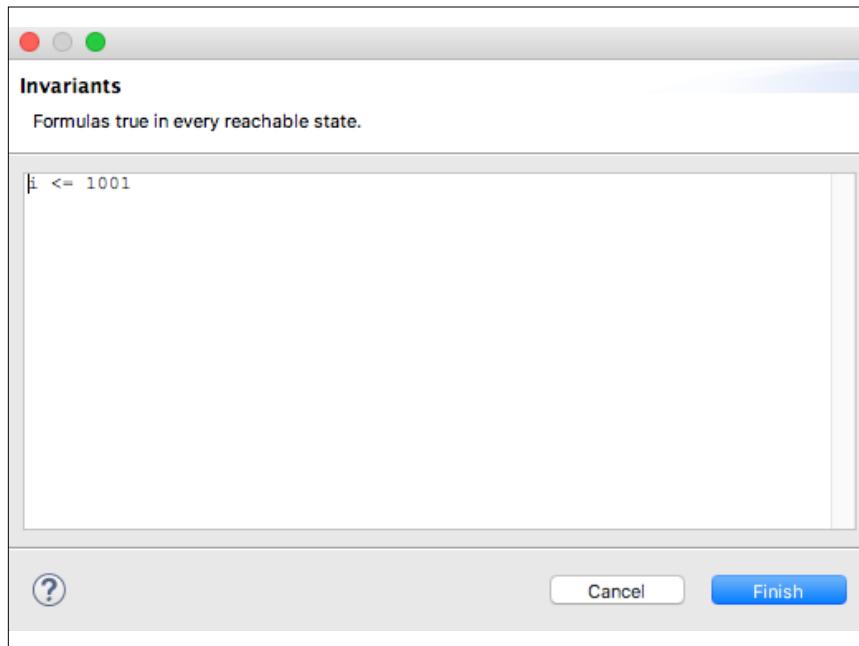
Add

Edit

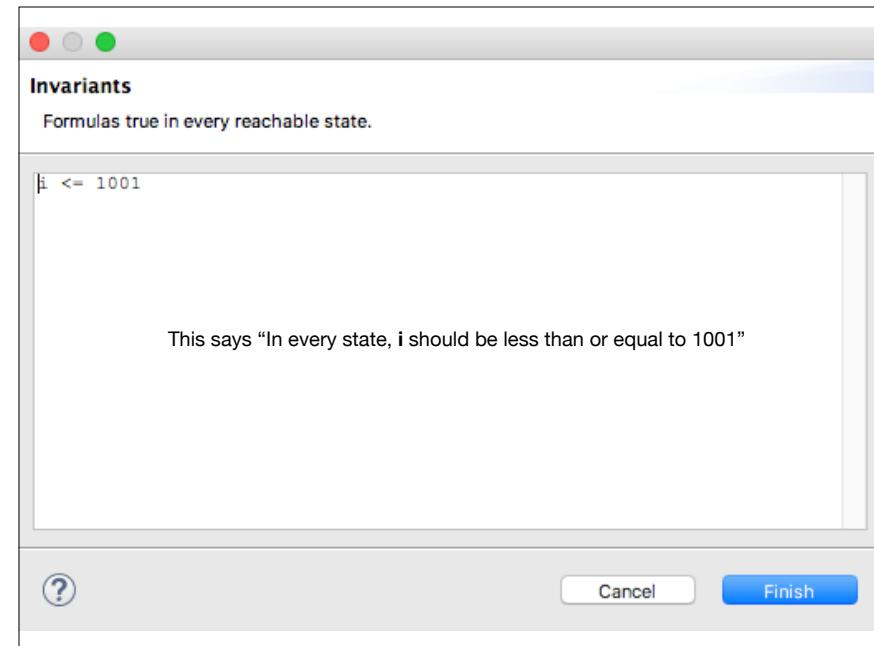
Remove

Properties

133



134-1



134-2

Run the model again,
everything should still be
fine

135

Change the Add action

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

136-1

Change the Add action

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

136-2

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"

137-1

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"

137-2

137-3

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"

138-1

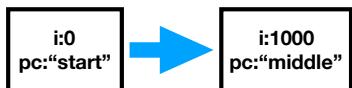
Error Trace

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"

138-2

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"



138-3

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"



138-4



139

Die Hard Variables

VARIABLES big, small

Die Hard Initial States

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

141

Die Hard Next States

142-1

Die Hard Next States

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

142-2

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**

142-3

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**

142-4

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**

142-5

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**

142-6

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**

142-7

Die Hard Next States

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

143

FillSmall

144-1

FillSmall

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

144-2

FillSmall

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

144-3

EmptySmall

EmptySmall

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

145-1

145-2

EmptySmall

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

145-3

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**

146

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

147-1

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
```

147-2

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
```

147-3

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

148-1

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)
```

148-2

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)
```

148-3

SmallToBig

Putting it all together

149-1

SmallToBig

Putting it all together

```
SmallToBig == IF big + small < 5
```

149-2

SmallToBig

Putting it all together

```
SmallToBig == IF big + small < 5  
    THEN /\ big' = big + small
```

149-3

SmallToBig

Putting it all together

```
SmallToBig == IF big + small < 5  
    THEN /\ big' = big + small  
        /\ small' = 0
```

149-4

SmallToBig

Putting it all together

```
SmallToBig == IF big + small < 5
    THEN /\ big' = big + small
        /\ small' = 0
    ELSE /\ small' = small - (5 - big)
```

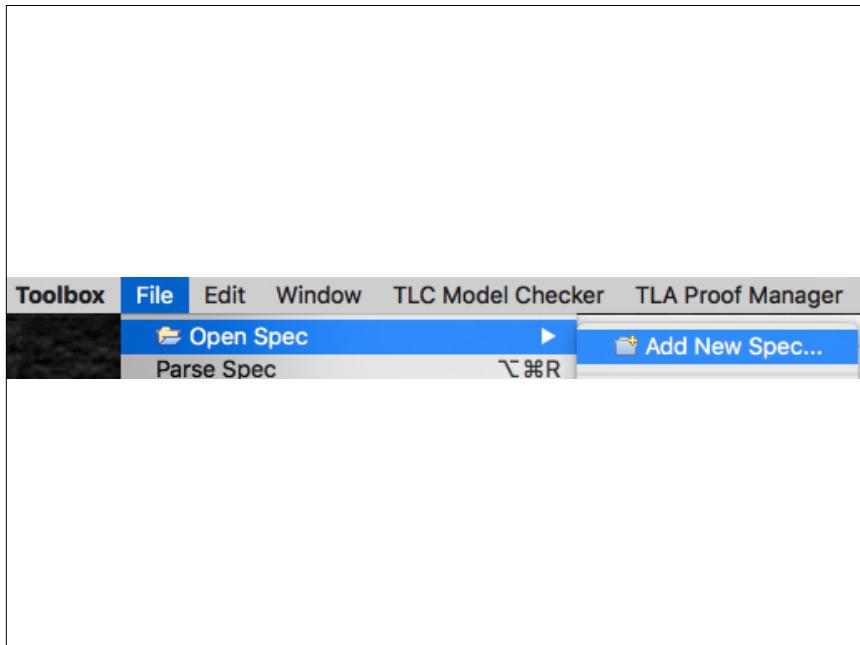
149-5

SmallToBig

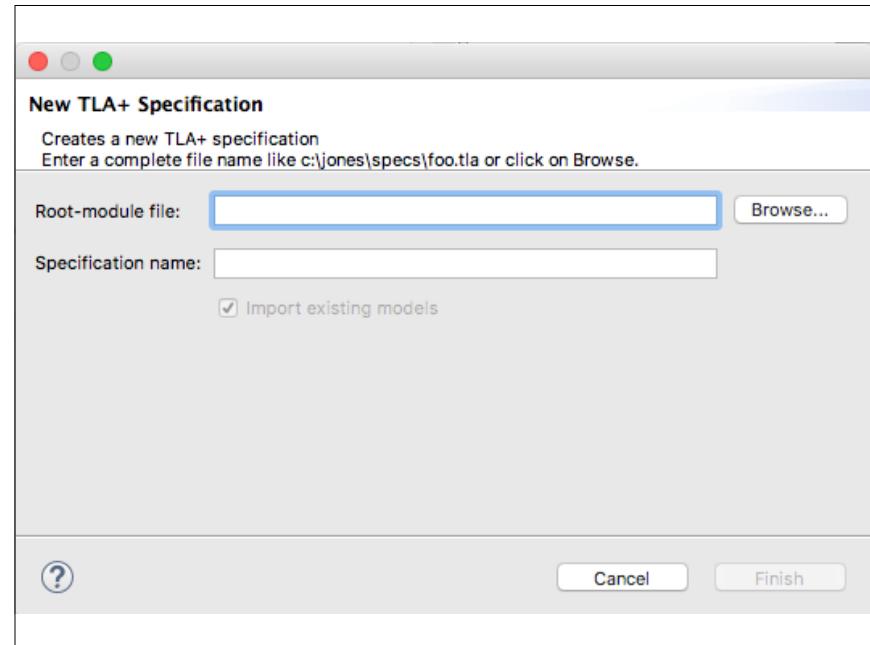
Putting it all together

```
SmallToBig == IF big + small < 5
    THEN /\ big' = big + small
        /\ small' = 0
    ELSE /\ small' = small - (5 - big)
        /\ big' = 5
```

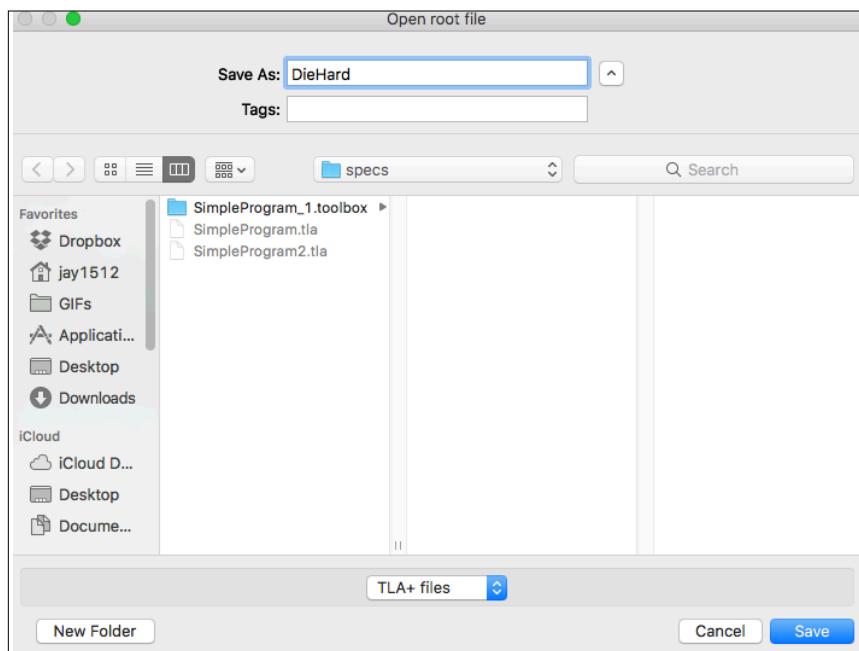
149-6



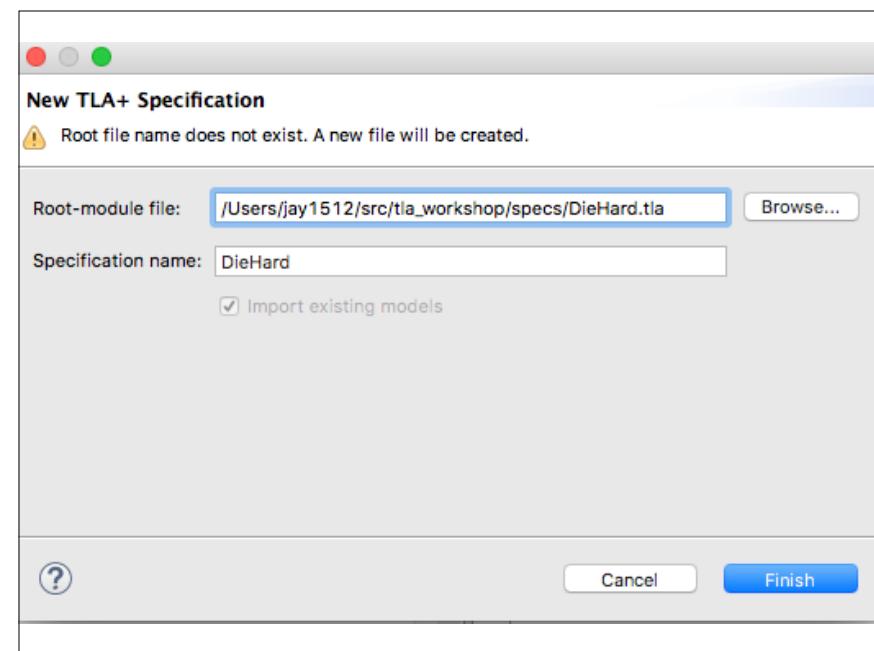
150



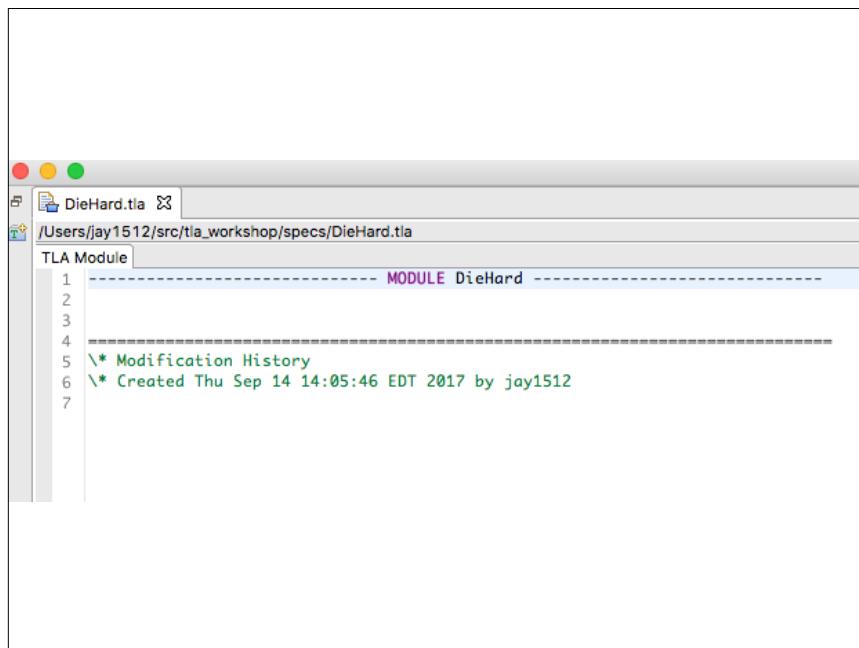
151



152



153



154

```
----- MODULE DieHard -----
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 ==

EmptySmall ==  $\wedge$  small' = 0
 $\wedge$  big' = big

EmptyBig ==

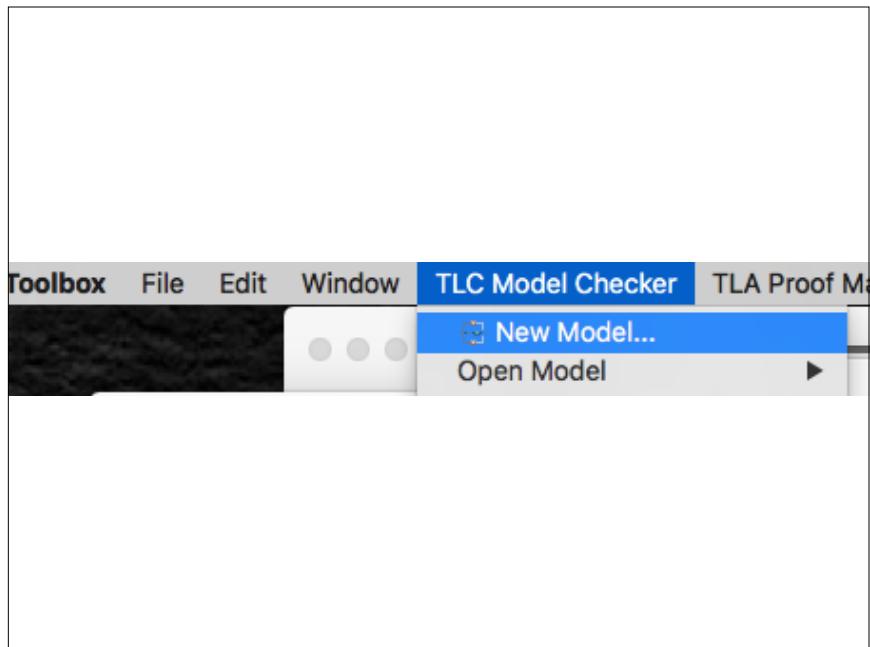
SmallToBig == IF big + small  $\leq$  5
THEN  $\wedge$  big' = big + small
 $\wedge$  small' = 0
ELSE  $\wedge$  big' = 5
 $\wedge$  small' = small - (5 - big)

BigToSmall ==

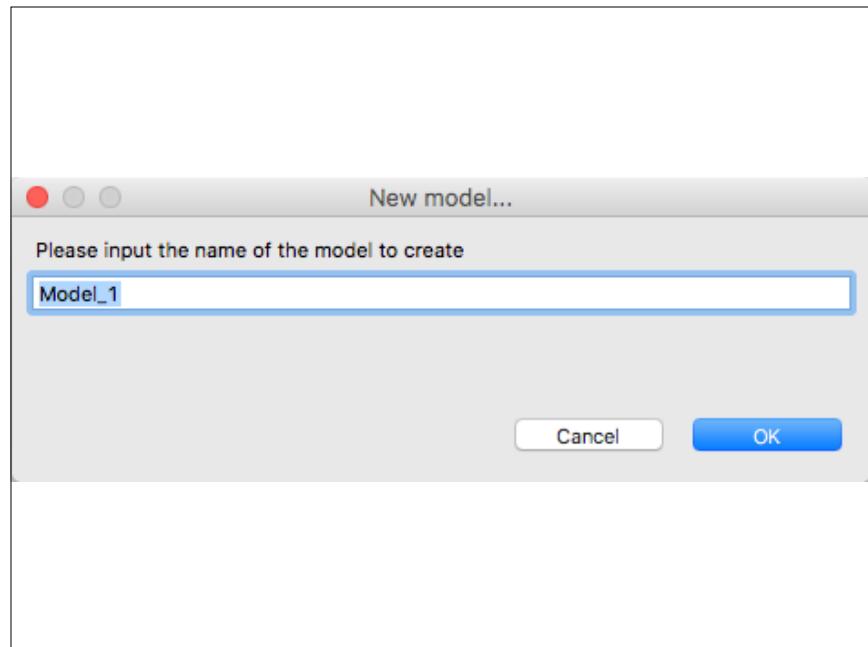
Next ==  $\vee$  FillSmall
 $\vee$  FillBig
 $\vee$  EmptySmall
 $\vee$  EmptyBig
 $\vee$  SmallToBig
 $\vee$  BigToSmall
```

specs/DieHard_incomplete.tla

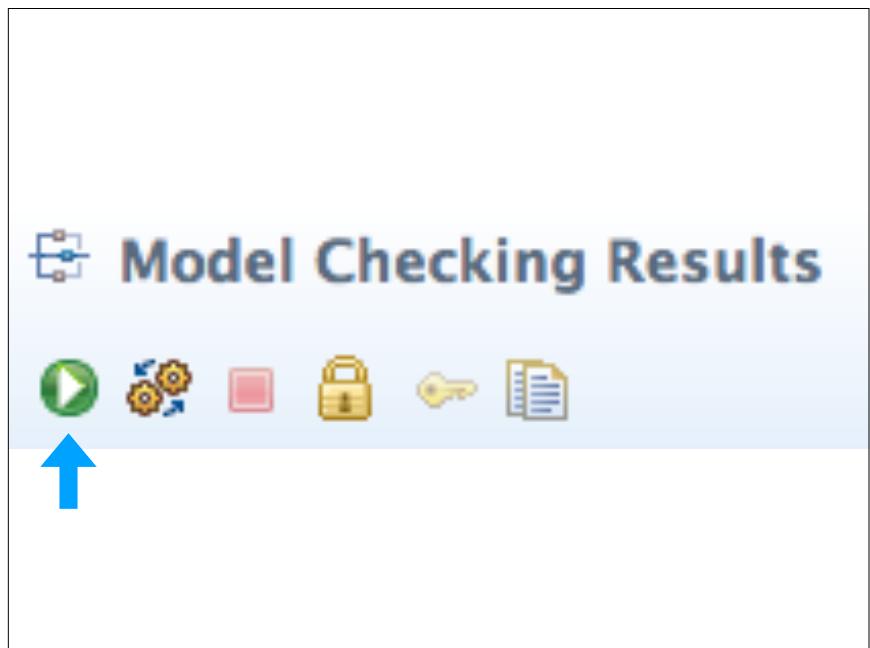
155



156



157



158

The screenshot shows the 'Model Checking Results' interface with a 'Statistics' section. It displays state space progress: Time 2017-09-14 14:13:..., Diameter 10, States Found 97, Distinct States 16, and Queue Size 0. To the right, a table shows coverage details at 2017-09-14 14:13:33. The table has columns for Module, Location, and Count. The data is as follows:

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

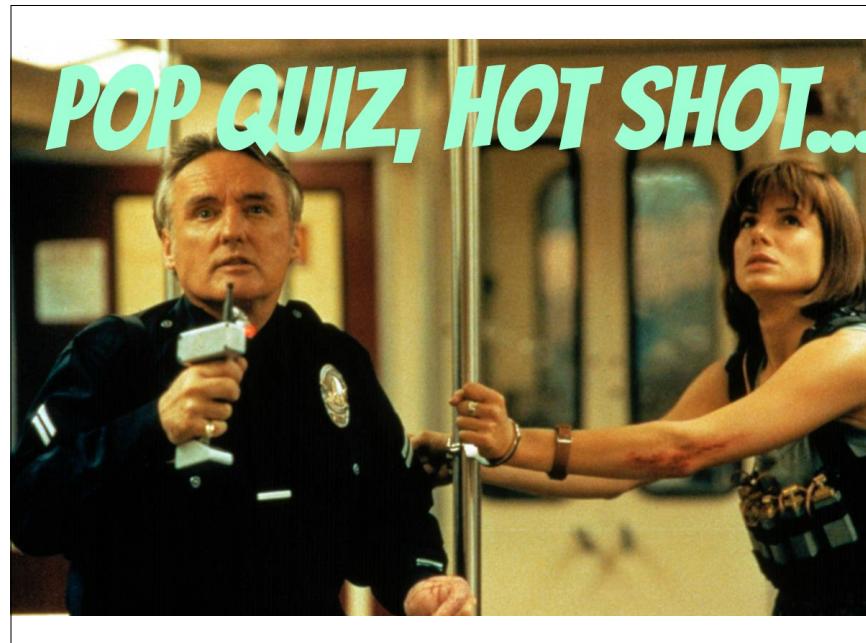
159-1

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		

159-2

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		

159-3



```

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

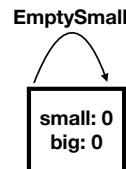
Next ==  $\vee$  FillSmall
        $\vee$  FillBig
        $\vee$  EmptySmall
        $\vee$  EmptyBig
        $\vee$  SmallToBig
        $\vee$  BigToSmall
  
```

160

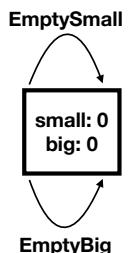
161



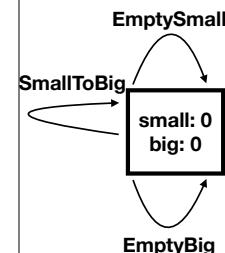
162-1



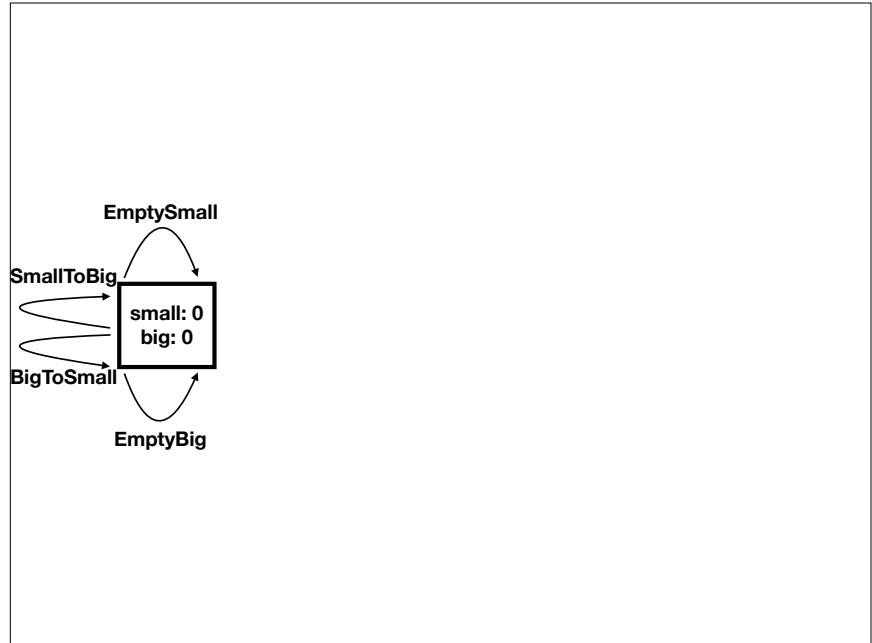
162-2



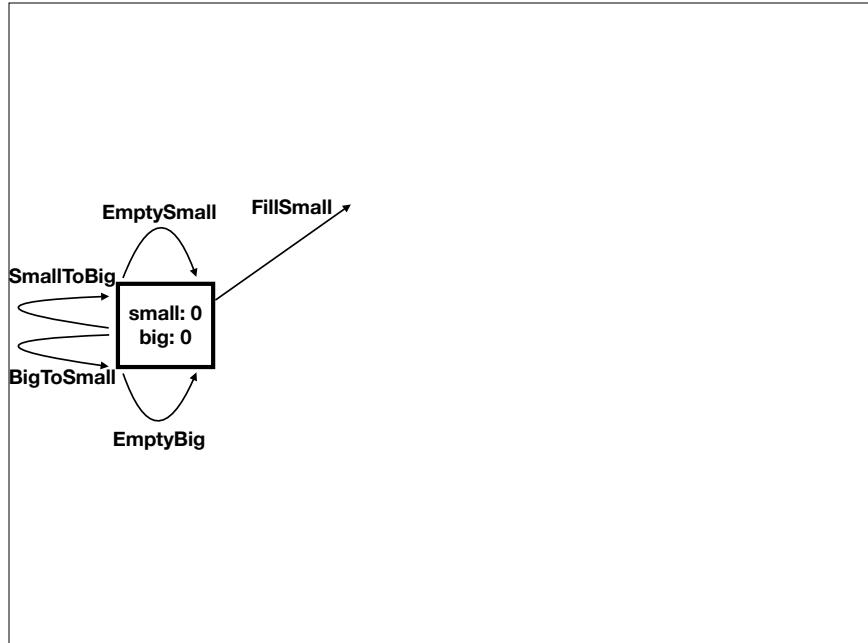
162-3



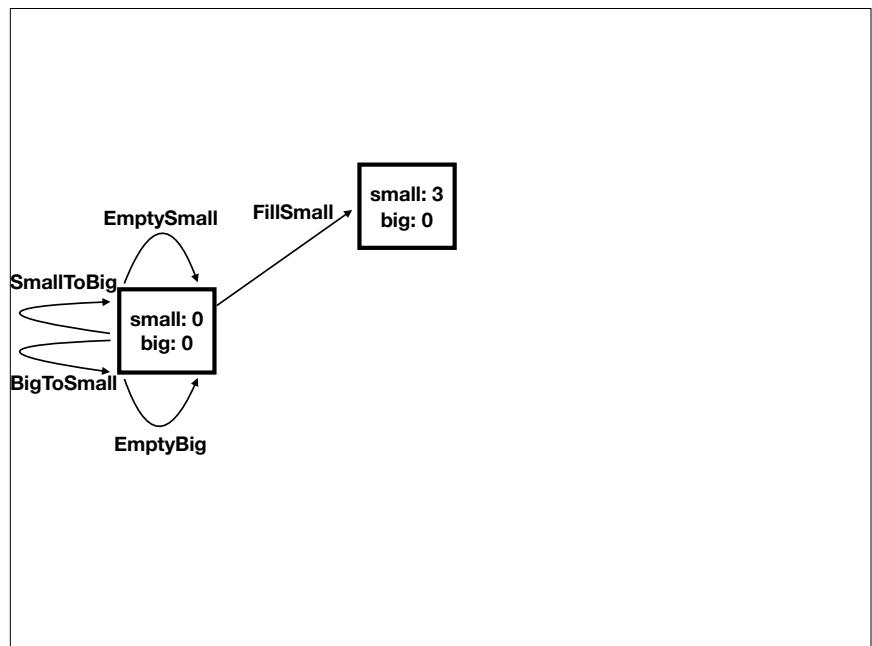
162-4



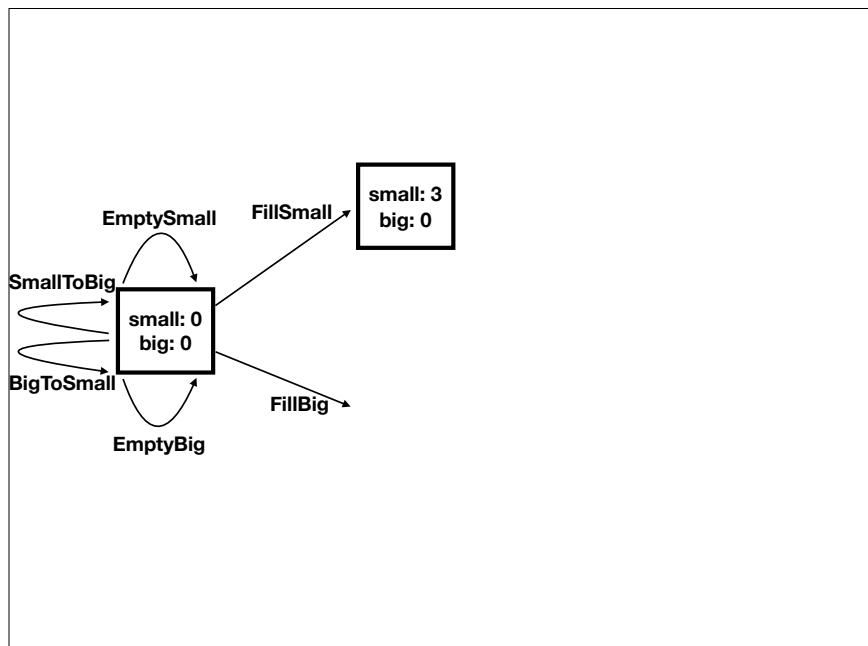
162-5



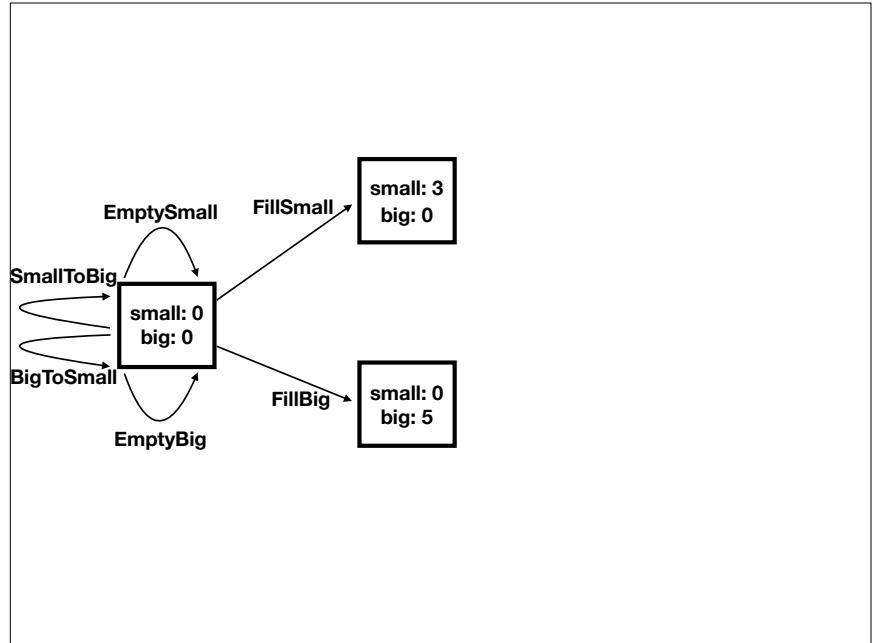
162-6



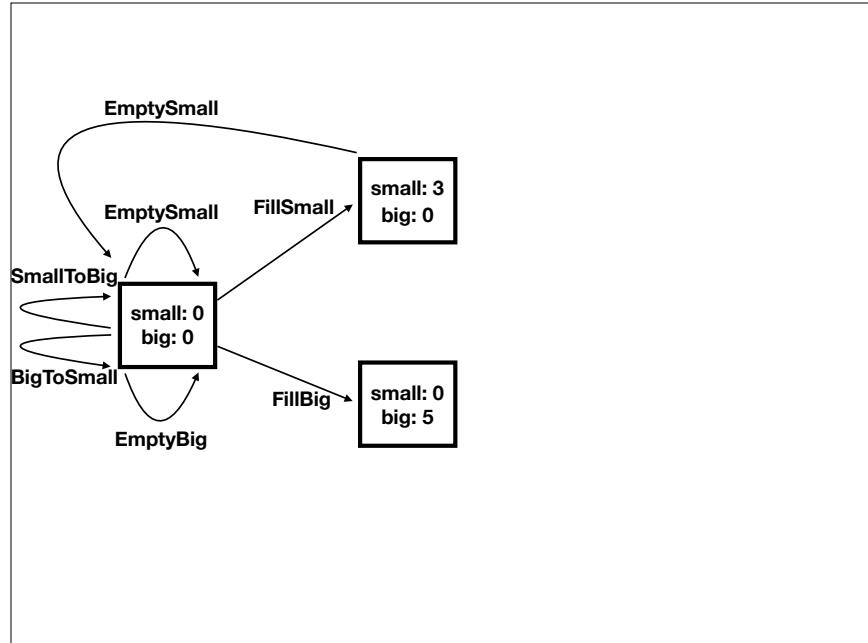
162-7



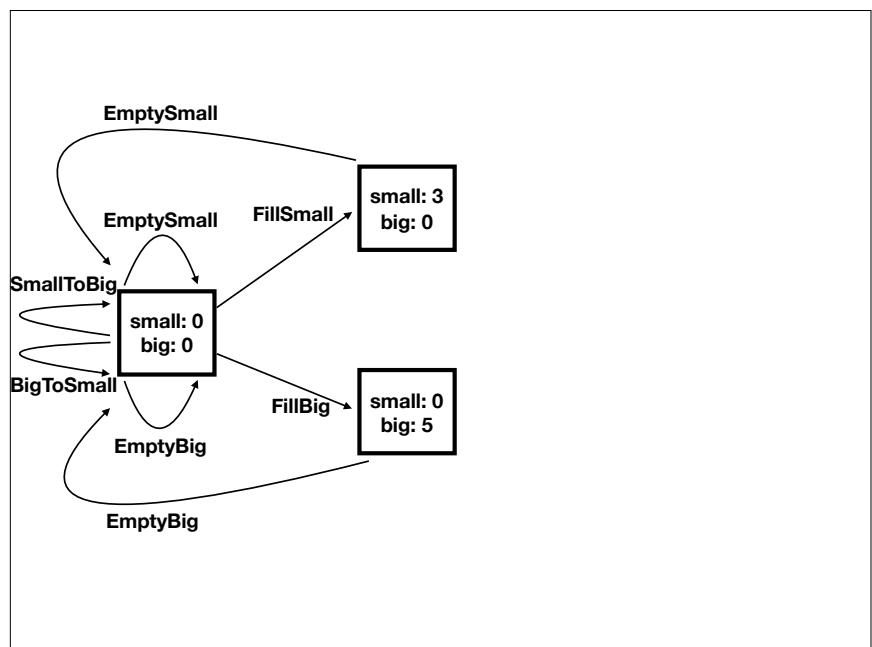
162-8



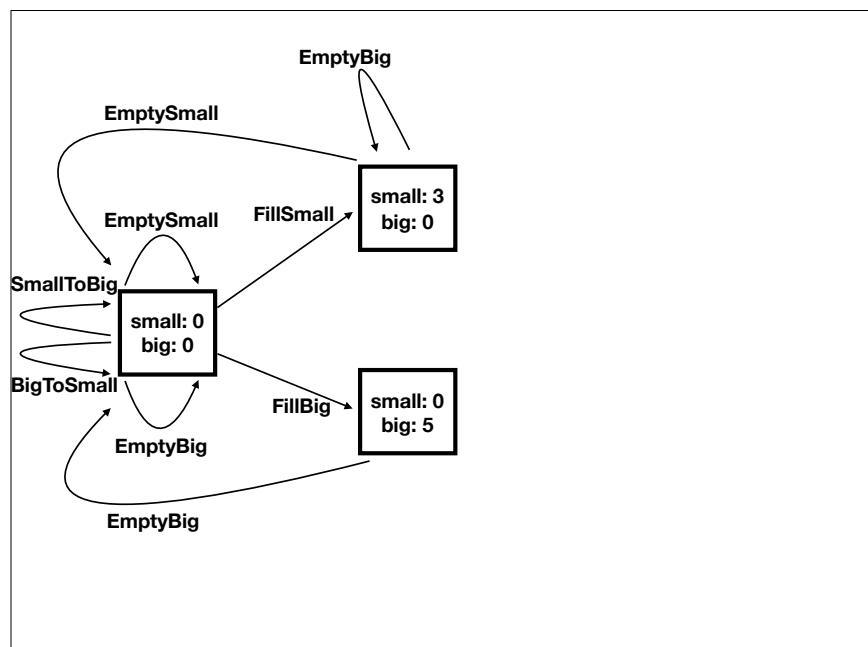
162-9



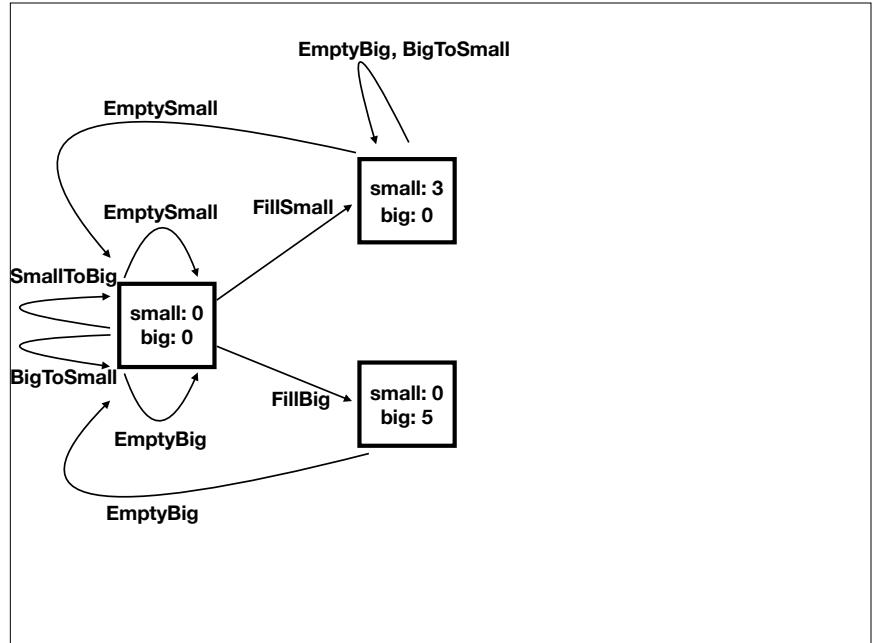
162-10



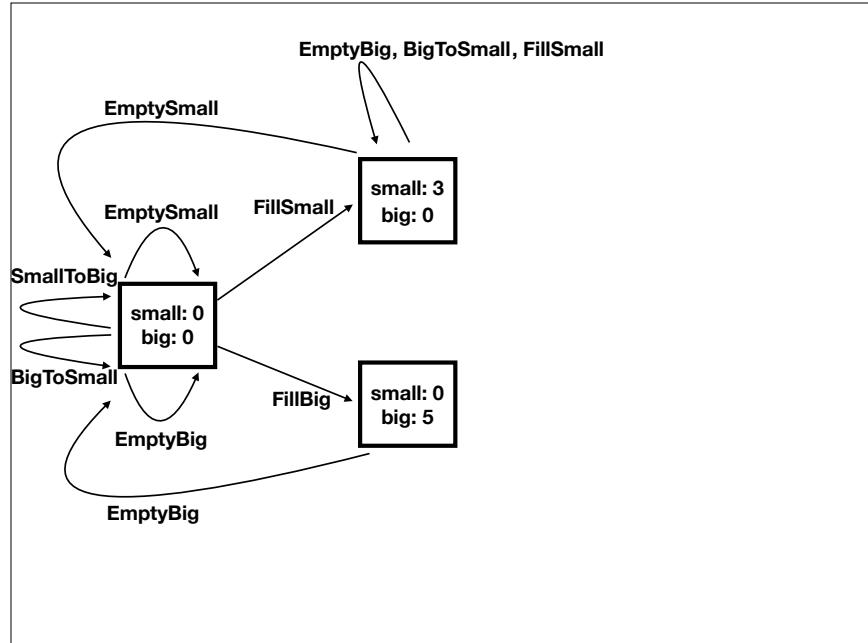
162-11



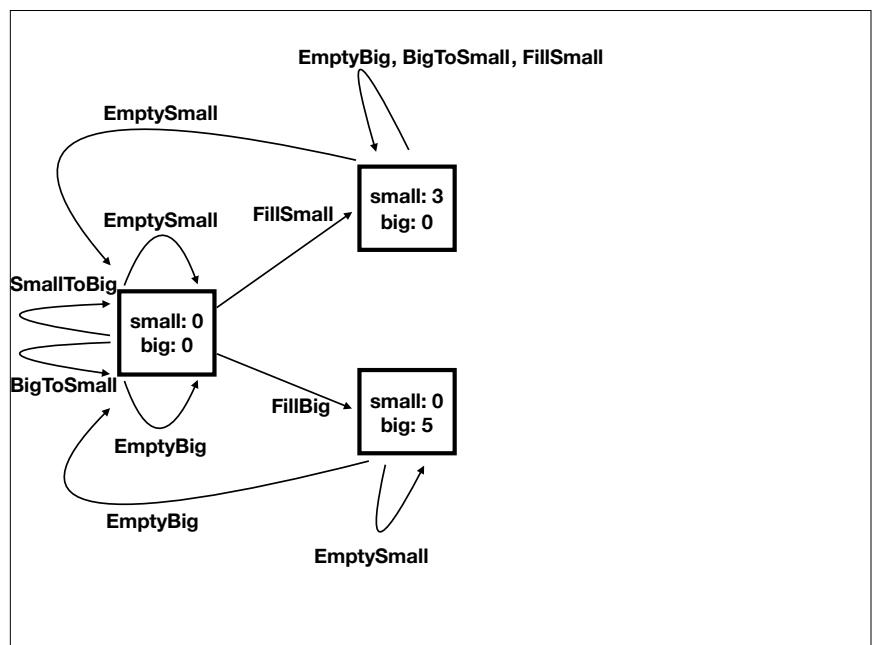
162-12



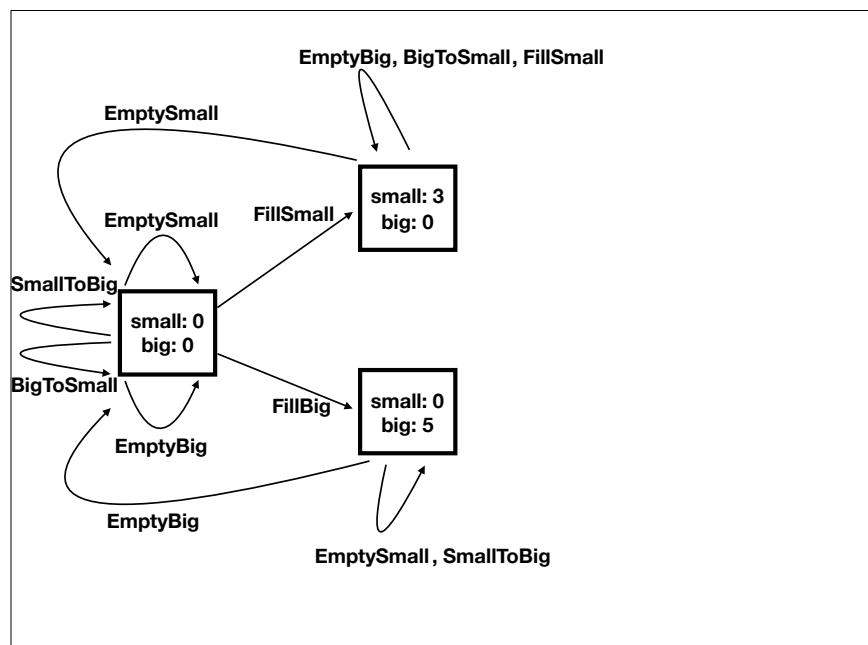
162-13



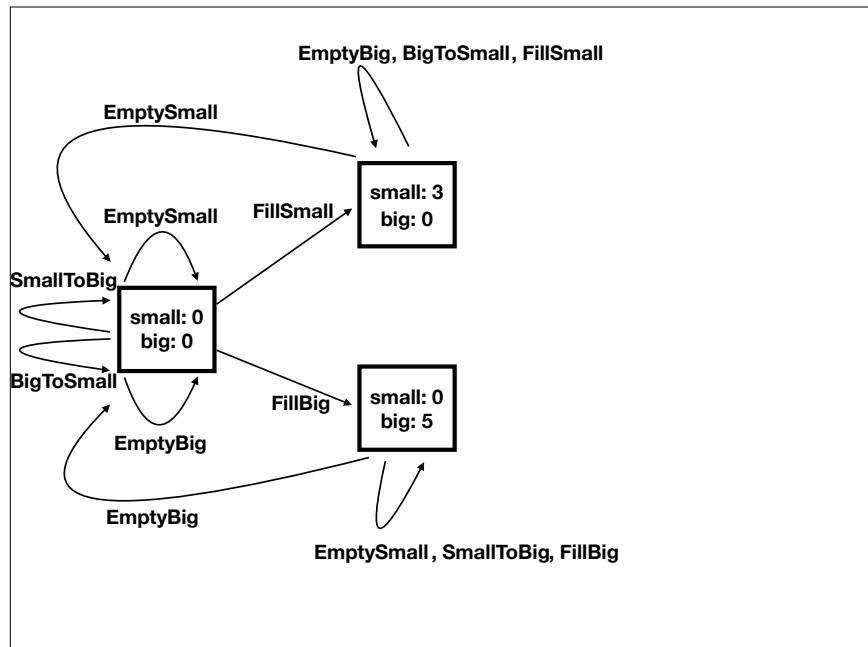
162-14



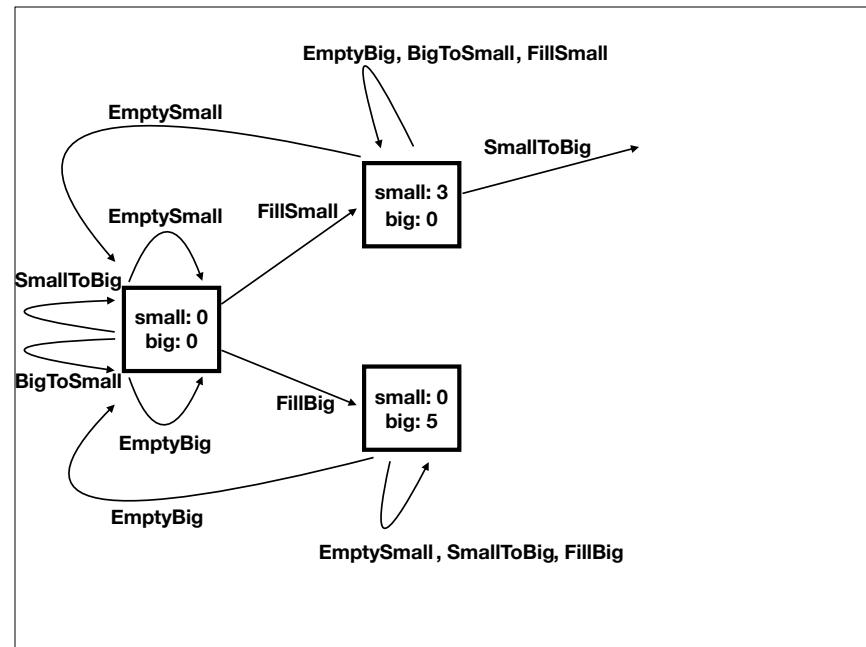
162-15



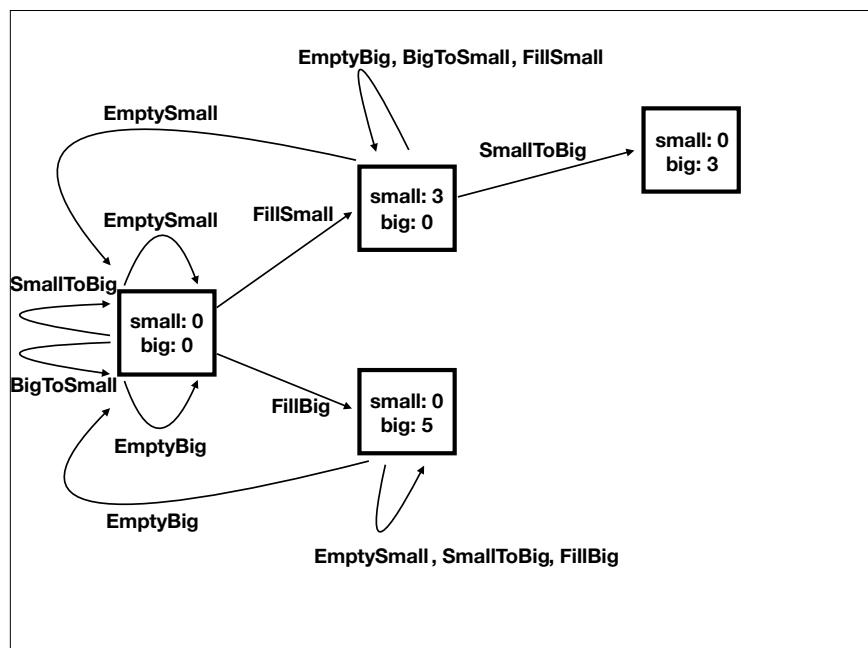
162-16



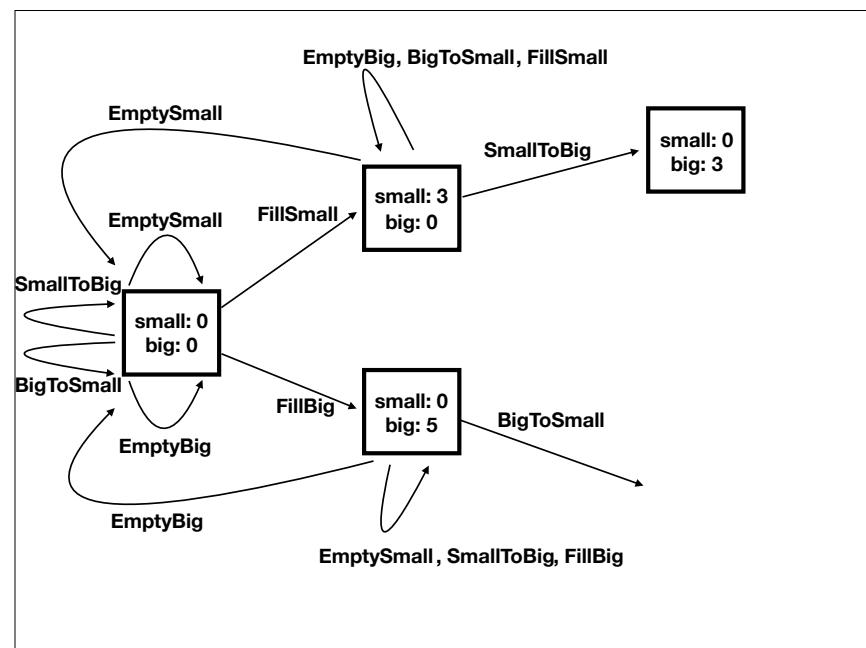
162-17



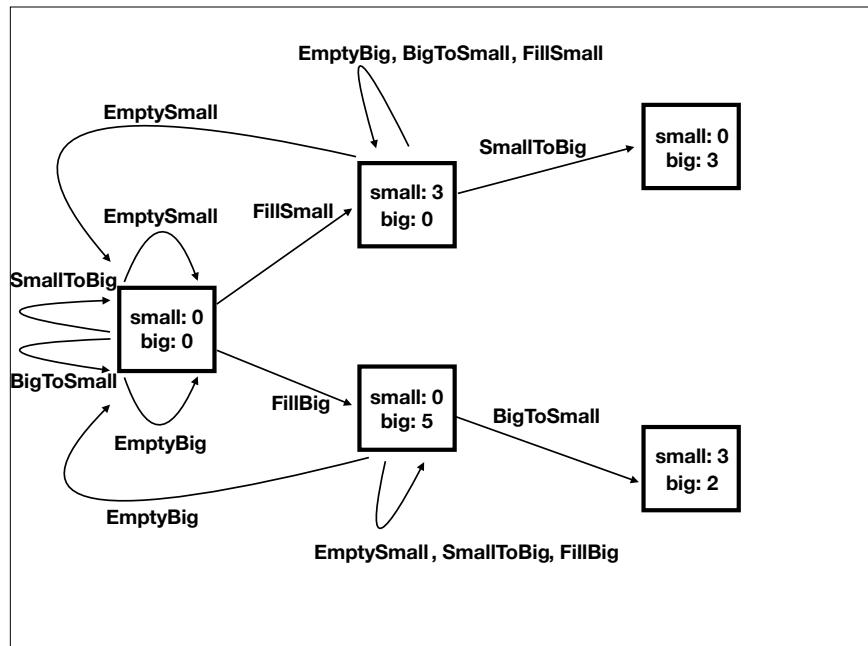
162-18



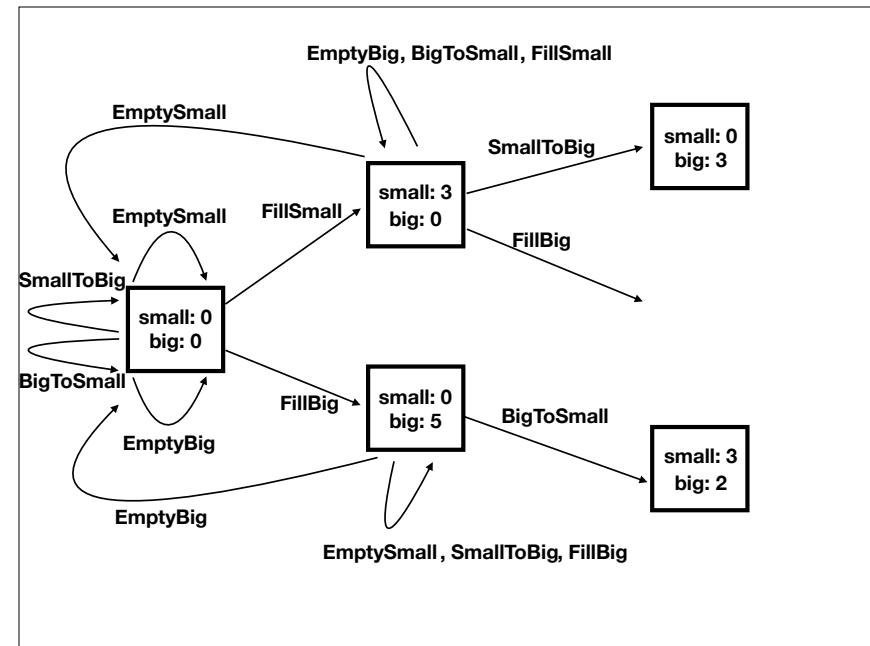
162-19



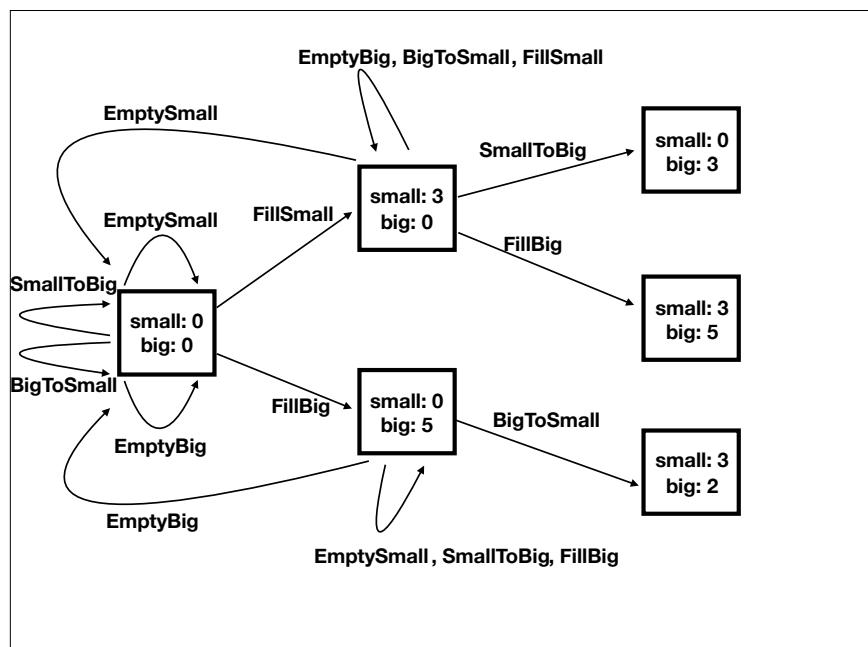
162-20



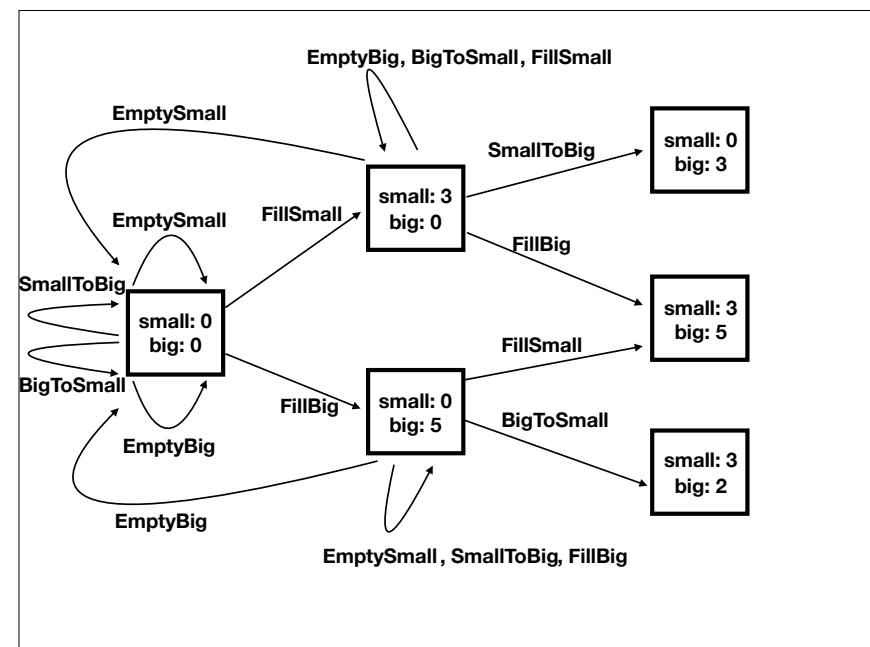
162-21



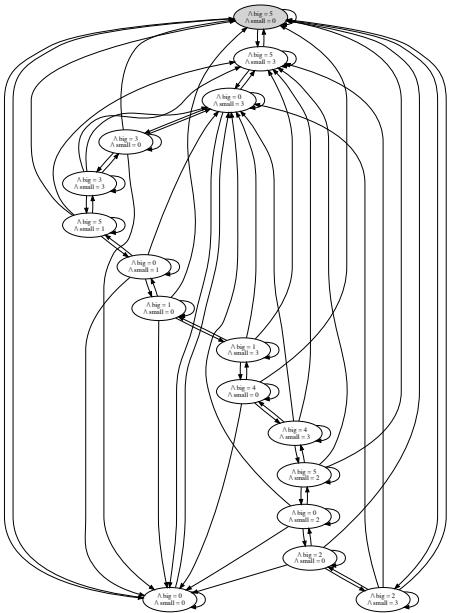
162-22



162-23



162-24



163

Die Hard

Die Hard

- Given the restrictions determined by our formula, TLC has found a total of 16 **unique** states.

164-1

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**.

164-2

164-3

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?

164-4

Invariants

165-1

Invariants

- It's time to explore the true power of TLA+ and TLC

165-2

Invariants

- It's time to explore the true power of TLA+ and TLC
- An invariant is “**something that should always be true**”

165-3

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**”

165-4

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

165-5

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

165-6

What to check?

Deadlock

Invariants

Formulas true in every reachable state.

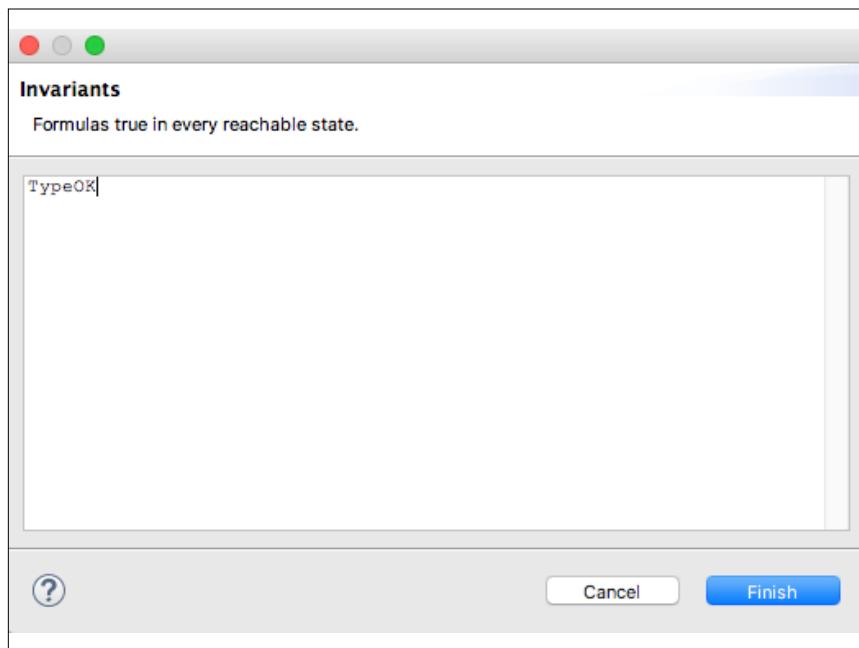
Add

Edit

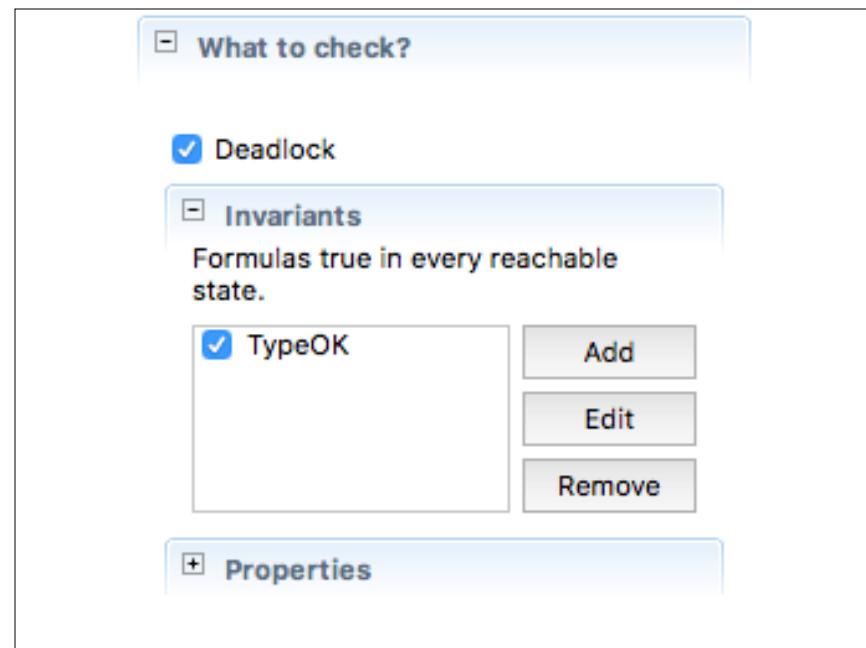
Remove

Properties

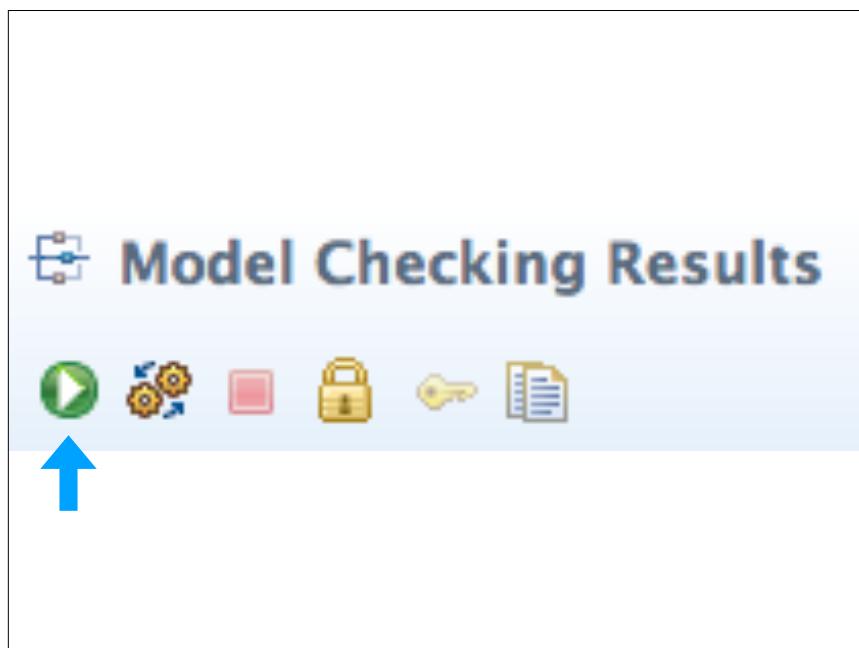
166



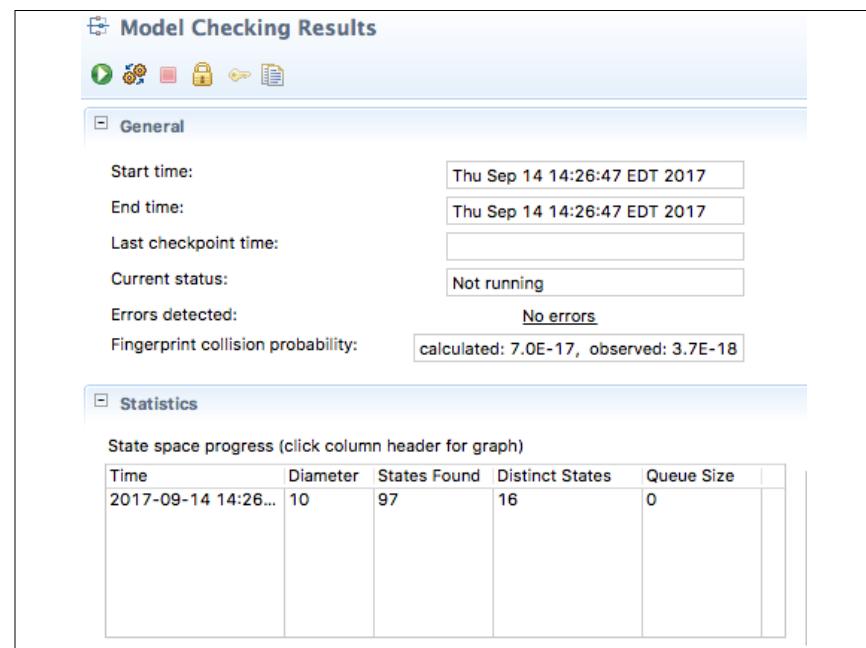
167



168



169



170

```
Init ==  $\wedge$  big = 6  
       $\wedge$  small = 0
```

171



172

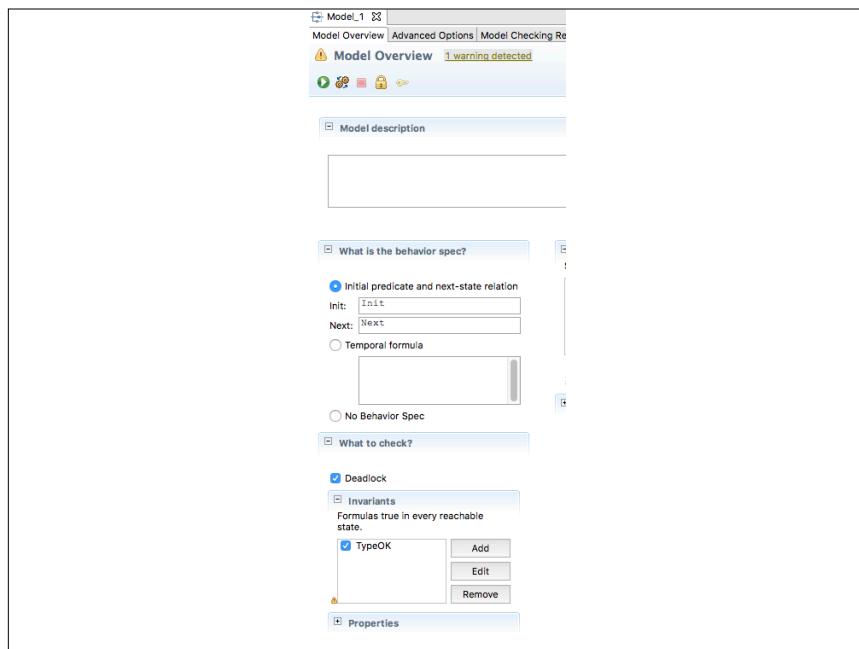
TypeOK

```
TypeOK ==  $\wedge$  small  $\in$  0..3  
       $\wedge$  big  $\in$  0..5
```

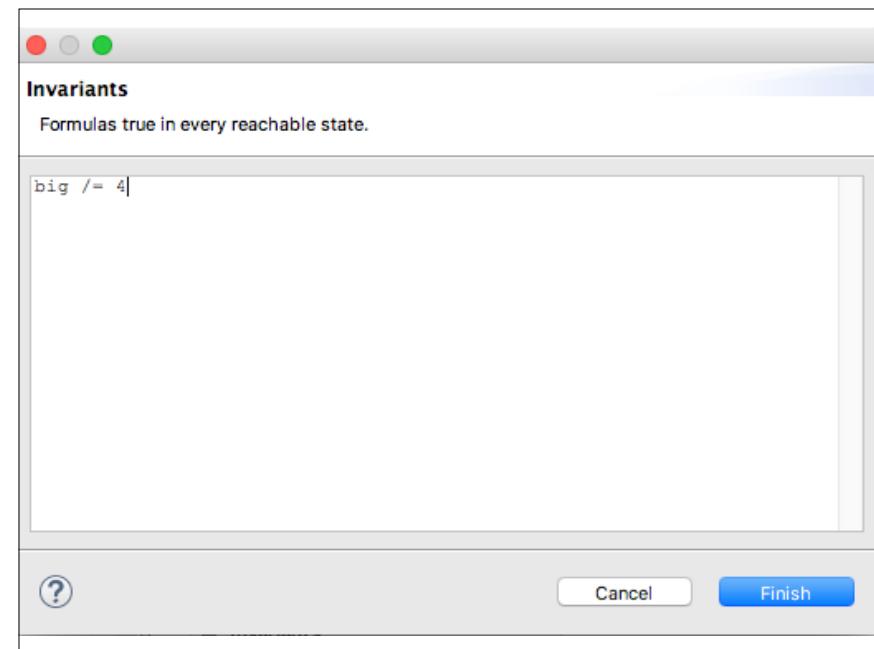
173

```
Init ==  $\wedge$  big = 5  
       $\wedge$  small = 0
```

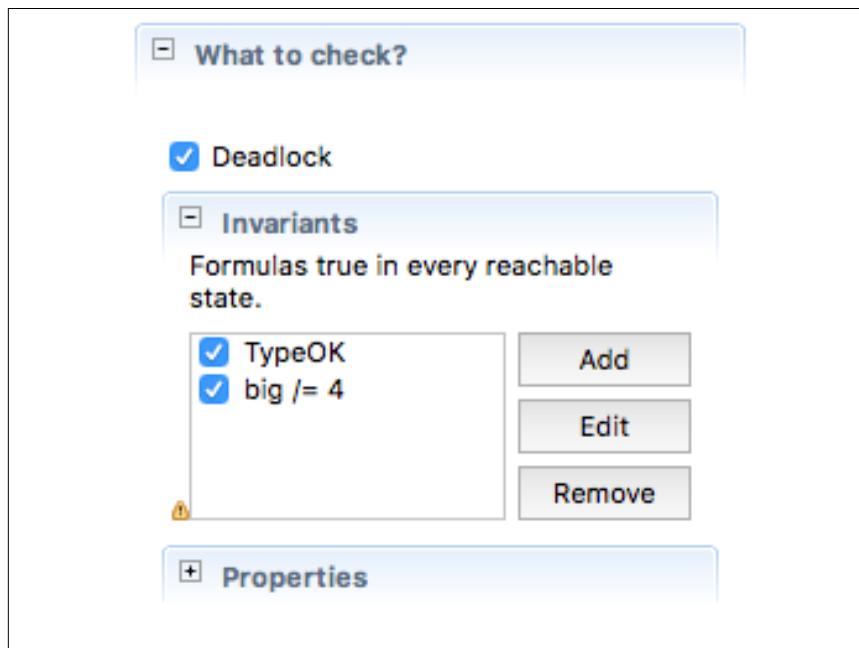
174



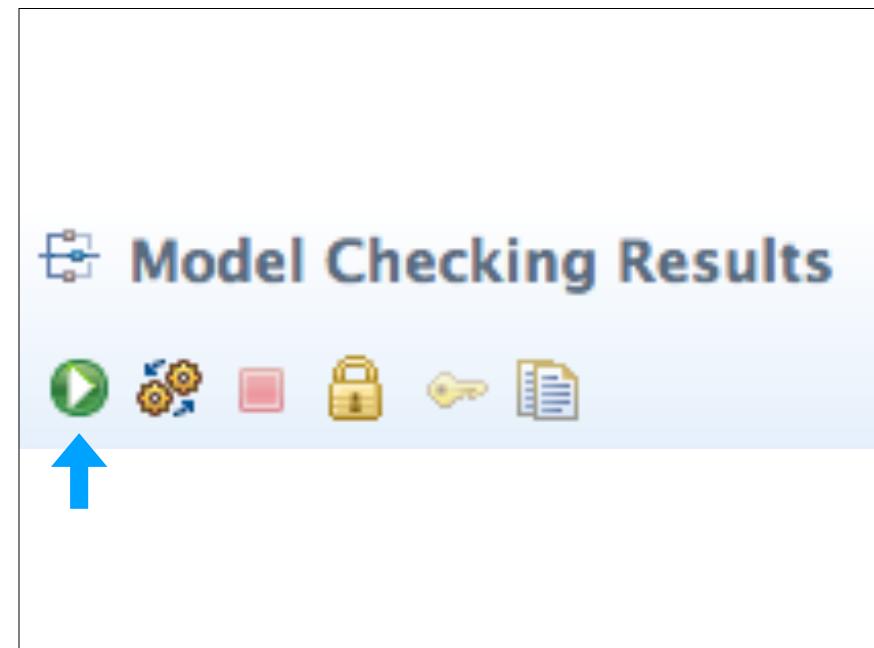
175



176



177



178

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

179

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

180-1

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

180-2

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

180-3

Woah

181-1

Woah

- This is amazingly powerful.

181-2

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.

181-3

Woah

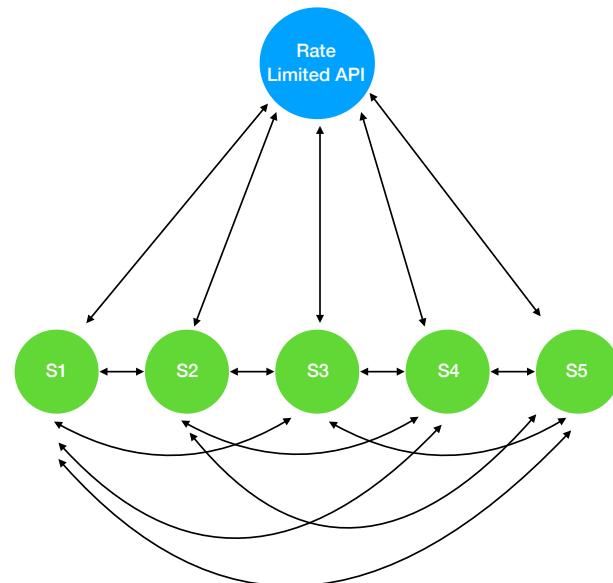
- This is amazingly powerful.
- TLC tries every single possible combination of states (i.e. behaviour/execution), and checks invariants for every state in every behaviour.
- The entire state history is kept, providing the best “stack trace” you’ve ever seen.

181-4

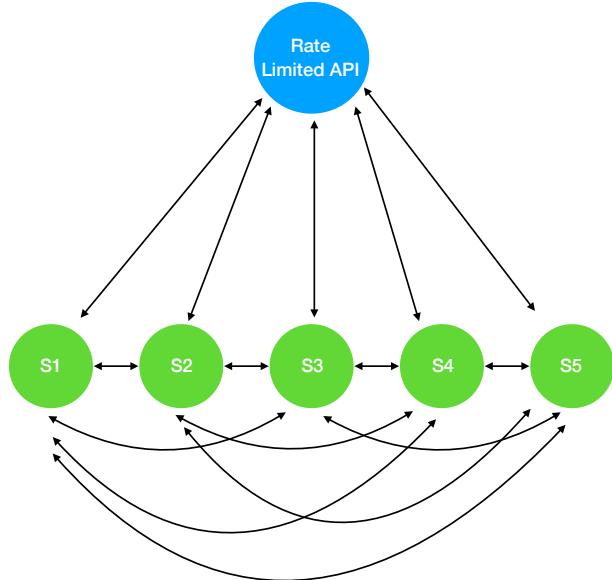
Woah

- 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.

181-5



182-1



182-2

AWS Fault-Tolerant Replication Algorithm

183-1

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.

183-2

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.

183-4

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.

183-3

Transaction Commit

184-1

Transaction Commit

- Consider a replicated database.

184-2

Transaction Commit

- Consider a replicated database.
- Multiple “resource managers,” each carrying a full replica.

184-3

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.

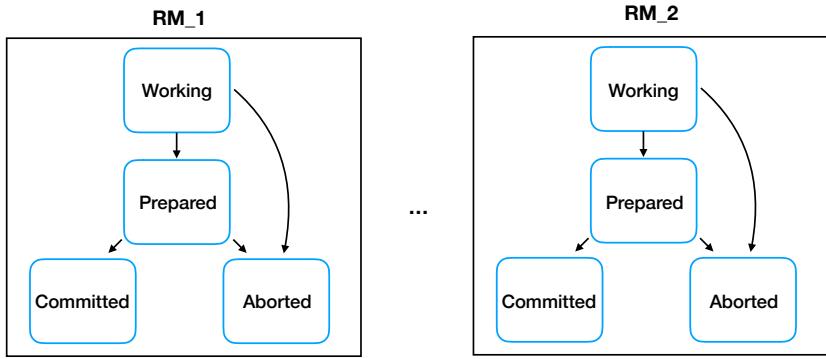
184-4

Resource Managers



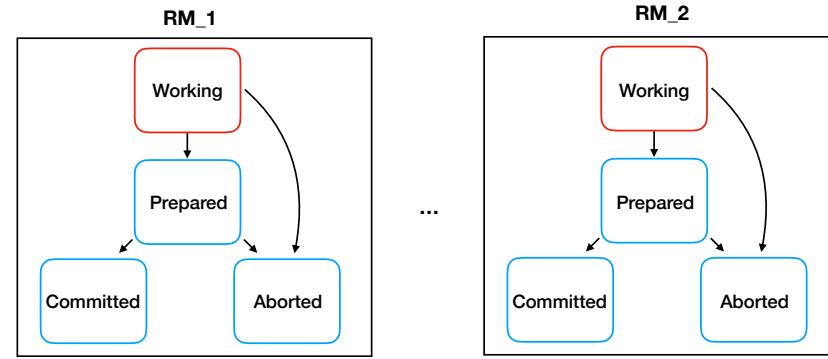
185

Resource Managers



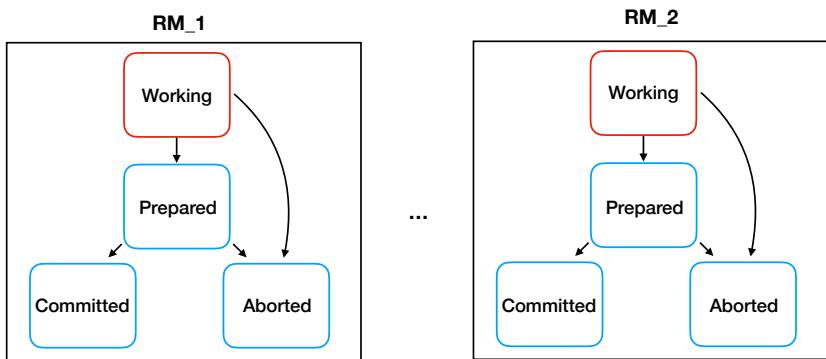
186

Resource Managers



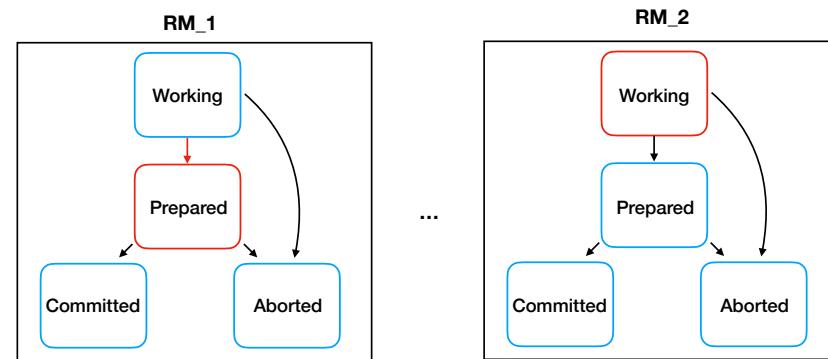
187

Resource Managers



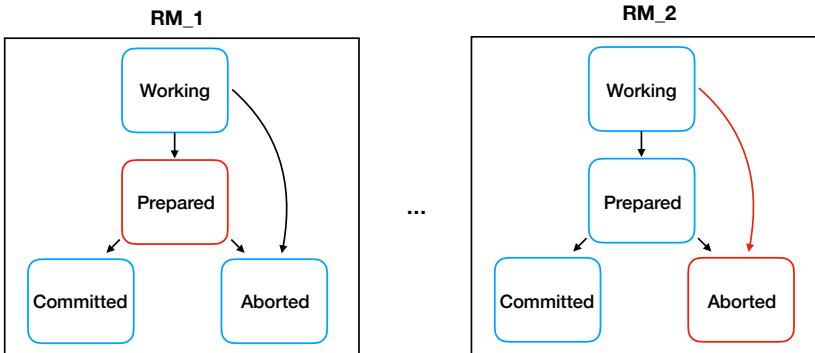
188

Resource Managers



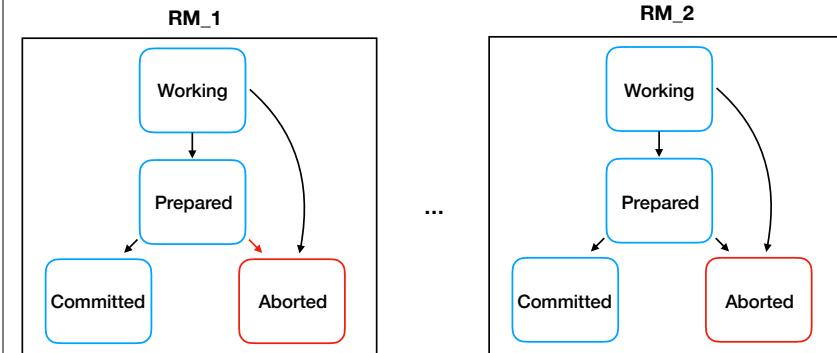
189

Resource Managers



190

Resource Managers



191

Resource Managers

```
rmState["rm1": "working"  
rmState["rm2": "working"]
```

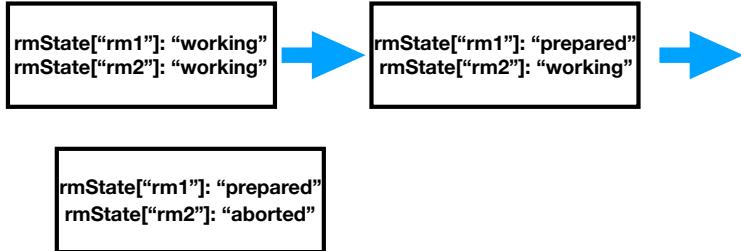
192-1

Resource Managers

```
rmState["rm1": "working"  
rmState["rm2": "working"] → rmState["rm1": "prepared"  
rmState["rm2": "working"]
```

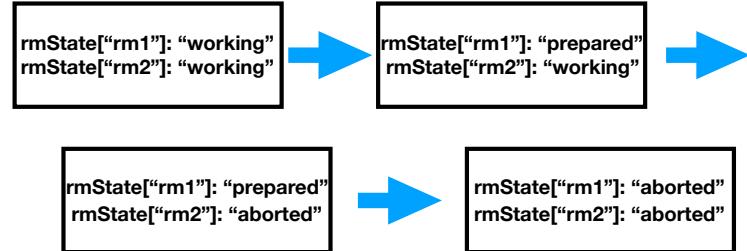
192-2

Resource Managers



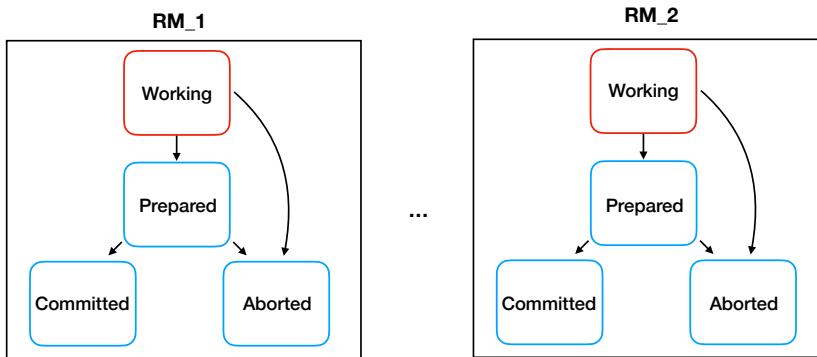
192-3

Resource Managers



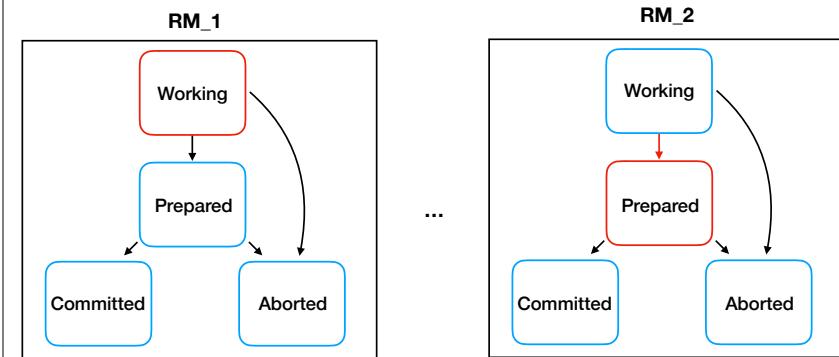
192-4

Resource Managers



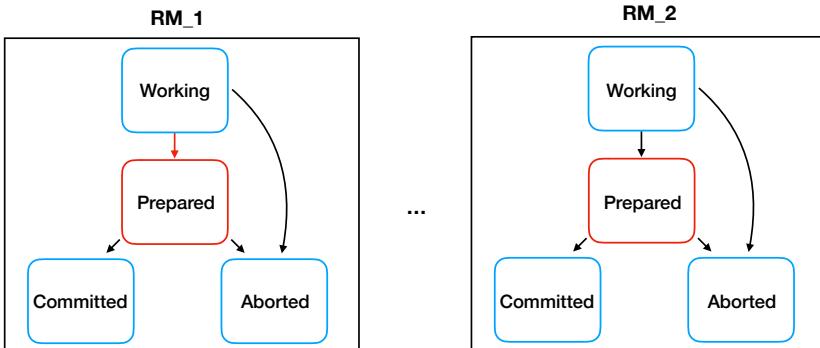
193

Resource Managers



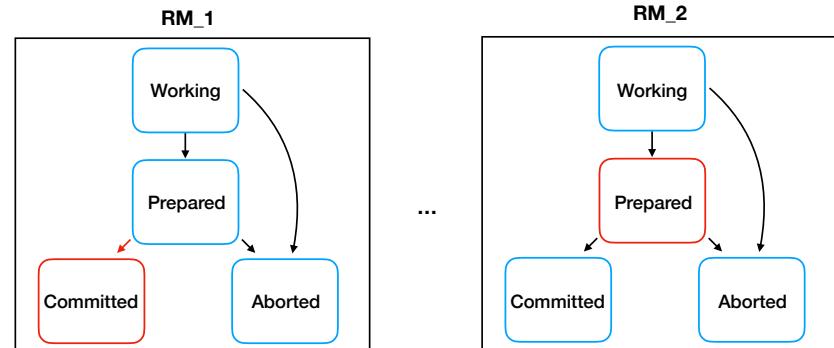
194

Resource Managers



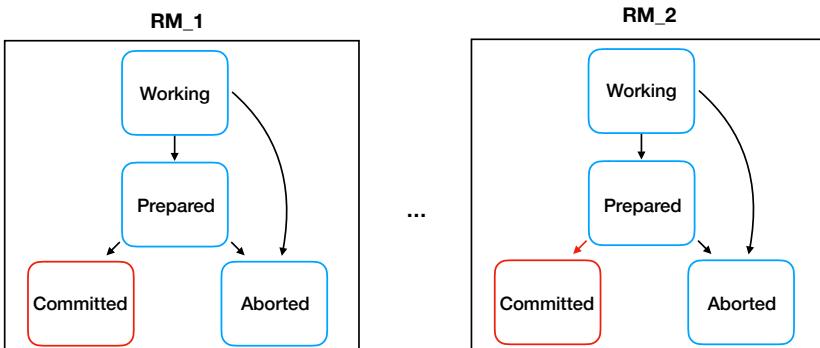
195

Resource Managers



196

Resource Managers



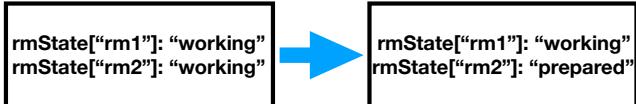
197

Resource Managers

```
rmState["rm1": "working"]
rmState["rm2": "working"]
```

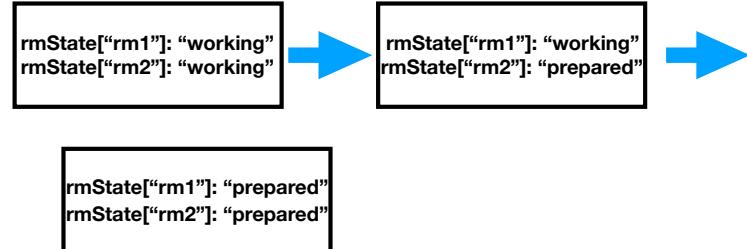
198-1

Resource Managers



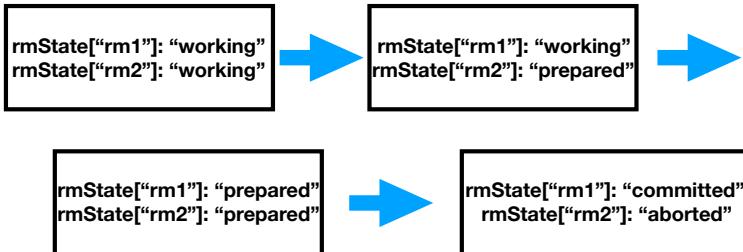
198-2

Resource Managers



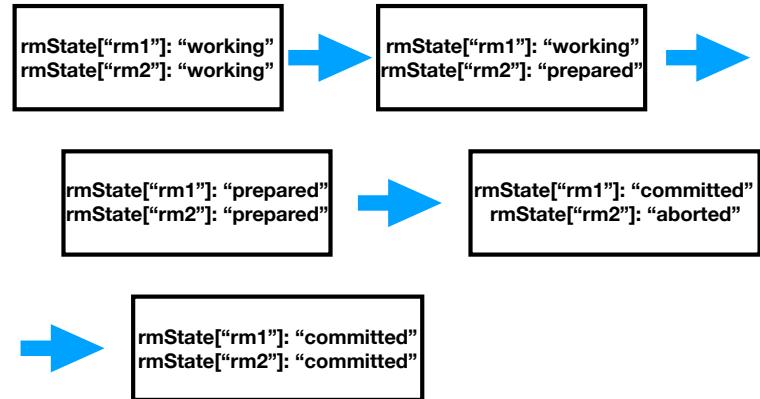
198-3

Resource Managers



198-4

Resource Managers



198-5

Resource Managers

199-1

Resource Managers

- We just looked at two possible behaviours.

199-2

Resource Managers

- We just looked at two possible behaviours.
- How many in total are possible?

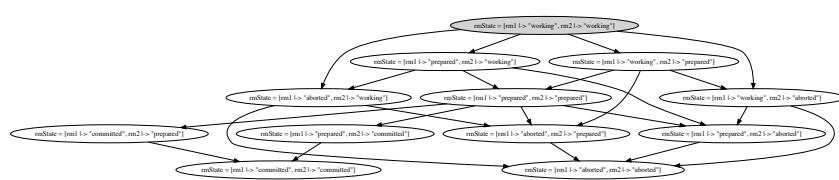
199-3

Resource Managers

- We just looked at two possible behaviours.
- How many in total are possible?
- Lots and lots! And with ever additional Resource Manager added, it grows and grows.

199-4

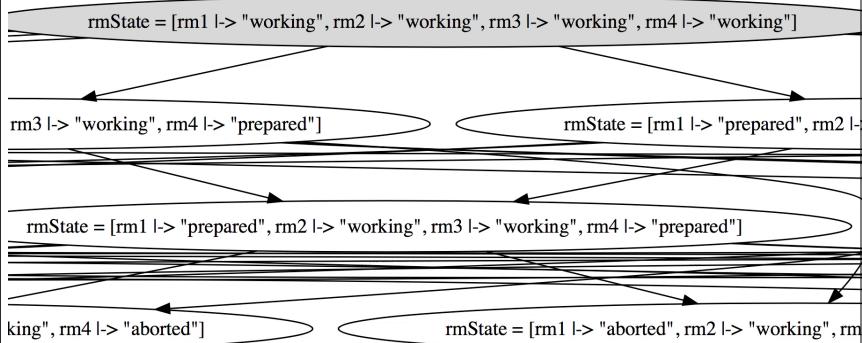
2 Resource Managers



12 Distinct States

200

4 Resource Managers



96 Distinct States

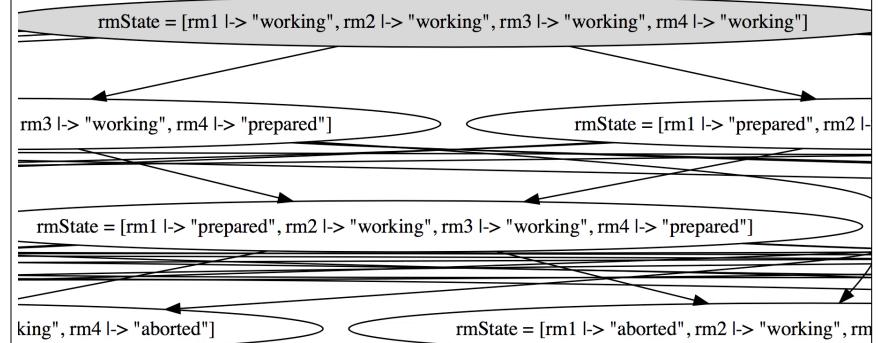
3 Resource Managers



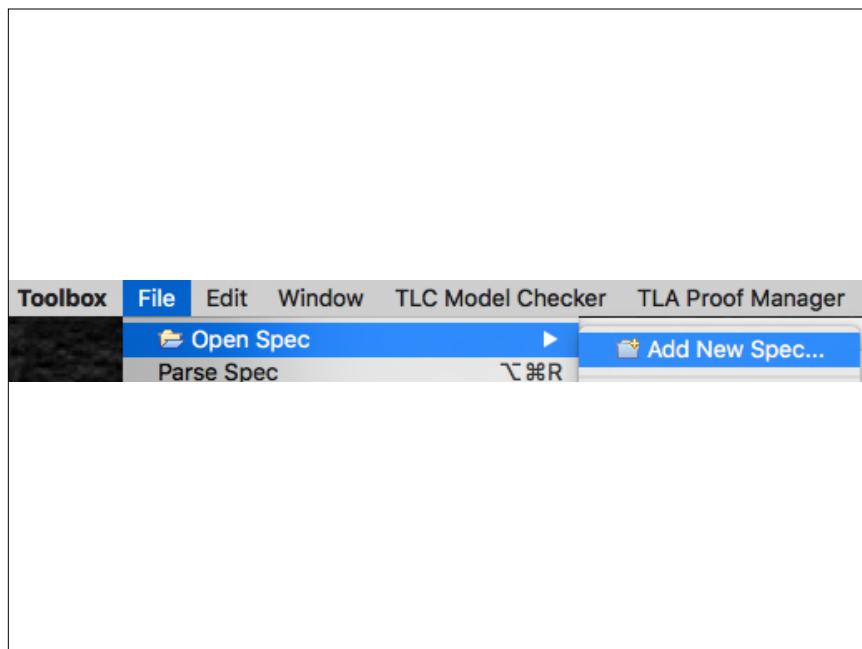
34 Distinct States

201

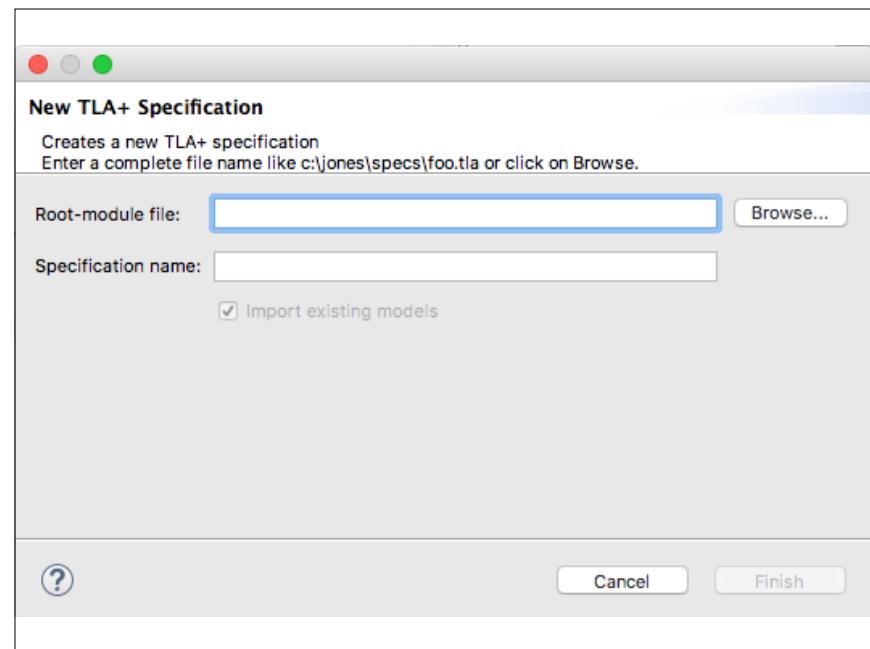
4 Resource Managers



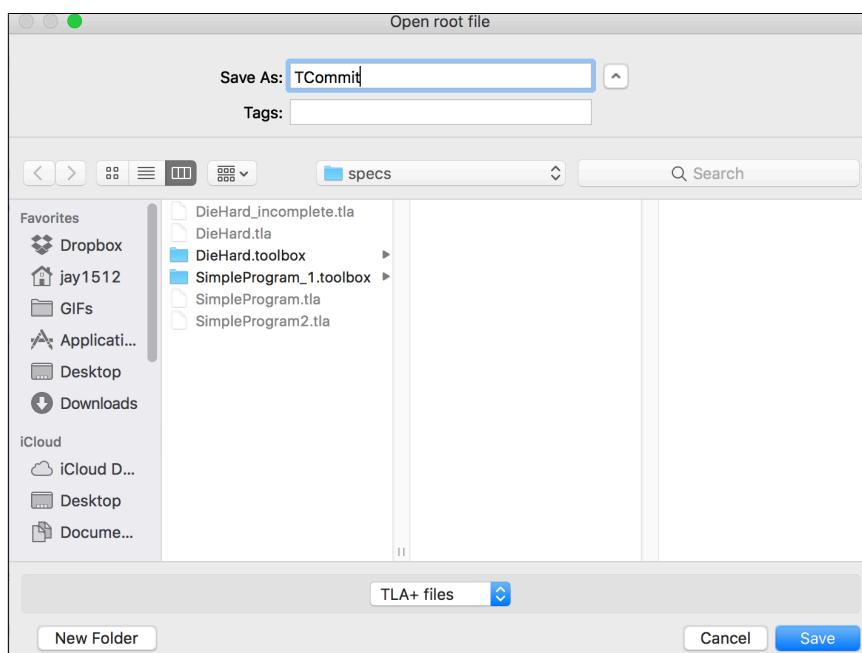
96 Distinct States



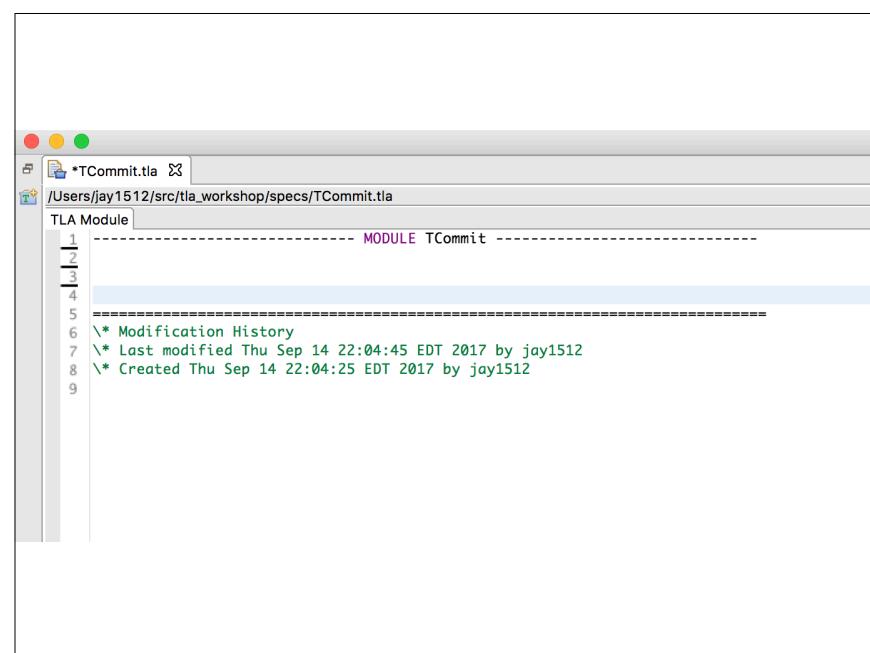
203



204



205



206

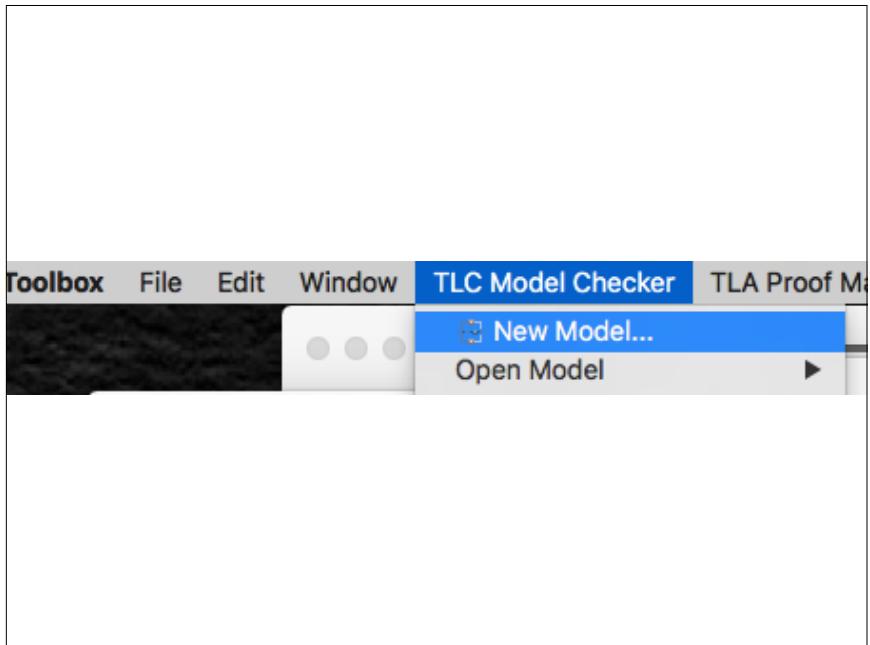
specs/TCommit.tla

207

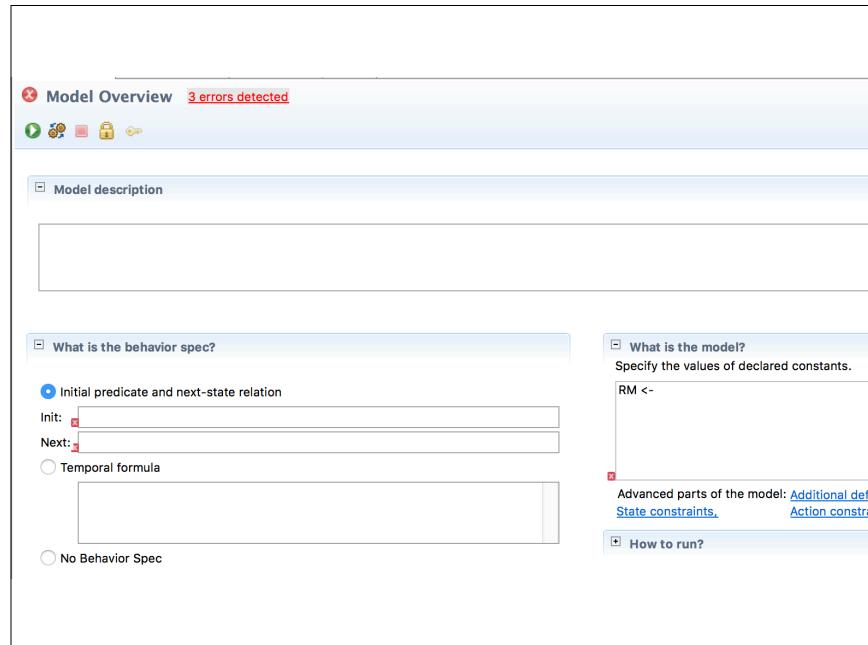
```
TCCommit.tla ┌───────────────── MODULE TCommit ──────────────────┐
1 (*-----*)
2 (* This specification is explained in "Transaction Commit", Lecture 5 of *)
3 (* the TLA Video Course. *)
4 (*-----*)
5 (*-----*)
6 CONSTANT RM      \* The set of participating resource managers
7
8 VARIABLE rmState \* rmState[rm] is the state of resource manager r.
9 -----
10 TCTypeOK == 
11   (*-----*)
12   (* type-checking function *)
13   (*-----*)
14 rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
15
16 TCInit == rmState = [r \in RM !-> "working"]
17 (*-----*)
18 (* The initial predicate *)
19 (*-----*)
20
21 conCommit == \A r \in RM : rmState[r] \in {"prepared", "committed"}
22 (*-----*)
23 (* True iff all RMs are in the "prepared" or "committed" state. *)
24 (*-----*)
25
26 notCommitted == \A r \in RM : rmState[r] # "committed"
27 (*-----*)
28 (* True iff no resource manager has decided to commit. *)
29 (*-----*)
30
31 (*-----*)
32 (* We now define the actions that may be performed by the RMs, and then *)
33 (* define the complete next-state action of the specification to be the *)
34 (* disjunction of the possible RM actions. *)
35 (*-----*)
36 Prepare(r) == \A rmState' : working
37   \wedge rmState' = [rmState EXCEPT !r = "prepared"]
38
39 Decide(r) == \V \A rmState[r] = "prepared"
40   \wedge conCommit
41   \wedge rmState' = [rmState EXCEPT !r = "committed"]
42   \wedge \A rmState'[r] \in {"working", "prepared"}
43   \wedge notCommitted
44   \wedge rmState' = [rmState EXCEPT !r = "aborted"]
45
46 TCNext == \E r \in RM : Prepare(r) \vee Decide(r)
47 (*-----*)
48 (* The next-state action *)
49 (*-----*)

-----
```

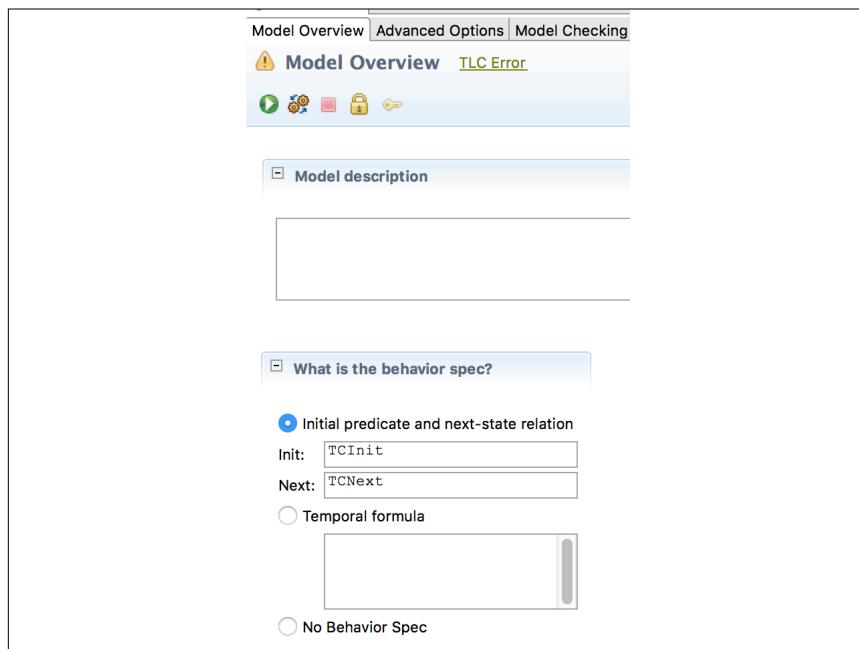
208



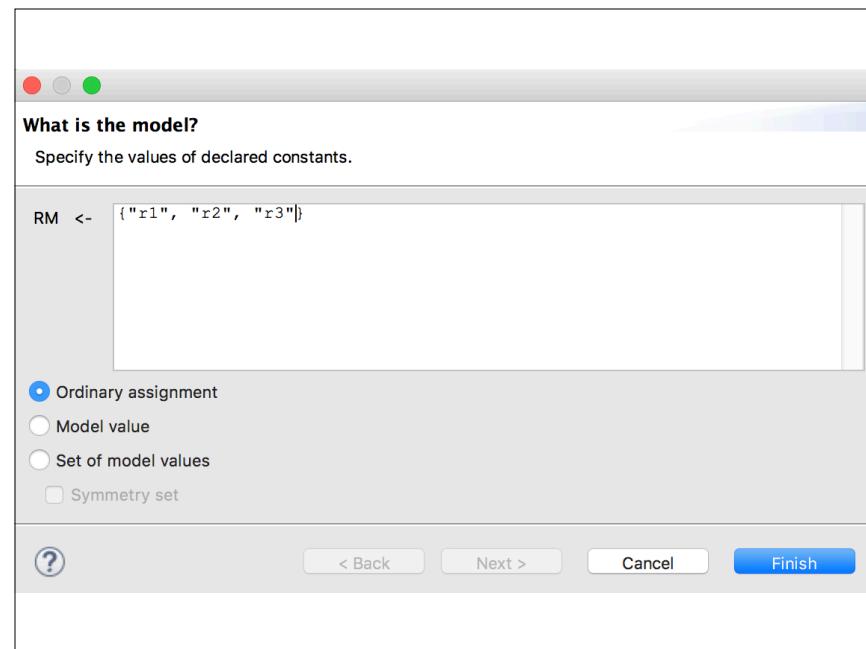
209



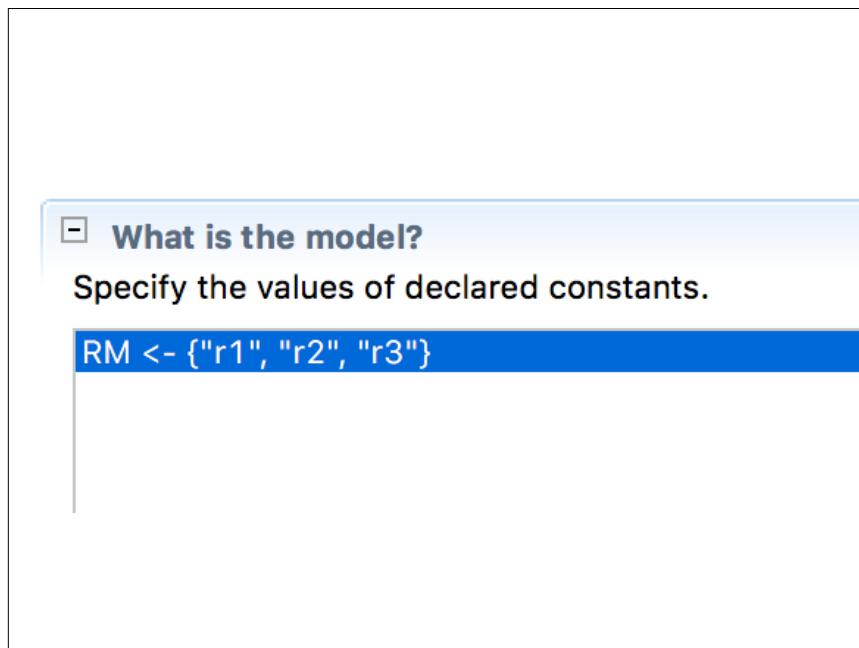
210



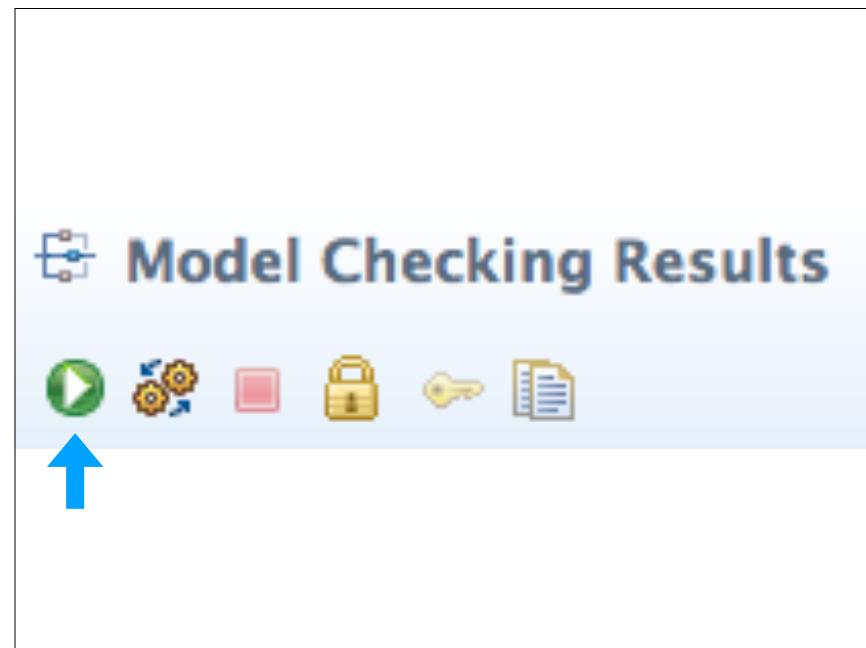
211



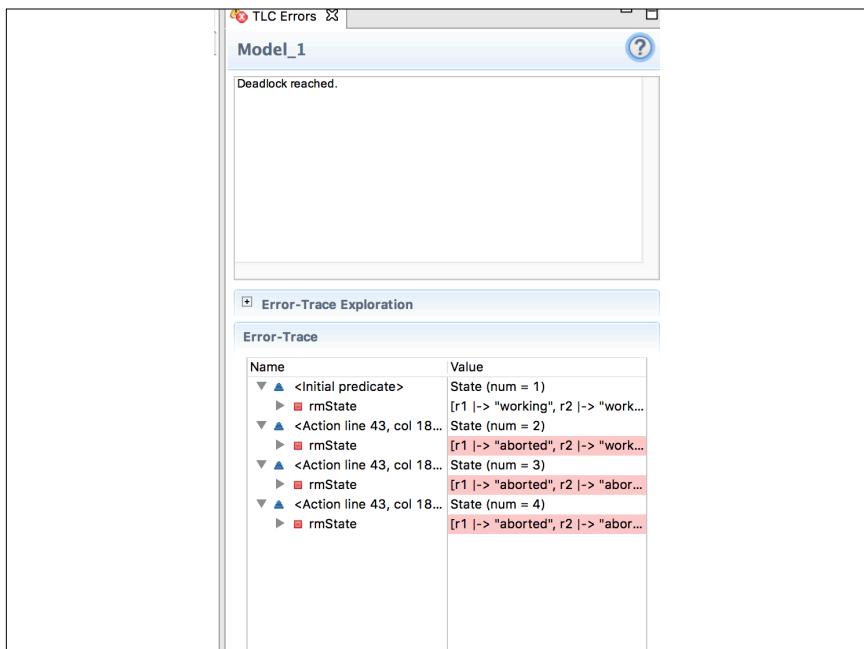
212



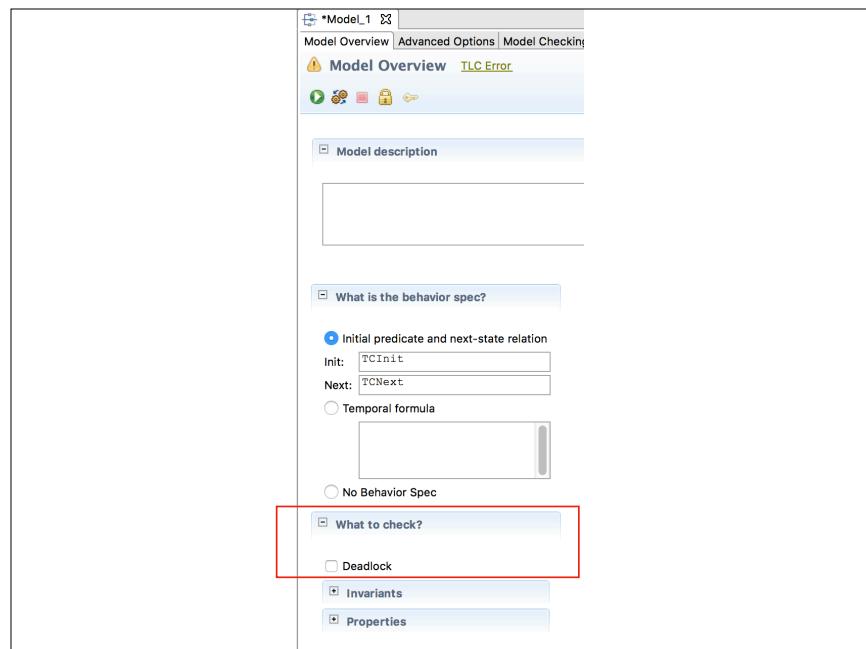
213



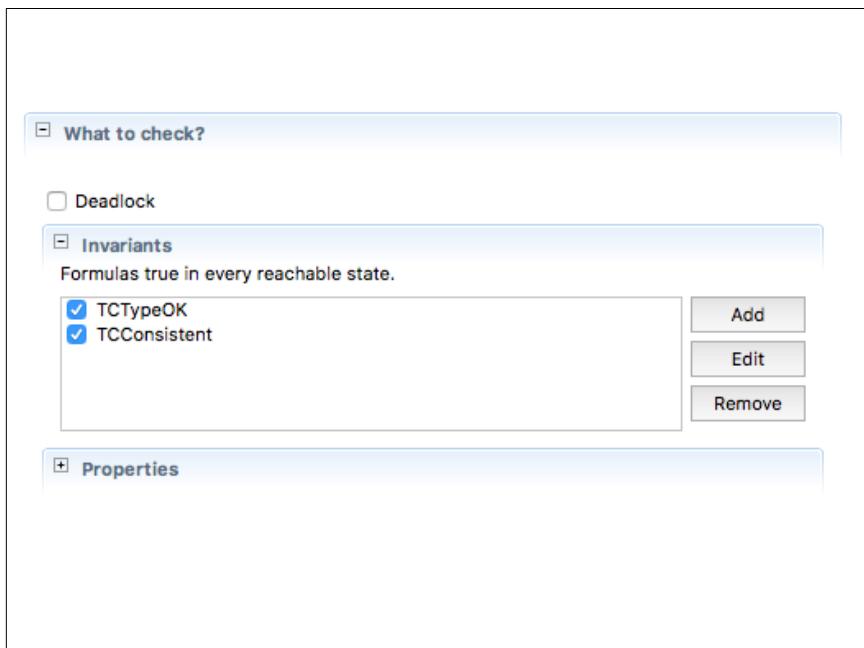
214



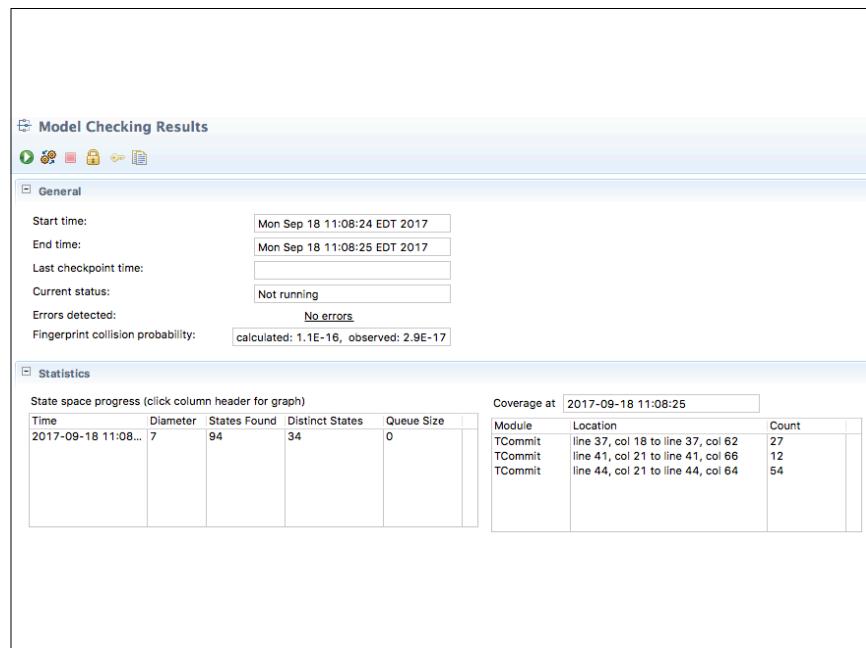
215



216



217



218

Let's try breaking something!

219

```
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.

```
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"]
```

220-1

```
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.
- This means that **canCommit** no longer needs to be true to allow a state transition where some RM **r** goes from “prepared” to “committed.”

220-2

220-3

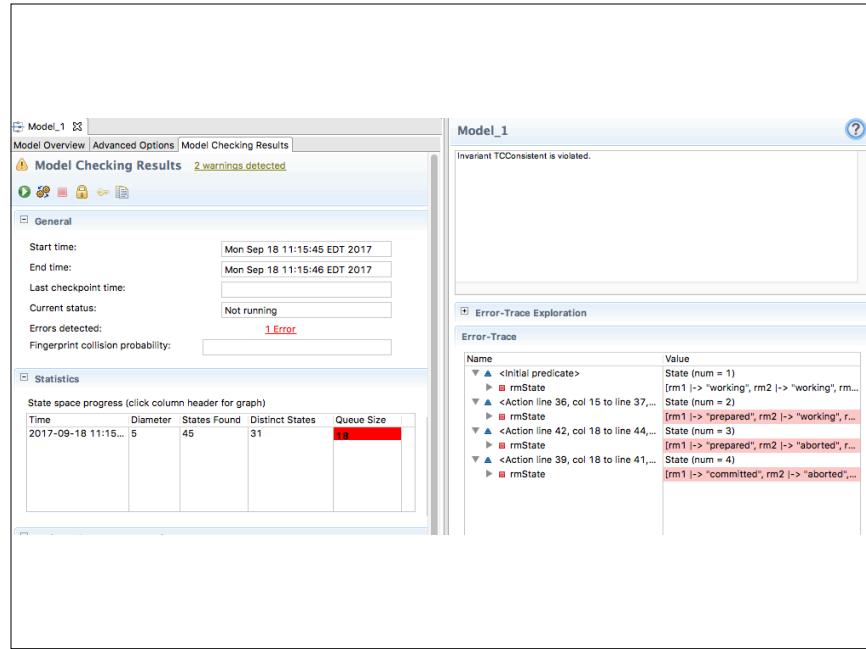
```

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"]

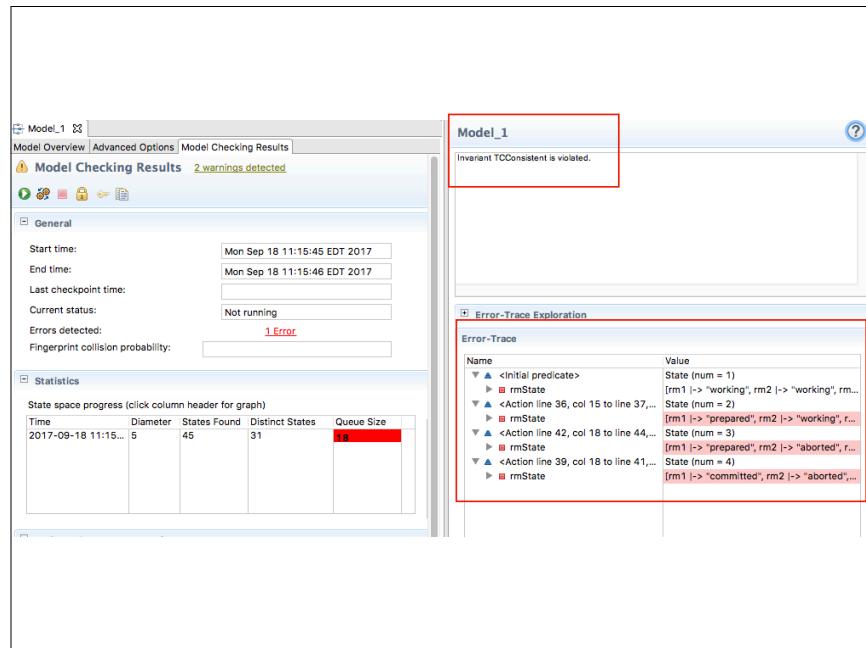
```

- The $\backslash*$ 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.

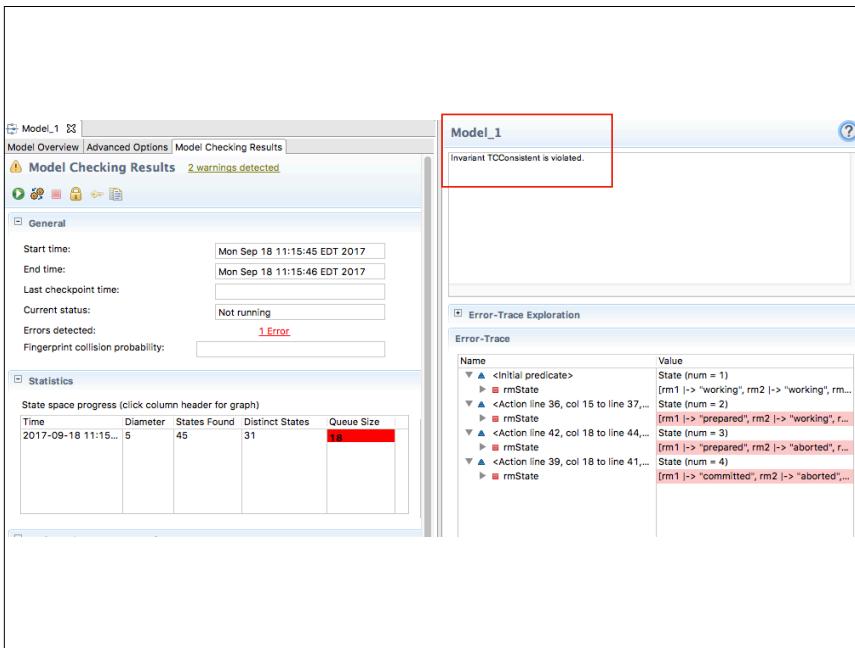
220-4



221-1



221-2



221-3

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"]

222

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

223

Resource Managers

rmState["rm1": "working"
rmState["rm2": "working"
rmState["rm3": "working"]

```

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) == \/\ rmState[r] = "prepared"
39           \/\ canCommit
40           /\ rmState' = [rmState EXCEPT ![r] = "committed"]
41           \/\ rmState' \in {"working", "prepared"}
42           /\ notCommitted
43           /\ rmState' = [rmState EXCEPT ![r] = "aborted"]
44
45

```

224

225-1

Resource Managers

```
rmState["rm1": "working"  
rmState["rm2": "working"  
rmState["rm3": "working"]]
```



Resource Managers

```
rmState["rm1": "working"  
rmState["rm2": "working"  
rmState["rm3": "working"]]
```



225-2

225-3

Resource Managers

```
rmState["rm1": "working"  
rmState["rm2": "working"  
rmState["rm3": "working"]]
```



```
rmState["rm1": "prepared"  
rmState["rm2": "aborted"  
rmState["rm2": "working"]]
```



Resource Managers

```
rmState["rm1": "working"  
rmState["rm2": "working"  
rmState["rm3": "working"]]
```



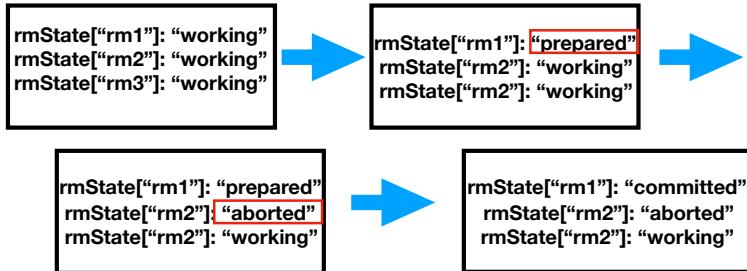
```
rmState["rm1": "prepared"  
rmState["rm2": "aborted"  
rmState["rm2": "working"]]
```



225-4

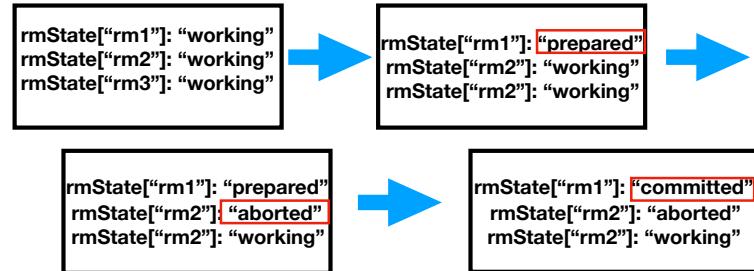
225-5

Resource Managers



225-6

Resource Managers



225-7

Syntax time

226

Functions

227-1

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”

227-2

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

227-3

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.

227-4

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

227-5

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

227-6

Functions

228-1

Functions

- TLA+ provides lots of ways to create functions. We'll only look at the commonly used ones

228-2

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**

228-3

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**

$[i \in \{2,3,4\} \mid\rightarrow i^2] = (2 :> 4 @@@ 3 :> 6 @@@ 4 :> 8)$

228-4

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"

228-5

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}

228-6

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"

228-7

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: i*2 for i in [2,3,4]}`

228-8

Functions

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} \mid\rightarrow x * 5 ]
```

```
f[1] = 5  
f[2] = 10  
f[3] = 15
```

229

Functions

- TLA+ has a special operator called **DOMAIN** which will return the domain of any function.

230-1

230-2

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}
```

230-3

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?

230-4

Functions

Functions

```
f == [x \in {1,2,3} |-> x*5]
```

231-1

231-2

Functions

```
f == [x \in {1,2,3} |-> x*5]
```

Answer:

231-3

Functions

```
f == [x \in {1,2,3} |-> x*5]
```

Answer:

```
{ f[i] : i \in DOMAIN f }
```

231-4

Sets of functions

232-1

Sets of functions

- It is very common in TLA+ to need to construct sets of functions.

232-2

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.

232-3

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.

232-4

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

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.

232-5

232-6

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**.

232-7

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**.

232-8

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.

232-9

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?

232-10

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"} }
```

233

:> 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)

234

Tuples

235-1

Tuples

- A tuple is like an array in a regular programming language.
But instead of 0-based, it's 1-based

235-2

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

235-3

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"`

235-4

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 `<< >>`

235-5

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

235-6

EXCEPT

236-1

EXCEPT

- A final important construct is EXCEPT

236-2

EXCEPT

- A final important construct is EXCEPT
- Say we have the following function

236-3

EXCEPT

- A final important construct is EXCEPT
- Say we have the following function

```
f == [x \in {2,4,6} | -> x*4]
```

236-4

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

236-5

# 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]
```

236-6

# 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:

236-7

# 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**

236-8

# 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
```

237

# Summary

- A **function** is a collection of **key:value** pairs.

# Summary

- A **function** is a collection of **key:value** pairs.
- The keys are called the **DOMAIN**, the values are the **range**.

238-2

238-3

# 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.

238-4

# 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.

238-5

```
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"}]
```

239-1

```
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"}]
```

239-2

```
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"}]
```

239-3

```
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"}]
```

239-4

```
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"}

239-5

```
TCTypeOK ==
 rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
```

240-1

```
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

240-2

```
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

240-3

```
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

240-4

```
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

240-5

```
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

240-6

```
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:

240-7

```
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"] }
```

240-8

```
TCTypeOK ==
rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
```

241-1

```
TCTypeOK ==
rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
```

- **rmState** is the only variable we're using in this spec

241-2

```
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.)

241-3

```
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:

241-4

```
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

241-5

```
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

241-6

```
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

241-7

## Types

- TLA+ doesn't have an elaborate type system, on purpose

242-1

242-2

# 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

242-3

# 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"

242-4

# 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

242-5

# 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`

242-6

# 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

242-7

```
TCInit == rmState = [r \in RM |-> "working"]
```

243-1

```
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**."

243-2

```
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.

243-3

```
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"]
```

243-4

```
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

243-5

```
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

243-6

## Initial State

244-1

## Initial State

```
rmState: [r1 |-> "working", r2 |-> "working"]
```

244-2

## Next state formula

$\text{TCNext} == \forall r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$

245-1

## Next state formula

$\text{TCNext} == \forall r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$

- Let's skip a bunch and scroll down to  $\text{TCNext}$ , the “next state formula”

245-2

## Next state formula

$\text{TCNext} == \forall r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$

- Let's skip a bunch and scroll down to  $\text{TCNext}$ , the “next state formula”
  - This says:

245-3

# 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 **Prepare( $r$ )** is TRUE or **Decide( $r$ )** is TRUE"

245-4

# 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 **Prepare( $r$ )** is TRUE or **Decide( $r$ )** is TRUE"
- Remember the fundamental concept of TLA+

245-5

# 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 **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

245-6

# 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 **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

245-7

## Next state formula

```
TCNext == \E r \in RM : Prepare(r) \vee Decide(r)
```

246-1

## Next state formula

```
TCNext == \E r \in RM : Prepare(r) \vee Decide(r)
```

- Let's look at `Prepare(r)` to get some clarity

246-2

## 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"]
```

246-3

## 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:

246-4

# Next state formula

$\text{TCNext} == \forall r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$

- Let's look at  $\text{Prepare}(r)$  to get some clarity

$\text{Prepare}(r) == \wedge \text{rmState}[r] = \text{"working"} \wedge \text{rmState}' = [\text{rmState EXCEPT } !r] = \text{"prepared"}]$

- This says:

- This formula is true for a pair of states if in the **current state**, resource manager  $r$  is "working"

246-5

# Next state formula

$\text{TCNext} == \forall r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$

- Let's look at  $\text{Prepare}(r)$  to get some clarity

$\text{Prepare}(r) == \wedge \text{rmState}[r] = \text{"working"} \wedge \text{rmState}' = [\text{rmState EXCEPT } !r] = \text{"prepared"}]$

- 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

246-6

# Next state formula

$\text{TCNext} == \forall r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$

$\text{Prepare}(r) == \wedge \text{rmState}[r] = \text{"working"} \wedge \text{rmState}' = [\text{rmState EXCEPT } !r] = \text{"prepared"}]$

rmState:  $[r1 \rightarrow \text{"prepared"}, r2 \rightarrow \text{"working"}]$

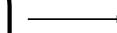
247-1

# Next state formula

$\text{TCNext} == \forall r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$

$\text{Prepare}(r) == \wedge \text{rmState}[r] = \text{"working"} \wedge \text{rmState}' = [\text{rmState EXCEPT } !r] = \text{"prepared"}]$

rmState:  $[r1 \rightarrow \text{"prepared"}, r2 \rightarrow \text{"working"}]$



247-2

# Next state formula

TCNext ==  $\exists r \in RM : Prepare(r) \vee Decide(r)$

Prepare(r) ==  $\wedge rmState[r] = "working"$   
 $\wedge rmState' = [rmState \text{ EXCEPT } !r] = "prepared"$



247-3

# Next state formula

TCNext ==  $\exists r \in RM : Prepare(r) \vee Decide(r)$

Prepare(r) ==  $\wedge rmState[r] = "working"$   
 $\wedge rmState' = [rmState \text{ EXCEPT } !r] = "prepared"$

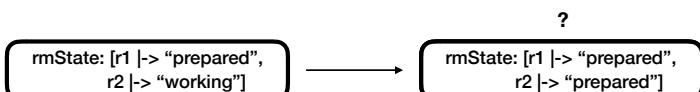


247-4

# Next state formula

TCNext ==  $\exists r \in RM : Prepare(r) \vee Decide(r)$

Prepare(r) ==  $\wedge rmState[r] = "working"$   
 $\wedge rmState' = [rmState \text{ EXCEPT } !r] = "prepared"$



247-5

# Next state formula

TCNext ==  $\exists r \in RM : Prepare(r) \vee Decide(r)$

Prepare(r) ==  $\wedge rmState[r] = "working"$   
 $\wedge rmState' = [rmState \text{ EXCEPT } !r] = "prepared"$

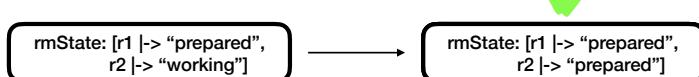


247-6

# Next state formula

TCNext ==  $\forall r \in RM : Prepare(r) \vee Decide(r)$

Prepare(r) ==  $\wedge rmState[r] = "working"$   
 $\wedge rmState' = [rmState \text{ EXCEPT } !r] = "prepared"$



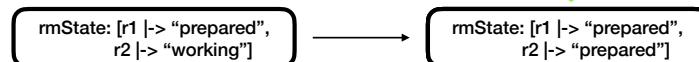
- The parts of the formula with no primed variables are called **enabling conditions**

247-7

# Next state formula

TCNext ==  $\forall r \in RM : Prepare(r) \vee Decide(r)$

Prepare(r) ==  $\wedge rmState[r] = "working"$  ←  
 $\wedge rmState' = [rmState \text{ EXCEPT } !r] = "prepared"$



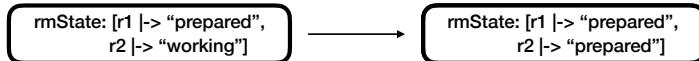
- The parts of the formula with no primed variables are called **enabling conditions**

247-8

# Next state formula

TCNext ==  $\forall r \in RM : Prepare(r) \vee Decide(r)$

Prepare(r) ==  $\wedge rmState[r] = "working"$  ←  
 $\wedge rmState' = [rmState \text{ EXCEPT } !r] = "prepared"$



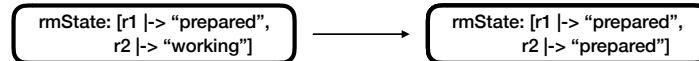
- The parts of the formula with no primed variables are called **enabling conditions**
- They describe what the **current state** must look like to be able to use this action

247-9

# Next state formula

TCNext ==  $\forall r \in RM : Prepare(r) \vee Decide(r)$

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**
- 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**

247-10

# Next state formula

```

TCNext == \E r \in RM : Prepare(r) \vee Decide(r)

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"]

```

248-1

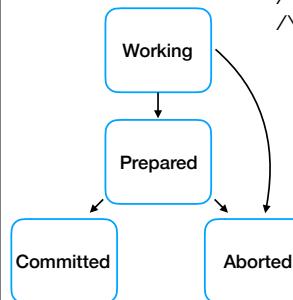
# Next state formula

```

TCNext == \E r \in RM : Prepare(r) \vee Decide(r)

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"]

```



248-2

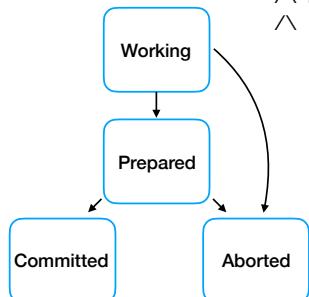
# Next state formula

```

TCNext == \E r \in RM : Prepare(r) \vee Decide(r)

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"]

```



248-3

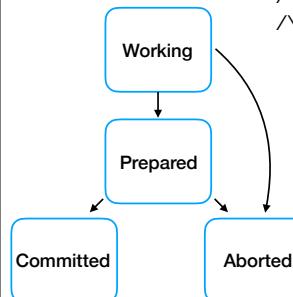
# Next state formula

```

TCNext == \E r \in RM : Prepare(r) \vee Decide(r)

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"]

```

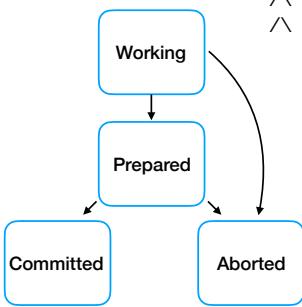


248-4

# Next state formula

$\text{TCNext} == \exists r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$

$\text{Decide}(r) == \bigvee \begin{array}{l} \wedge \text{rmState}[r] = \text{"prepared"} \\ \wedge \text{canCommit} \\ \wedge \text{rmState}' = [\text{rmState EXCEPT } !r = \text{"committed"}] \\ \vee \begin{array}{l} \wedge \text{rmState}[r] \in \{\text{"working"}, \text{"prepared"}\} \\ \wedge \text{notCommitted} \\ \wedge \text{rmState}' = [\text{rmState EXCEPT } !r = \text{"aborted"}] \end{array} \end{array}$



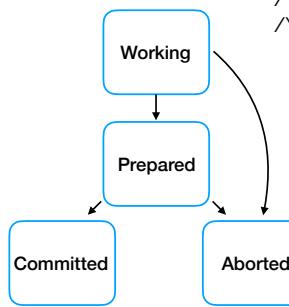
- This formula is true when either
  - RM r is in state “prepared”

248-5

# Next state formula

$\text{TCNext} == \exists r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$

$\text{Decide}(r) == \bigvee \begin{array}{l} \wedge \text{rmState}[r] = \text{"prepared"} \\ \wedge \text{canCommit} \\ \wedge \text{rmState}' = [\text{rmState EXCEPT } !r = \text{"committed"}] \\ \vee \begin{array}{l} \wedge \text{rmState}[r] \in \{\text{"working"}, \text{"prepared"}\} \\ \wedge \text{notCommitted} \\ \wedge \text{rmState}' = [\text{rmState EXCEPT } !r = \text{"aborted"}] \end{array} \end{array}$



- This formula is true when either
  - RM r is in state “prepared”

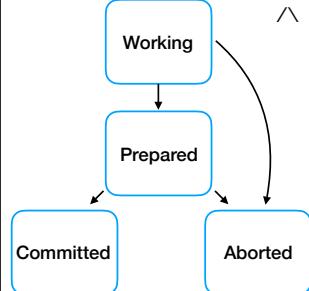
248-6

# Next state formula

$\text{TCNext} == \exists r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$

$\text{Decide}(r) == \bigvee \begin{array}{l} \wedge \text{rmState}[r] = \text{"prepared"} \\ \wedge \text{canCommit} \\ \wedge \text{rmState}' = [\text{rmState EXCEPT } !r = \text{"committed"}] \\ \vee \begin{array}{l} \wedge \text{rmState}[r] \in \{\text{"working"}, \text{"prepared"}\} \\ \wedge \text{notCommitted} \\ \wedge \text{rmState}' = [\text{rmState EXCEPT } !r = \text{"aborted"}] \end{array} \end{array}$

- This formula is true when either
  - RM r is in state “prepared”
  - And **canCommit** is true

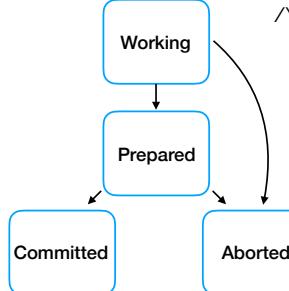


248-7

# Next state formula

$\text{TCNext} == \exists r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$

$\text{Decide}(r) == \bigvee \begin{array}{l} \wedge \text{rmState}[r] = \text{"prepared"} \\ \wedge \text{canCommit} \\ \wedge \text{rmState}' = [\text{rmState EXCEPT } !r = \text{"committed"}] \\ \vee \begin{array}{l} \wedge \text{rmState}[r] \in \{\text{"working"}, \text{"prepared"}\} \\ \wedge \text{notCommitted} \\ \wedge \text{rmState}' = [\text{rmState EXCEPT } !r = \text{"aborted"}] \end{array} \end{array}$

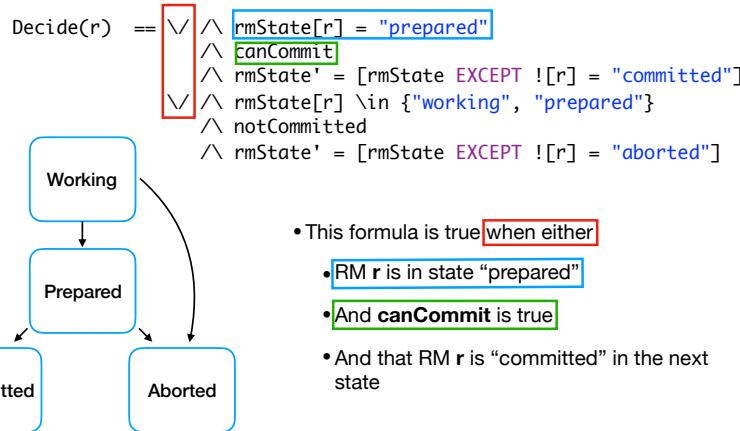


- This formula is true when either
  - RM r is in state “prepared”
  - And **canCommit** is true

248-8

# Next state formula

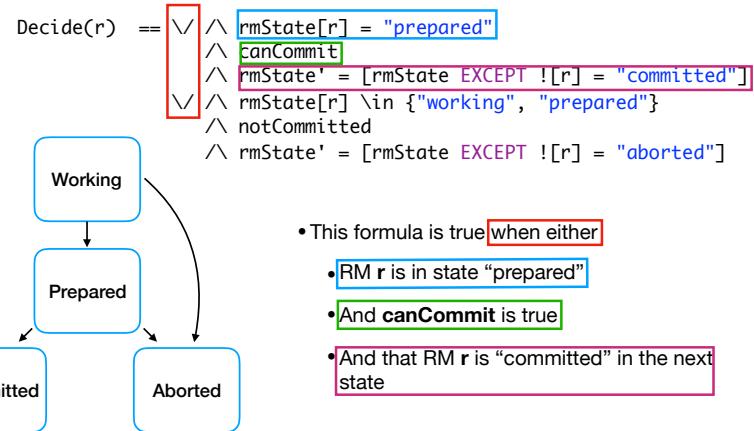
$\text{TCNext} == \forall r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$



248-9

# Next state formula

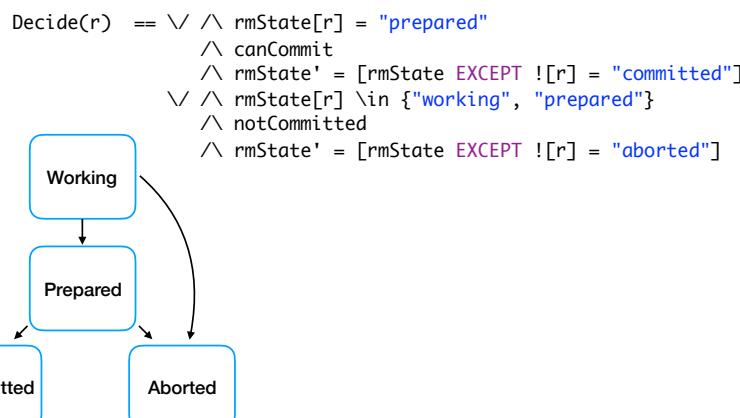
$\text{TCNext} == \forall r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$



248-10

# Next state formula

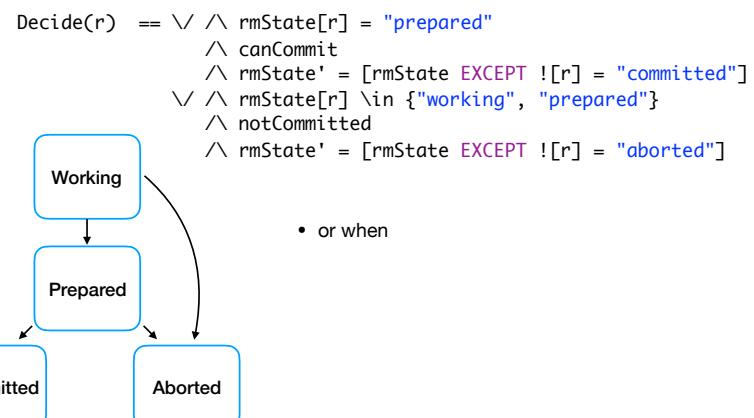
$\text{TCNext} == \forall r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$



249-1

# Next state formula

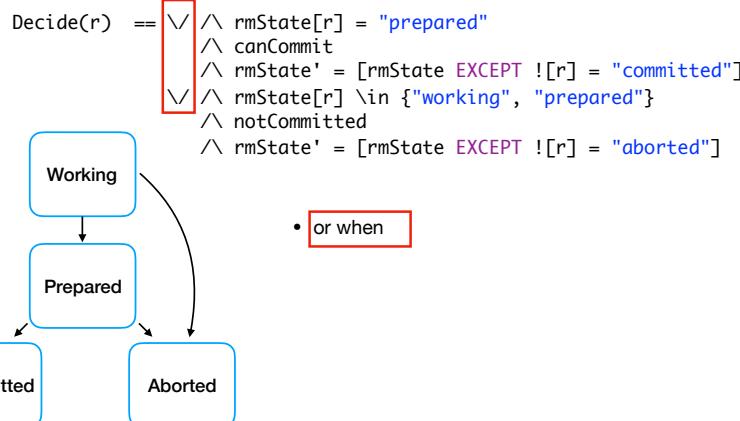
$\text{TCNext} == \forall r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$



249-2

# Next state formula

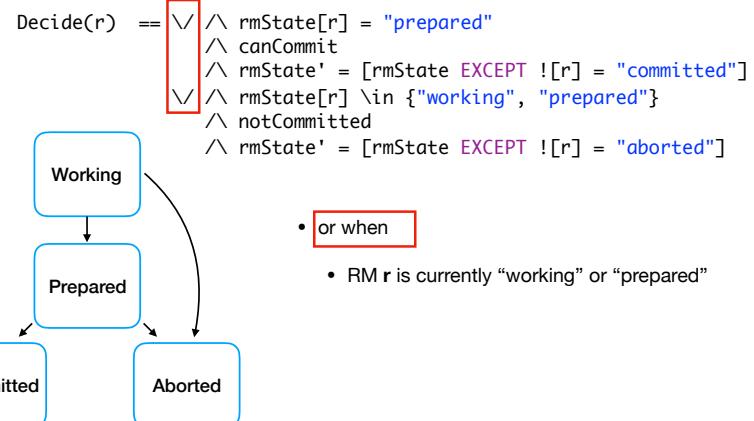
$\text{TCNext} == \exists r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$



249-3

# Next state formula

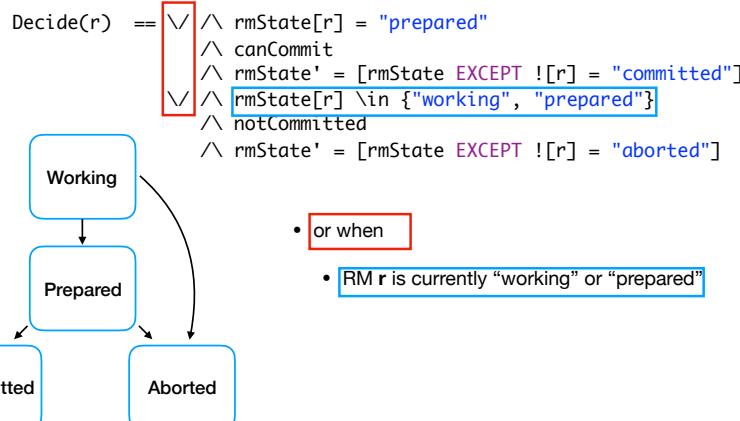
$\text{TCNext} == \exists r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$



249-4

# Next state formula

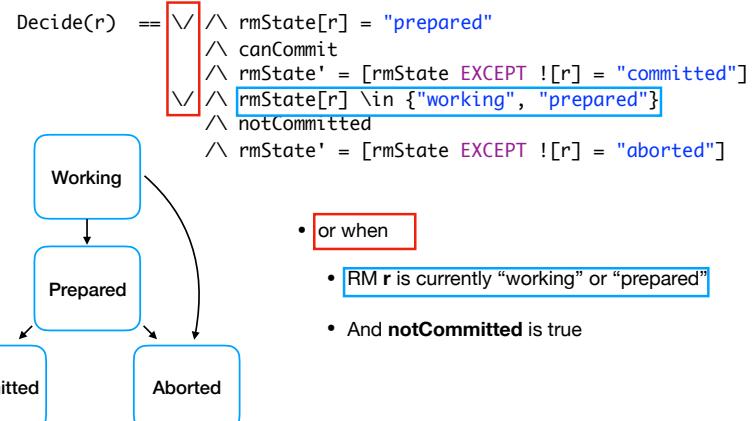
$\text{TCNext} == \exists r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$



249-5

# Next state formula

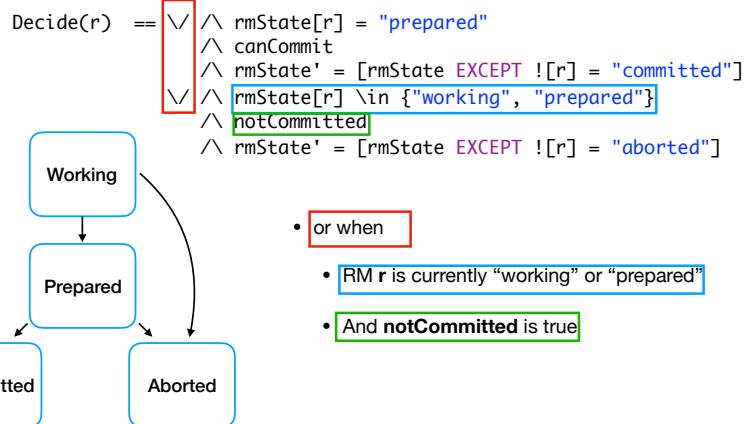
$\text{TCNext} == \exists r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$



249-6

# Next state formula

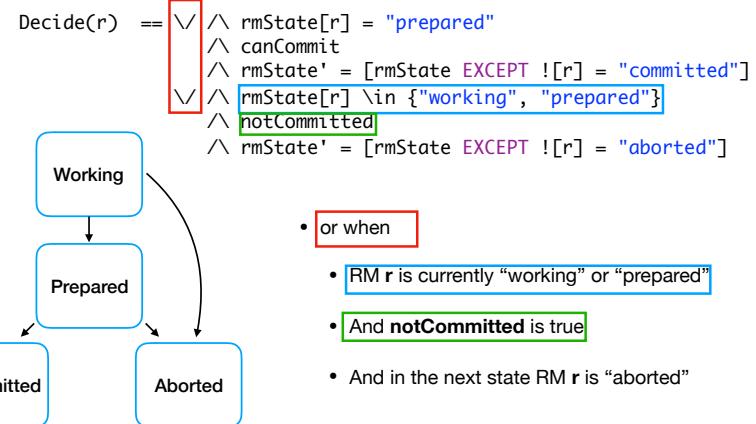
$\text{TCNext} == \exists r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$



249-7

# Next state formula

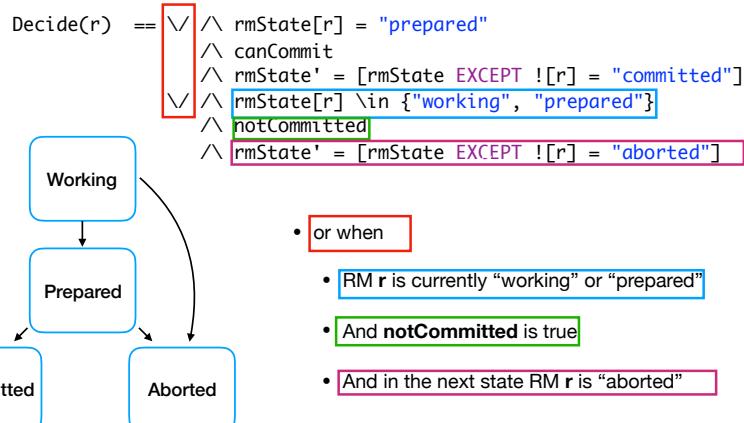
$\text{TCNext} == \exists r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$



249-8

# Next state formula

$\text{TCNext} == \exists r \in \text{RM} : \text{Prepare}(r) \vee \text{Decide}(r)$



249-9

# canCommit

$\text{canCommit} == \forall r \in \text{RM} : \text{rmState}[r] \in \{\text{"prepared"}, \text{"committed"}\}$

250-1

# 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))
```

250-2

# 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))
```

250-3

# 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))
```

250-4

# 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))
```

250-5

# notCommitted

```
notCommitted == \A r \in RM : rmState[r] # "committed"
```

"Not equal." You can also use /=

251-1

# notCommitted

```
notCommitted == \A r \in RM : rmState[r] # "committed"
```

## Python Equivalents

```
RM = ["rm1", "rm2"]
rmState = {"rm1": "working", "rm2": "working"}
...
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))
```

251-2

# notCommitted

```
notCommitted == \A r \in RM : rmState[r] # "committed"
```

## Python Equivalents

```
RM = ["rm1", "rm2"]
rmState = {"rm1": "working", "rm2": "working"}
...
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))
```

251-3

# notCommitted

```
notCommitted == \A r \in RM : rmState[r] # "committed"
```

## Python Equivalents

```
RM = ["rm1", "rm2"]
rmState = {"rm1": "working", "rm2": "working"}
...
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))
```

251-4

# 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))
```

251-5

# Next state formula

```
TCNext == \E r \in RM : Prepare(r) \vee Decide(r)
```

252-1

# Next state formula

```
TCNext == \E r \in RM : Prepare(r) \vee 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.

# Next state formula

```
TCNext == \E r \in RM : Prepare(r) \vee 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

252-2

252-3

# Next state formula

```
TCNext == \E r \in RM : Prepare(r) \vee 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

252-4

# TCConsistent

```
TCConsistent ==
 \A r1, r2 \in RM : ~ \wedge rmState[r1] = "aborted"
 \wedge rmState[r2] = "committed"
```

253-1

# 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

253-2

# 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

253-3

# 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"`

253-4

# 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"`

253-5

# TCConsistent

```
TCConsistent ==
 \A r1, r2 \in RM : ~ \wedge rmState[r1] = "aborted"
 \wedge rmState[r2] = "committed"
```

254-1

# 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
```

254-2

# TCommit Recap

255-1

# TCommit Recap

- We have specified **what** a “transaction commit” system should do

255-2

# TCommit Recap

- We have specified **what** a “transaction commit” system should do
  - All RMs must agree on committing or aborting a transaction

255-3

# 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

255-4

# 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

255-5

# 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

255-6

# 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)

255-7

# Two Phase Commit

256-1

## Two Phase Commit

- A simple solution of the Transaction Commit problem

256-2

## Two Phase Commit

- A simple solution of the Transaction Commit problem
- Introduces a “Transaction Manager” (TM) to coordinate with the RMs

256-3

## 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

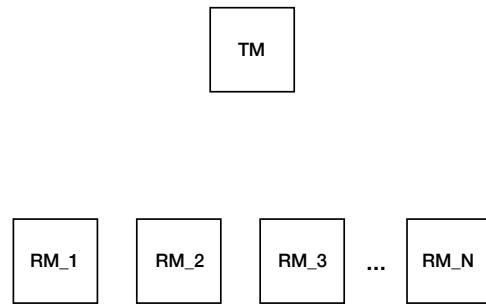
256-4

## Transaction Manager



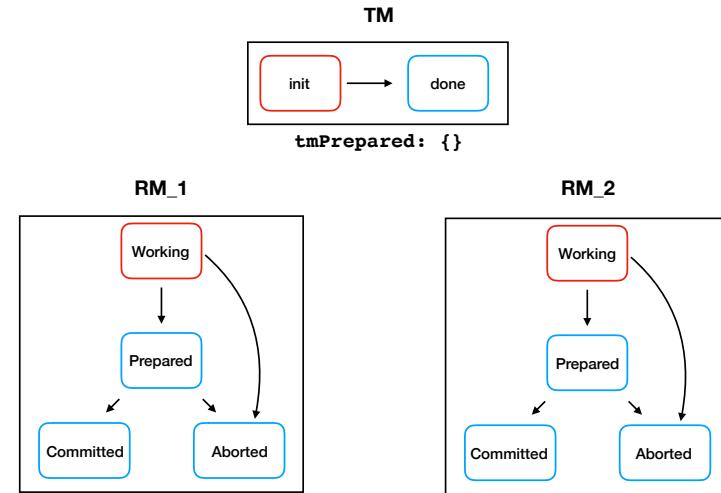
257-1

# Transaction Manager



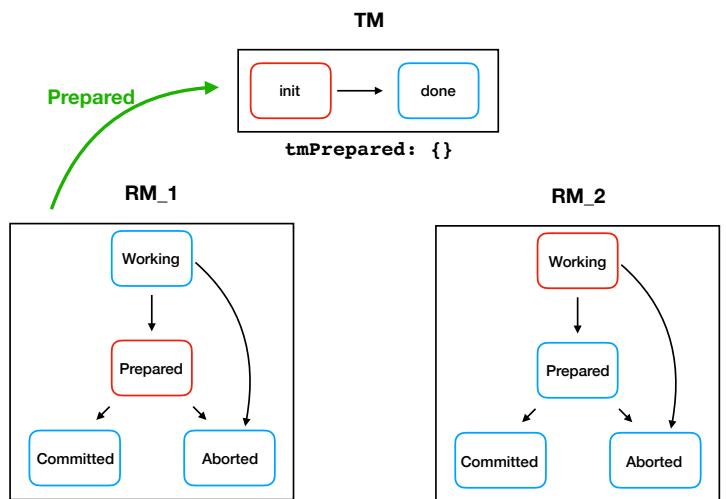
257-2

# Transaction Manager



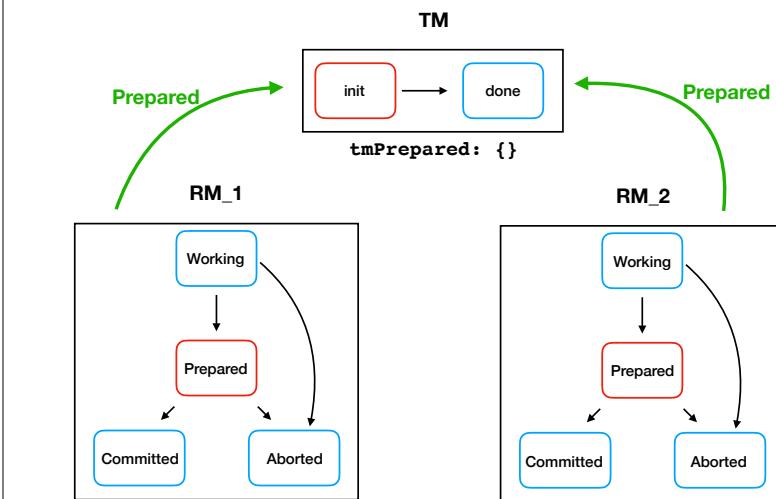
258

# Transaction Manager



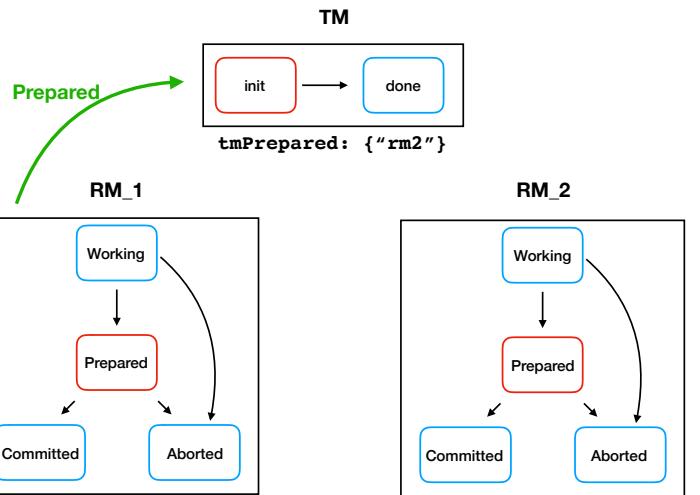
259

# Transaction Manager



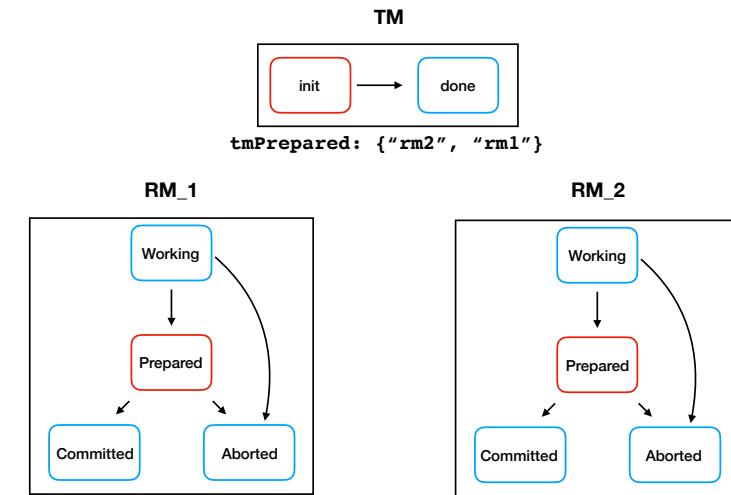
260

# Transaction Manager



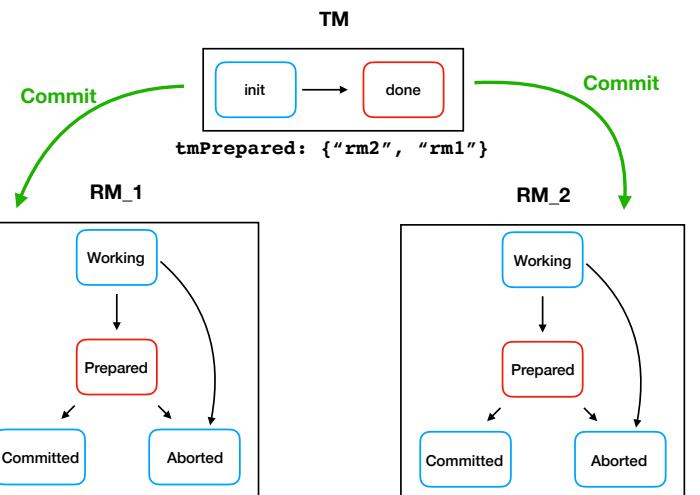
261

# Transaction Manager



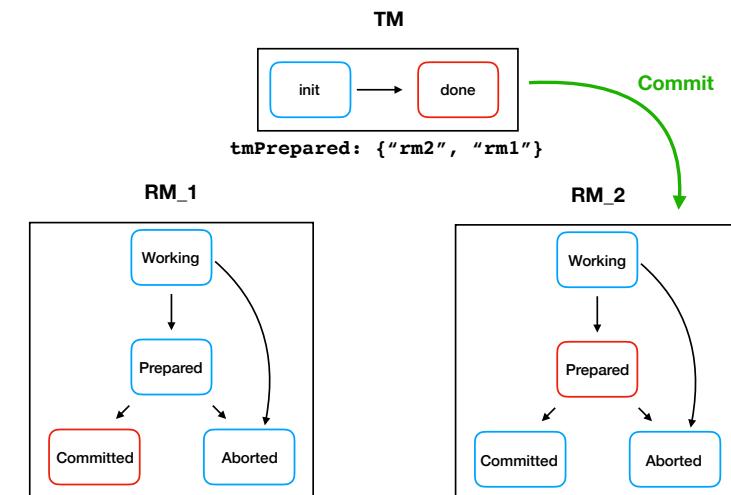
262

# Transaction Manager



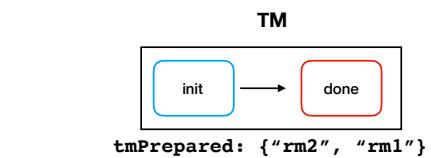
263

# Transaction Manager



264

# Transaction Manager



265

Toolbox File Edit Window TLC Model Checker TLA Proof Manager

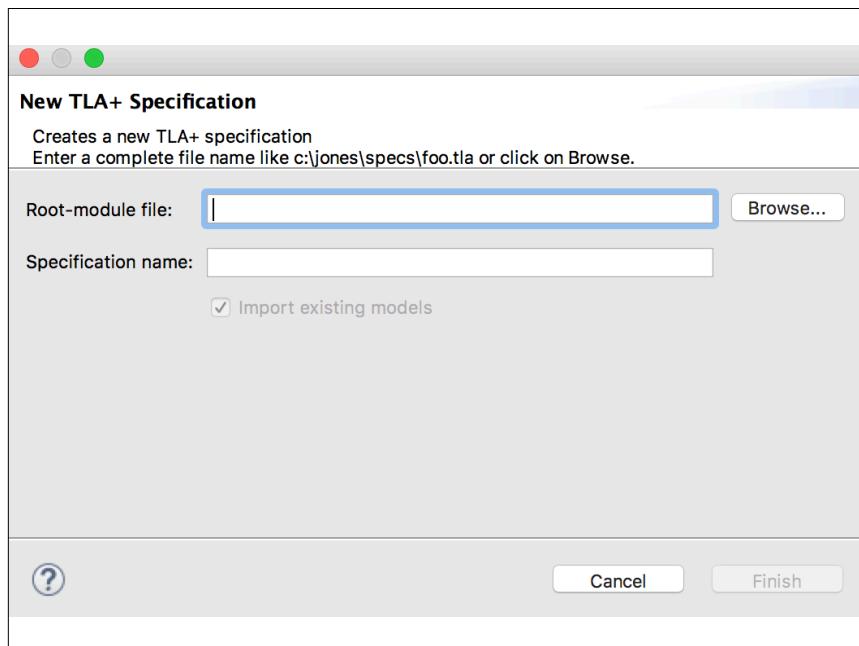
Open Spec

Add New Spec...

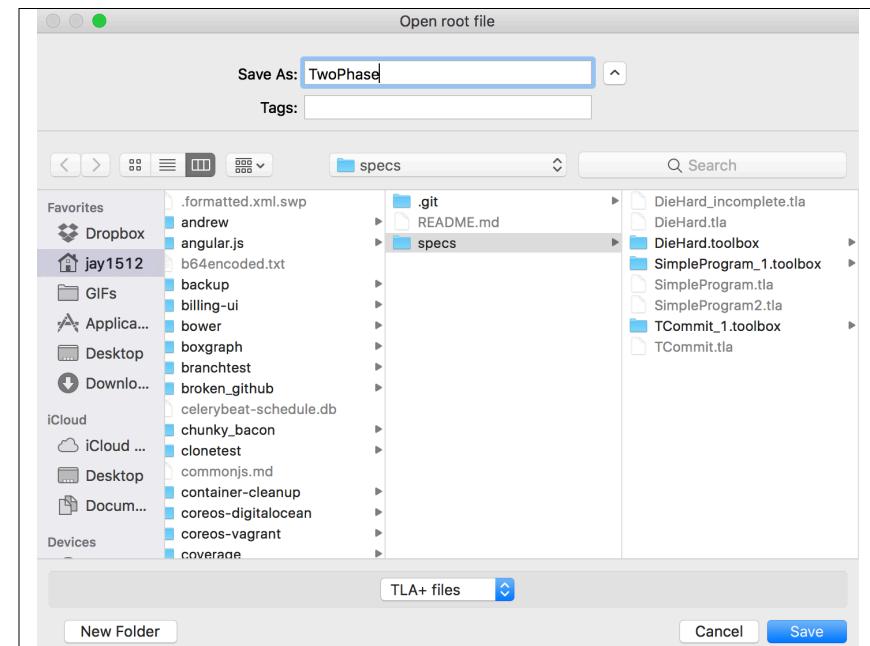
Parse Spec

▼ R

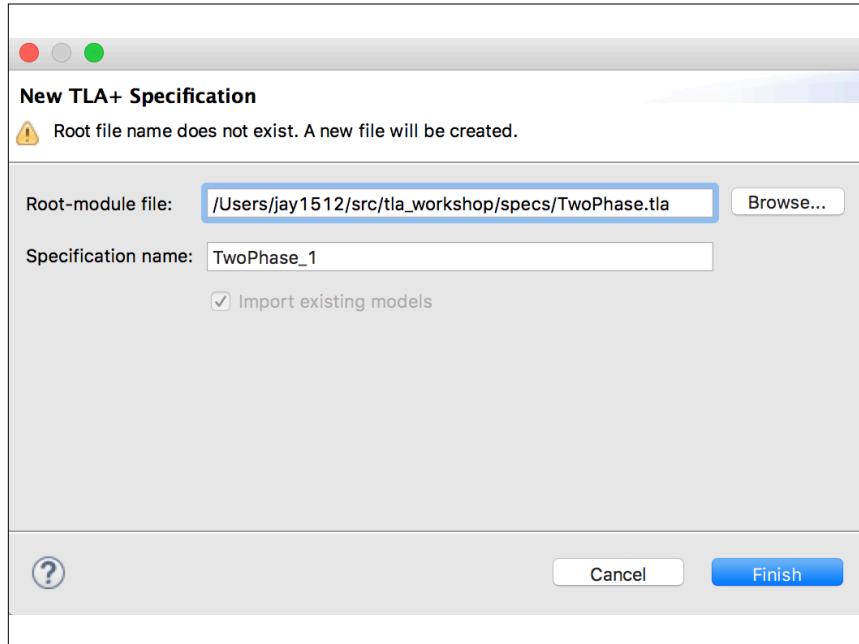
266



267



268



269

```

----- MODULE TwoPhase -----
(* This specification is discussed in "Two-Phase Commit", Lecture 6 of the *)
(* TLA+ course. See also the slides for Lecture 6. *)
(* 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"}]

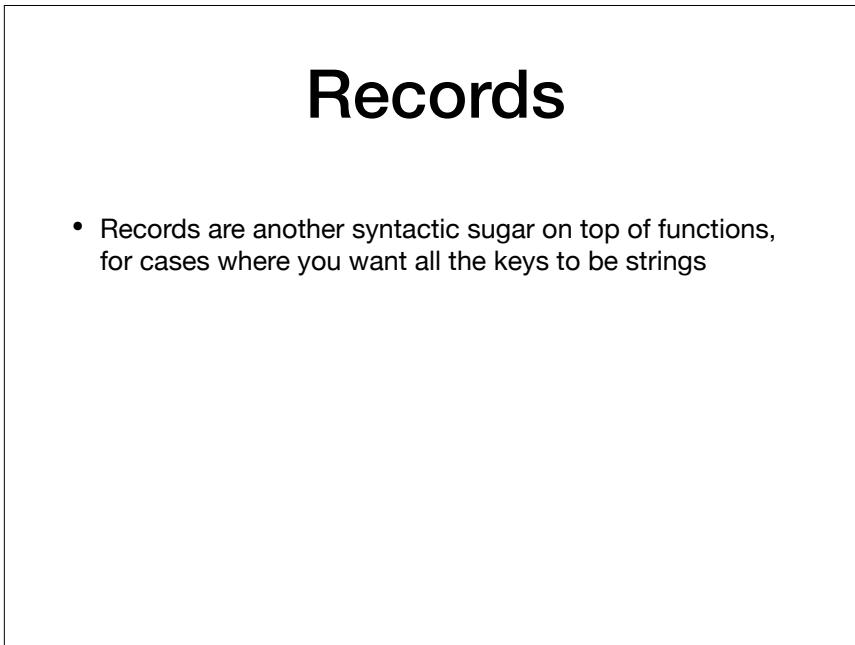
```

specs/TwoPhase.tla

270

## Records

- Records are another syntactic sugar on top of functions, for cases where you want all the keys to be strings



271-1

## 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]
```

271-2

# 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}
```

271-3

# 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"}
```

271-4

# 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
```

271-5

# 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"}
```

271-6

# 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

271-7

# 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}
```

271-8

# 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}
```

271-9

# Sets of records

272-1

## Sets of records

- We also often need to easily create sets of records

272-2

## 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

```
[nodes: {1,2}, edges: {"a","b","c"}]
```

272-4

## 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

272-3

## 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"] }
```

272-5

# Sets of records

```
[nodes: {1,2}, edges: {"a","b","c"}]
```

- This is equivalent to the following Python:

273-1

# 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})
```

273-2

# Sets of records

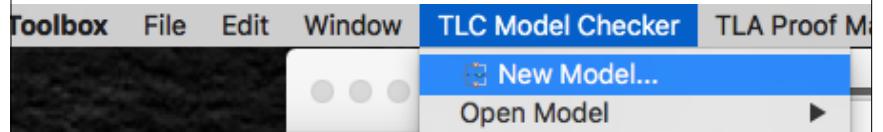
```
[nodes: {1,2}, edges: {"a","b","c"}]
```

- This is equivalent to the following Python:

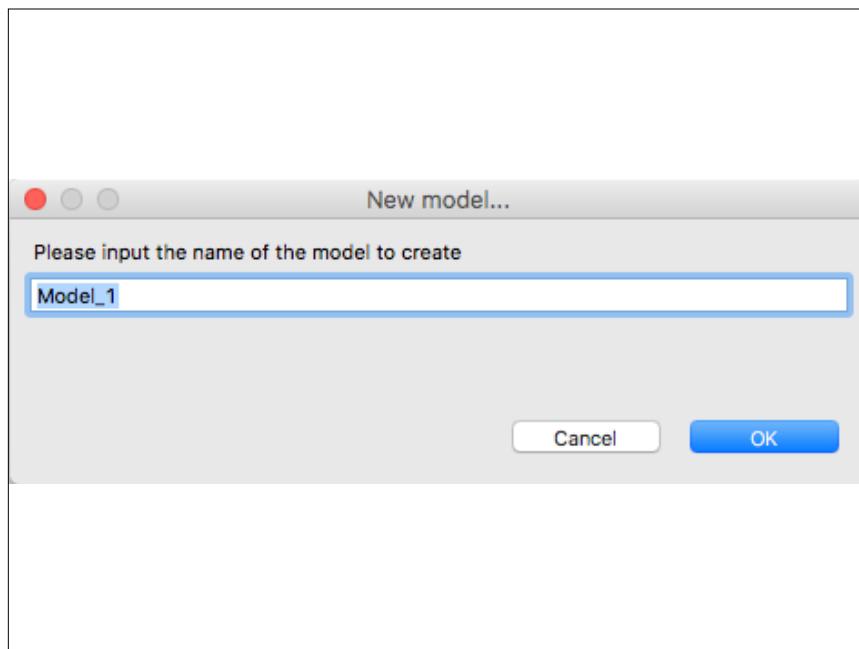
```
set_of_records = set()
for node in [1,2]:
 for edge in ["a","b","c"]:
 set_of_records.add({"nodes": node, "edges": edge})
```

```
{ {"nodes": 1, "edges": "a"},
 {"nodes": 1, "edges": "b"},
 {"nodes": 1, "edges": "c"},
 {"nodes": 2, "edges": "a"},
 {"nodes": 2, "edges": "b"},
 {"nodes": 2, "edges": "c" } }
```

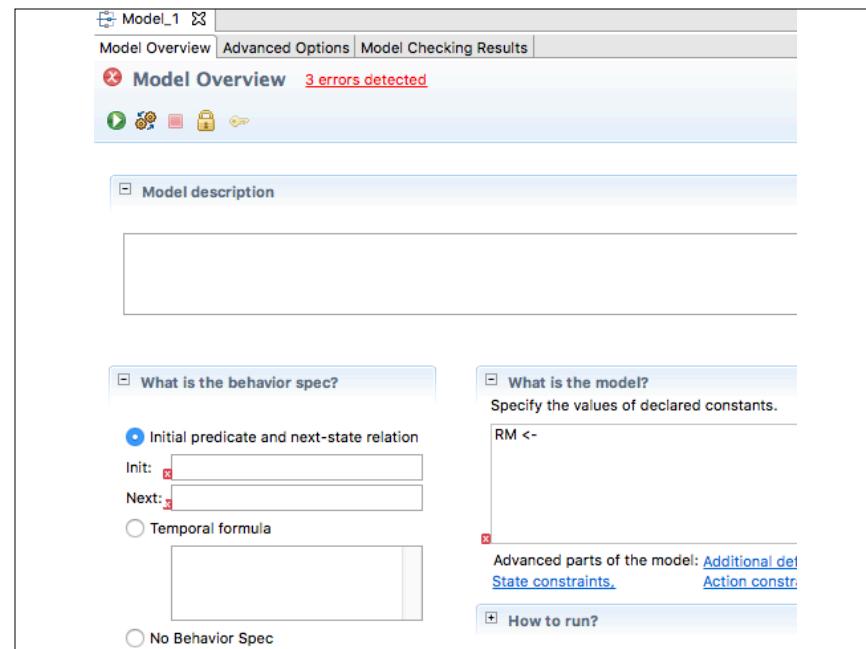
273-3



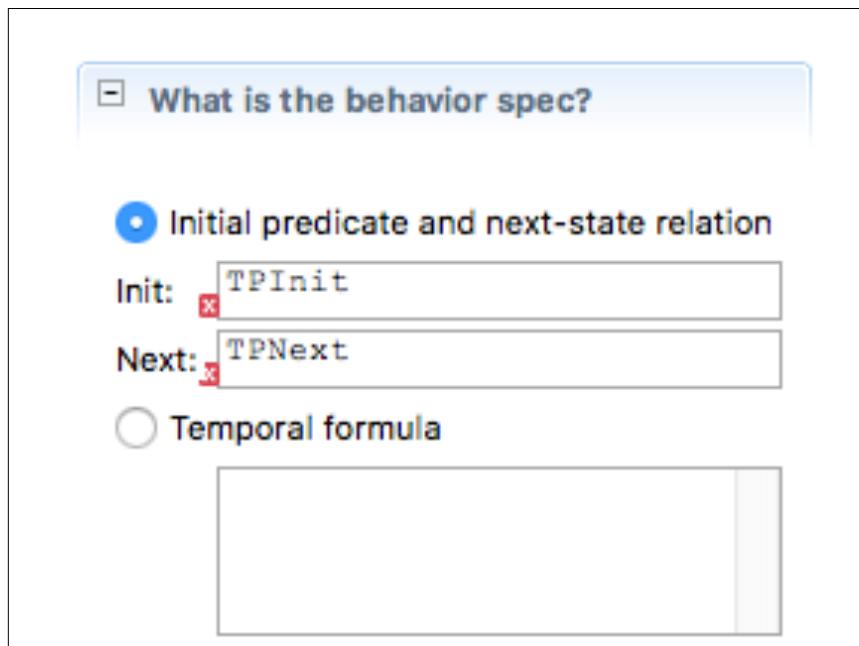
274



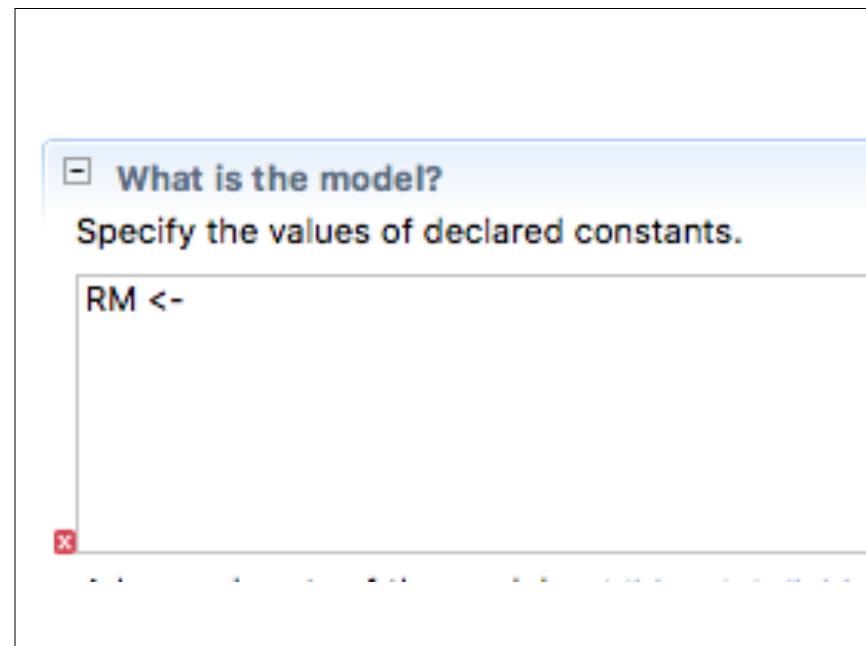
275



276



277



278

What is the model?  
Specify the values of declared constants.

```
RM <- {"r1", "r2", "r3"}
```

Ordinary assignment  
 Model value  
 Set of model values  
 Symmetry set

[?](#)    < Back    Next >    Cancel    **Finish**

279

**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 diag.](#) [State constraints.](#) [Action const](#)

How to run?

280

## Model Checking Results

A blue arrow points upwards from the bottom of the page towards the icons.

281

[Model Overview](#) [Advanced Options](#) [Model Checking Results](#)

**Model Checking Results**

General... Stops the current TLC model checker run.

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          |

282

\*Model\_1

Model Overview Advanced Options Model Checking Results

### Model Checking Results

- General

Start time: Tue Sep 19 12:08:40 EDT 2017  
 End time: Tue Sep 19 12:08:40 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 | Coverage at                                                                                            |
|---------------------|----------|--------------|-----------------|------------|--------------------------------------------------------------------------------------------------------|
| 2017-09-19 12:08... | 11       | 1146         | 288             | 0          | Module<br>TwoPhase<br>TwoPhase<br>TwoPhase<br>TwoPhase<br>TwoPhase<br>TwoPhase<br>TwoPhase<br>TwoPhase |

- Evaluate Constant Expression

Expression:

```
[nodes: {1,2}, edges: {"a", "b", "c"}]
```

283

### What to check?

- Deadlock
- Invariants

Formulas true in every reachable state.

- TPTTypeOK
- TCConsistent

- Add
- Edit
- Remove

### Properties

284

### 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 | Coverage at         |
|---------------------|----------|--------------|-----------------|------------|---------------------|
| 2017-09-19 16:47... | 11       | 1146         | 288             | 0          | 2017-09-19 16:47:41 |

| Module   | Location                           | Count |
|----------|------------------------------------|-------|
| TwoPhase | line 81, col 36 to line 80, col 39 | 56    |
| 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

Expression:

```
[nodes: {1,2}, edges: {"a", "b", "c"}]
```

285

### Evaluate Constant Expression

Expression:

```
[nodes: {1,2}, edges: {"a", "b", "c"}]
```

286

Evaluate Constant Expression

Expression:  
[nodes: {1,2}, edges: {"a","b","c"}]

Value:  
{ [nodes l-> 1, edges l-> "a"],  
  [nodes l-> 1, edges l-> "b"],  
  [nodes l-> 1, edges l-> "c"],  
  [nodes l-> 2, edges l-> "a"],  
  [nodes l-> 2, edges l-> "b"],  
  [nodes l-> 2, edges l-> "c"] }

287

# 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.

288-1

# 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

288-2

# 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"

288-3

# TwoPhase

```
CONSTANT RM /* The set of resource managers

VARIABLES
 rmState,
 tmState,
 tmPrepared,
```

msgs

- **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"})

288-4

# TwoPhase

```
CONSTANT RM /* The set of resource managers

VARIABLES
 rmState,
 tmState,
 tmPrepared,
```

msgs

- **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

288-5

# 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"}]
```

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"}]
```

289-3

# 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"}]
```



289-4

# 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"}]
```

290-1

# 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"}]
```

290-2

# 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: {"Commit", "Abort"}]

Value: {[type |> "Commit"], [type |> "Abort"]}

290-3

# 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"}]
```

291-1

# 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"}]
```

```
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"],  
[type |> "Abort"],  
[type |> "Prepared", rm |> "r1"],  
[type |> "Prepared", rm |> "r2"],  
[type |> "Prepared", rm |> "r3"] ]}

291-2

291-3

# 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
```

292-1

# 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
```

292-2

# 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"}]
```

292-3

# 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"],
 [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"],
```

292-4

# 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
```

293-1

# 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
```

293-2

# 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
```

\subseteqq means “subset or equal”

293-3

# 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
```

\subseteqq means “subset or equal”

293-4

# msgs

294-1

# msgs

- The way we will model message transfer is by keeping a global set of all messages ever sent, **msgs**

294-2

# 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

294-3

# 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

294-4

# 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!

294-5

# TPInit

```
TPInit ==
 (*****
 (* The initial predicate. *)
 (*****)
 \wedge rmState = [r \in RM \mapsto "working"]
 \wedge tmState = "init"
 \wedge tmPrepared = {}
 \wedge msgs = {}
)
```

295-1

# TPInit

```
TPInit ==
 (*****
 (* The initial predicate. *)
 (*****)
 \wedge rmState = [r \in RM \mapsto "working"]
 \wedge tmState = "init"
 \wedge tmPrepared = {}
 \wedge msgs = {}
)
```

295-2

# TPInit

```
TPInit ==
 (*****
 (* The initial predicate. *)
 (*****)
 \wedge rmState = [r \in RM \mapsto "working"]
 \wedge tmState = "init"
 \wedge tmPrepared = {}
 \wedge msgs = {}
)
```

Evaluate this expression. What do you get?

295-3

## TPInit

```
TPInit ==
 (*****
 (* The initial predicate.
 *)
 *****)
 \& rmState = [r \in RM |-> "working"]
 \& tmState = "init"
 \& tmPrepared = {}
 \& msgs = {}
```

Evaluate this expression. What do you get?

Value:  
[r1 |-> "working", r2 |-> "working", r3 |-> "working"]

295-4

## TPInit

```
TPInit ==
 (*****
 (* The initial predicate.
 *)
 *****)
 \& rmState = [r \in RM |-> "working"]
 \& tmState = "init"
 \& tmPrepared = {}
 \& msgs = {}
```

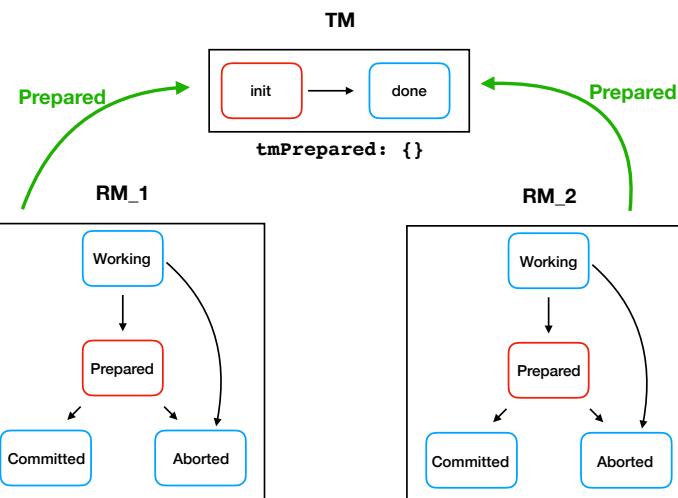
Evaluate this expression. What do you get?

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"

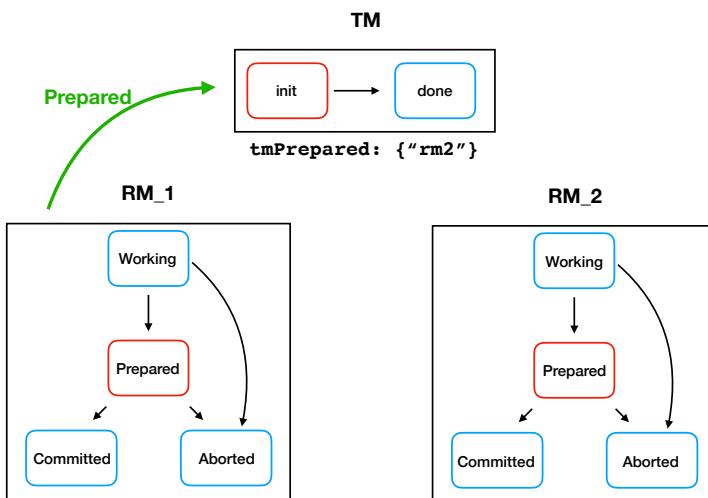
295-5

## TMRcvPrepared(r)



296

## TMRcvPrepared(r)



297

## TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 /\ tmState = "init"
 /\ [type |-> "Prepared", rm |-> r] \in msgs
 /\ tmPrepared' = tmPrepared \union {r}
 /\ UNCHANGED <<rmState, tmState, msgs>>
```

This formula describes the TM receiving a “Prepared” message from a RM

298-1

## TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 /\ tmState = "init"
 /\ [type |-> "Prepared", rm |-> r] \in msgs
 /\ tmPrepared' = tmPrepared \union {r}
 /\ 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?**

298-3

## TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 /\ tmState = "init"
 /\ [type |-> "Prepared", rm |-> r] \in msgs
 /\ tmPrepared' = tmPrepared \union {r}
 /\ 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”**

298-2

## TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 /\ tmState = "init"
 /\ [type |-> "Prepared", rm |-> r] \in msgs
 /\ tmPrepared' = tmPrepared \union {r}
 /\ 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?**

298-4

# 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?**

tmState = “init” means that the TM must be in the “init” state

298-5

# 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

298-6

# TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \union {r}
 \wedge UNCHANGED <<rmState, tmState, msgs>>
```

299-1

# TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \union {r}
 \wedge UNCHANGED <<rmState, tmState, msgs>>
```

299-2

## 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 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"}

299-3

## TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \union {r}
 \wedge UNCHANGED <<rmState, tmState, msgs>>
```

300-1

## 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 is new

300-2

300-3

# 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"

300-4

# TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \cup {r}
 \wedge UNCHANGED <>rmState, tmState, msgs>>
```

301-1

# TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \cup {r}
 \wedge UNCHANGED <>rmState, tmState, msgs>>
```

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
```

300-5

# TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \cup {r}
 \wedge UNCHANGED <>rmState, tmState, msgs>>
```

- Remember, a formula describes the valid pairs of states, i.e., the valid next state given the current state

301-2

# 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

301-3

# 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:

301-4

# 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"
```

301-5

# TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \cup {r}
 \wedge UNCHANGED <<rmState, tmState, msgs>>
```

- 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"
```

301-6

## TMRcvPrepared(r)

```
TMRcvPrepared(r) ==
 \wedge tmState = "init"
 \wedge [type |-> "Prepared", rm |-> r] \in msgs
 \wedge tmPrepared' = tmPrepared \cup {r}
 \wedge UNCHANGED <> rmState, tmState, msgs>>
```

```
rmState: [rm1 |-> "working", rm2 |-> "prepared", rm3 |-> "working"]
msgs: {[type |-> "Prepared", rm |-> "rm2"]}
tmPrepared: {"rm2"}
tmState: "what is going on here"
```

302-1

## 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?

302-2

## 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

302-3

## RMPrepare(r)

```
RMPrepare(r) ==
 \wedge rmState[r] = "working"
 \wedge rmState' = [rmState EXCEPT ![r] = "prepared"]
 \wedge msgs' = msgs \cup {[type |-> "Prepared", rm |-> r]}
 \wedge UNCHANGED <> tmState, tmPrepared>>
```

303-1

## 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”

303-2

## 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
- In the new state:

303-4

## 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

303-3

## 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”

303-5

## RMPREPARE(r)

```
RMPREPARE(r) ==
 \rmState[r] = "working"
 \rmState' = [rmState EXCEPT ![r] = "prepared"]
 \msgs' = msgs \union {[type l-> "Prepared", rm l-> 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

303-6

## 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)
```

## 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

304-2

## 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

304-3

# TPNext

```
TPNext ==
 \vee TMCommit \vee TMAbort
 \vee \exists 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*

304-4

# TPNext

```
TPNext ==
 \vee TMCommit \vee TMAbort
 \vee \exists 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**

304-5

# TPNext

```
TPNext ==
 \vee TMCommit \vee TMAbort
 \vee \exists 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

304-6

# TPNext

```
TPNext ==
 \vee TMCommit \vee TMAbort
 \vee \exists 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

304-7

# TPNext

```
TPNext ==
 \vee TMCommit \vee TMAbort
 \vee \exists 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

304-8

# TPNext

```
TPNext ==
 \vee TMCommit \vee TMAbort
 \vee \exists 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

304-9

# INSTANCE TCommit

# INSTANCE TCommit

- This “imports” all variable declarations, constants, and definitions from the **TCommit** module

305-1

305-2

## INSTANCE TCommit

- This “imports” all variable declarations, constants, and definitions from the **TCommit** module
- This is how we got access to the **TCConsistent** invariant

305-3

## INSTANCE TCommit

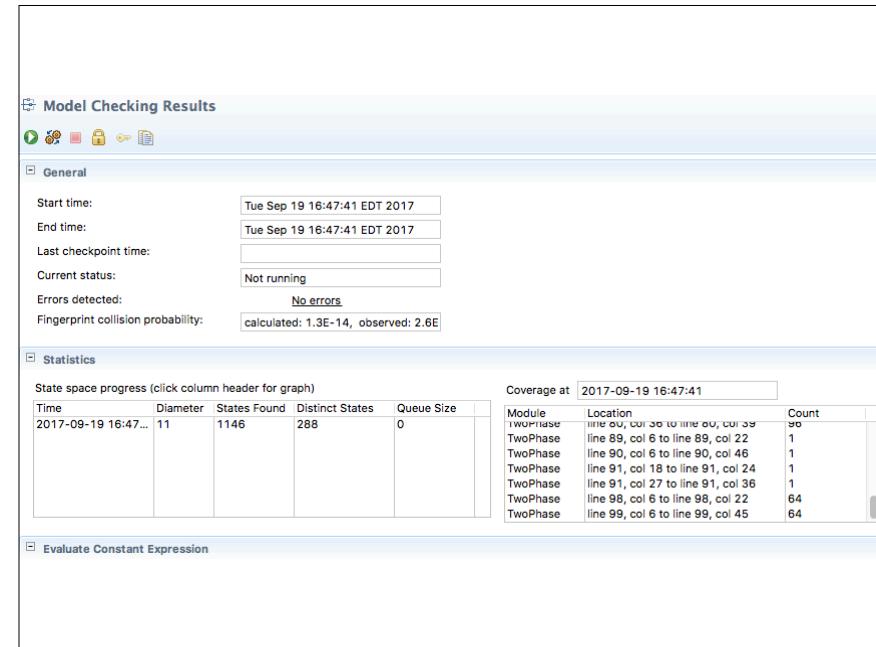
- This “imports” all variable declarations, constants, and definitions from the **TCommit** module
- This is how we got access to the **TCConsistent** invariant
- Because the model checker still passes with the **TCConsistent** invariant enabled, it means that the **TwoPhase** specification “implements” the **TCommit** specification

305-4

## 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

306



307



308

## 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!

309