Precise and Verifiable Distributed and Concurrent System Design using TLA+

Jay Parlar
@parlar
#tla-workshop

Special Thanks

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

Workshop, NOT a lecture

PLEASE Ask Questions

Workshop, NOT a lecture

PLEASE Ask Questions

But please save philosophical pondering until after the workshop

Useful Resources

Useful Resources

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

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

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

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

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/

Goals

Goals

- Introduce the concepts of TLA+

Goals

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

Goals

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

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

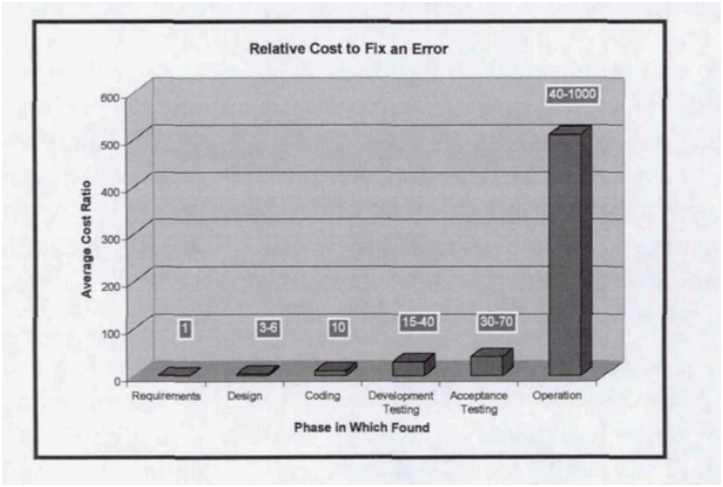
Goals

- 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

Goals

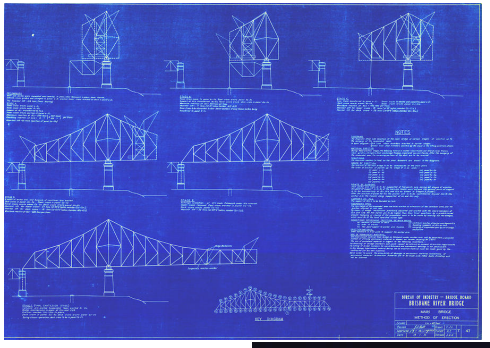
- Introduce the concepts of TLA+
- Introduce a good amount of the syntax of TLA+
- Introduce the model checker, TLC
- Provide a foundation on which you can continue learning
- Convince everyone that thinking **early** is a good thing
- World domination

6-7



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

7



8

```
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 Gf do
        f(u, v) ← 0
        f(v, u) ← 0

    while there exists a path p from s to t in the residual network Gf do
        let cf be the flow capacity of the residual network Gf
        cf(p) ← min{cf(u, v) | (u, v) in p}
        for each edge (u, v) in p do
            f(u, v) ←  f(u, v) + cf(p)
            f(v, u) ← −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

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/

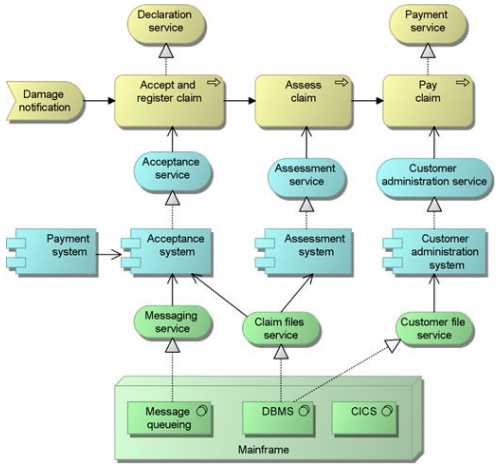
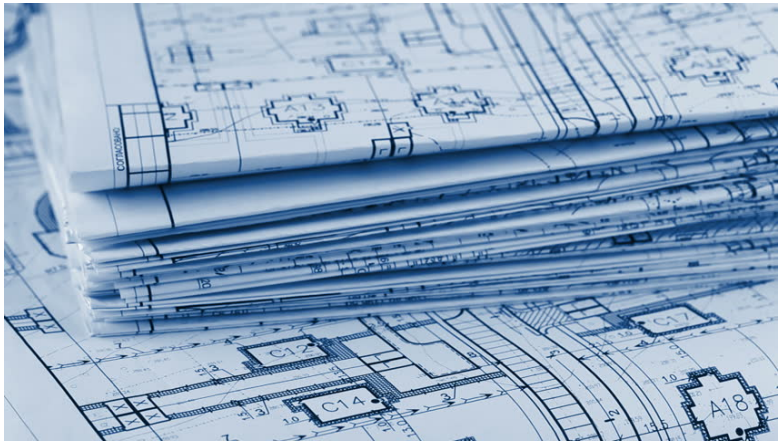


Image via https://en.wikipedia.org/wiki/ArchiMate



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+

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

"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.

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.

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.

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.

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**.

Raft

Raft

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

Raft

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

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.

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+.

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**

Raft

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

Raft

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

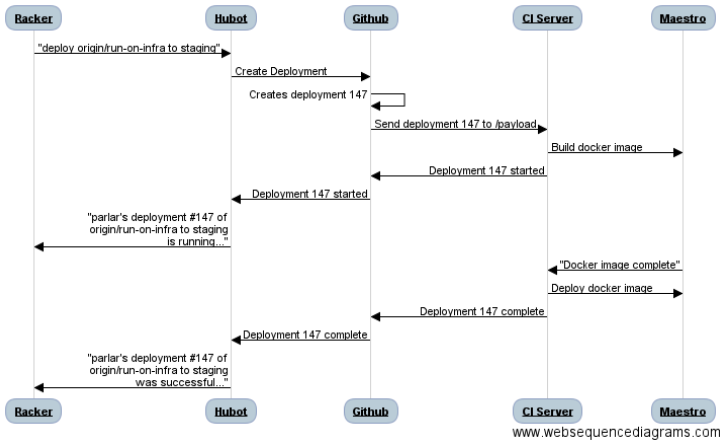

Raft

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


Raft

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


Encore Origin Deployments



```
\* This represents CI receiving a message from Maestro, when Maestro
\* has finished trying to build an image
HandleImageBuildDone(m) ==
    /\ \/ /\ m.type = PRBuildComplete
          /\ CASE m.status = "success"  -> UpdatePR(m.sha, "success", m)
               [] m.status = "error"    -> UpdatePR(m.sha, "error", m)
               [] m.status = "failure"  -> UpdatePR(m.sha, "failure", m)
       \/ /\ m.type = DeployBuildComplete
          /\ CASE m.status = "success"               -> MaestroDeploy(m.sha, m.id, m )
               [] m.status \in {"error", "failure"}  -> UpdateDeployStatus(m.sha, m.id, "failure", m)

   /\ UNCHANGED <<next_id, registry, commit_status, maestro_deploys_complete, gh_deploys_complete>>

\* Represents CI receiving a NewDeployment message from GH
HandleNewDeploy(m) ==
  /\ m.type = NewDeployment
  /\ \/ /\ m.sha \in registry
        /\ MaestroDeploy(m.sha, m.id, m) \* We already have the image, so tell Maestro to go ahead and
     \/ /\ m.sha \notin registry
        /\ BuildDeployImage(m.sha, m.id, m) \* We don't have the image, so tell Maestro to build it
  /\ UNCHANGED <<registry, next_id, commit_status, maestro_deploys_complete, gh_deploys_complete>>


\* Represents CI receiving a deployment success/failure message from Maestro. Only "success" and "fail
HandleDeployComplete(m) ==
    /\ m.type = DeployImageComplete
    /\ \/ /\ m.status = "success"
          /\ UpdateDeployStatus(m.sha, m.id, "success", m)
       \/ /\ m.status = "failure"
          /\ UpdateDeployStatus(m.sha, m.id, "failure", m)

    /\ UNCHANGED <<registry, next_id, commit_status, maestro_deploys_complete, gh_deploys_complete>>

\* Represents GH receiving a commit_status status update from CI
HandlePRStatusUpdate(m) ==
  /\ m.type = PRStatus
  /\ commit_status' = [commit_status EXCEPT ![m.sha] = m.status]
  /\ Discard(m)
```

22

Deployment Pipeline

23-1

Deployment Pipeline

- “Only deploy if an image exists”

23-2

Deployment Pipeline

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

23-3

Deployment Pipeline

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

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”

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.


```
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
```

```python
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)
```


PlusCal

PlusCal

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

PlusCal

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

PlusCal

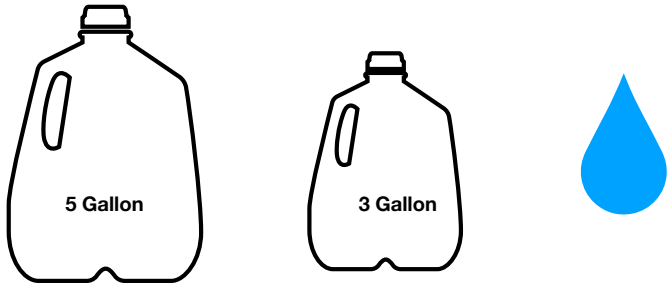
- A programming language-like layer on top of TLA+
- Some people find it easier to read and write than TLA+
- TLC compiles PlusCal code into TLA+

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

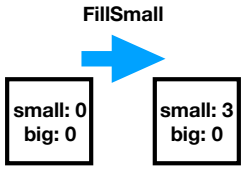
How do we measure out exactly 4 gallons?

i.e. “The Die Hard Problem”



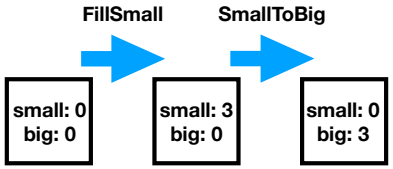
One Possible Execution

One Possible Execution



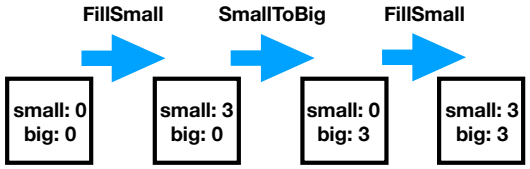
28-2

One Possible Execution



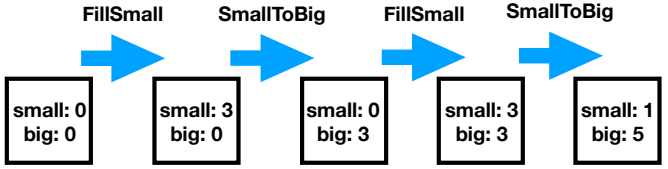
28-3

One Possible Execution



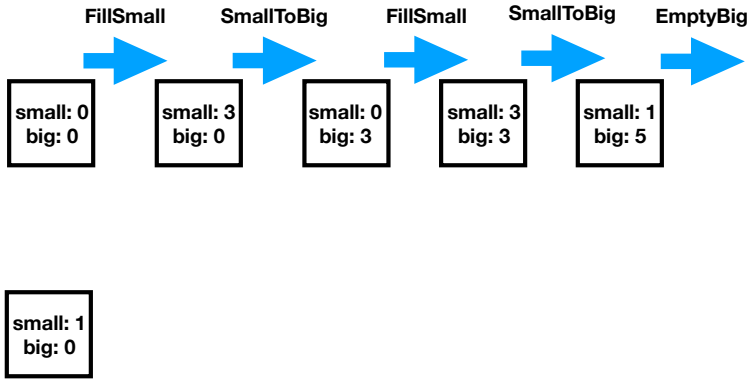
28-4

One Possible Execution



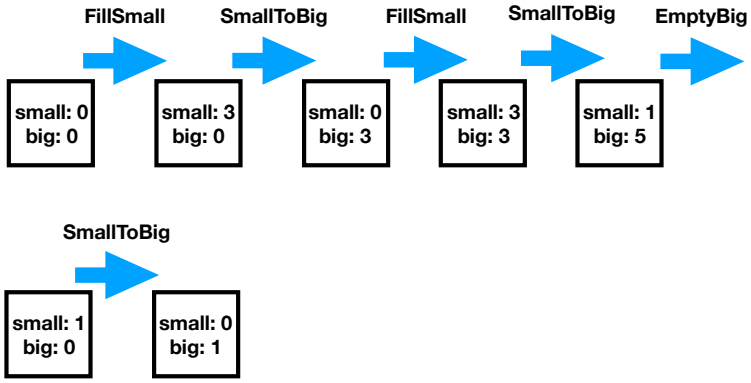
28-5

One Possible Execution



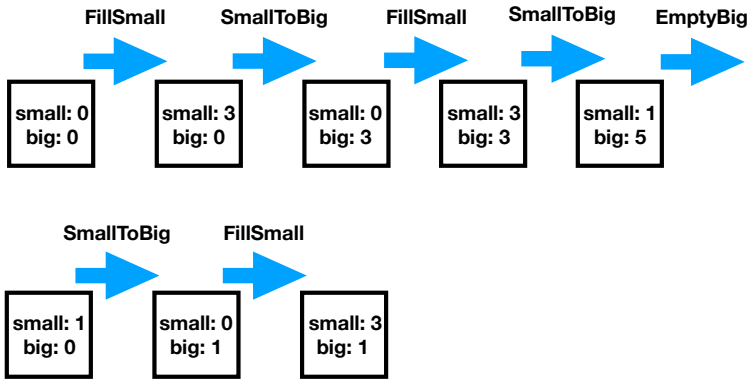
28-6

One Possible Execution



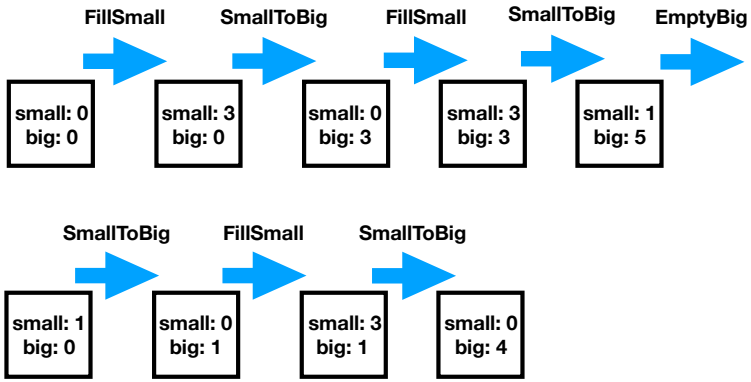
28-7

One Possible Execution



28-8

One Possible Execution

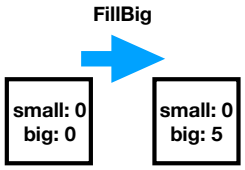


28-9

Another Possible Execution

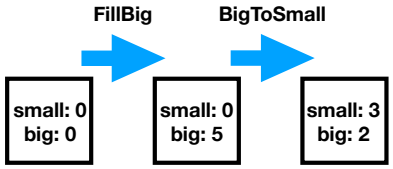
29-1

Another Possible Execution



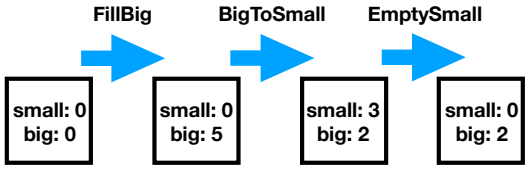
29-2

Another Possible Execution



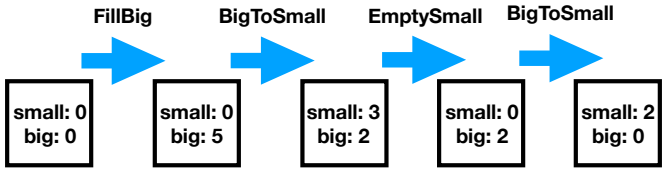
29-3

Another Possible Execution



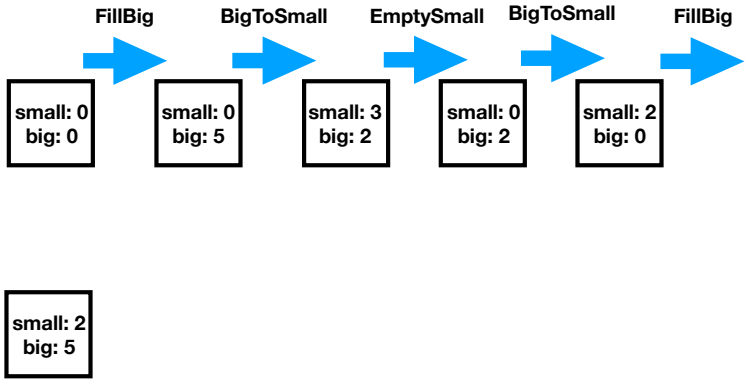
29-4

Another Possible Execution



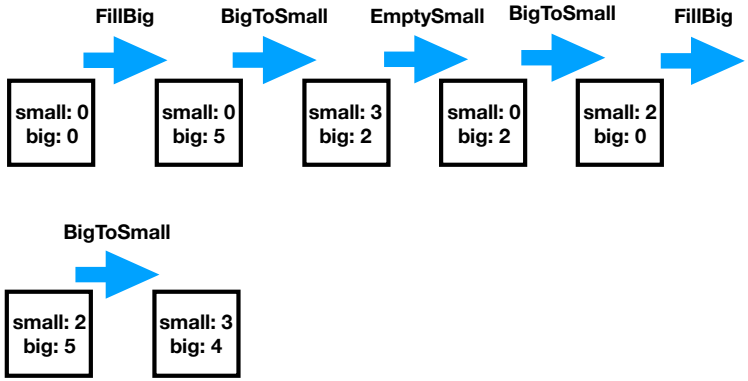
29-5

Another Possible Execution



29-6

Another Possible Execution

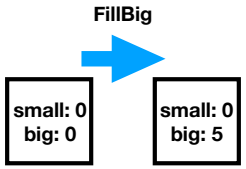


29-7

Yet Another Possible Execution

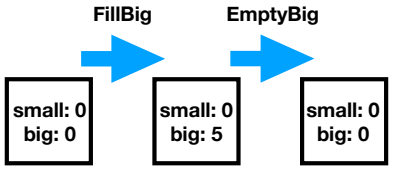
30-1

Yet Another Possible Execution



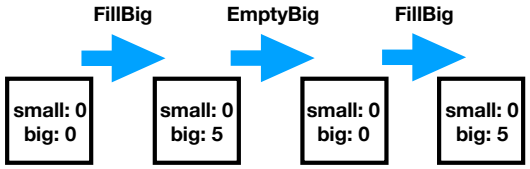
30-2

Yet Another Possible Execution



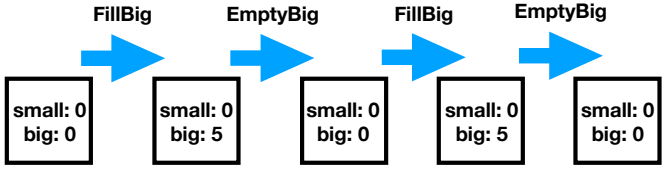
30-3

Yet Another Possible Execution



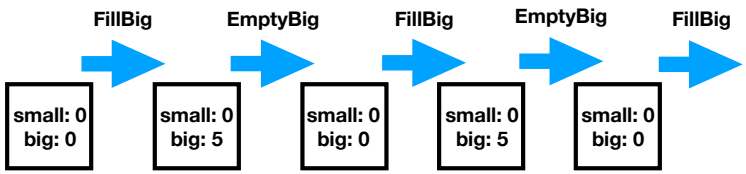
30-4

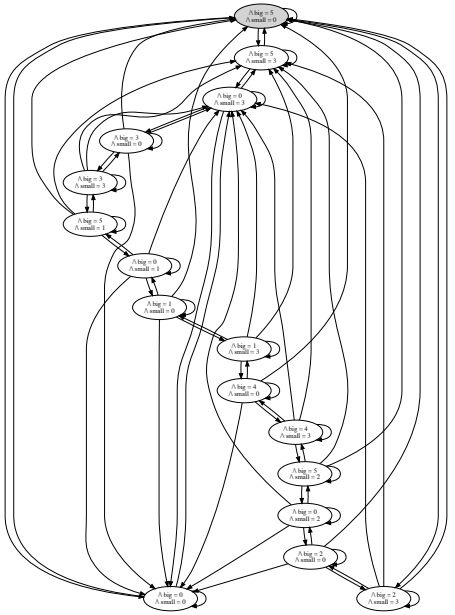
Yet Another Possible Execution



30-5

Yet Another Possible Execution





TLA+ Terms

TLA+ Terms

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

TLA+ Terms

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

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.

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+ 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.

How could we describe digital systems (behaviours)?

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

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**

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**

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**

Die Hard State Machine

Die Hard State Machine

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

Die Hard State Machine

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

Die Hard State Machine

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

Die Hard Next States

Die Hard Next States

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

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**

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**

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:

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

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)

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)

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

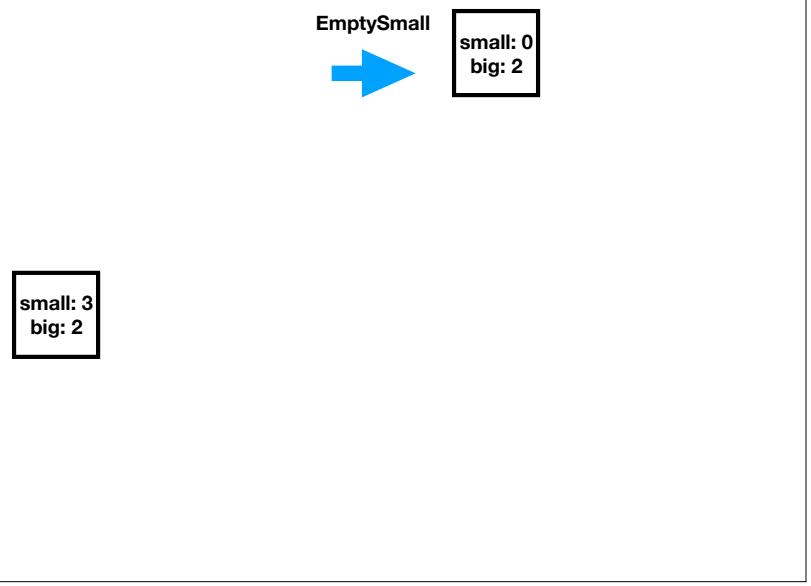
ELSE big = 5; small = small - (5 - big)

etc. etc.

Possible next states

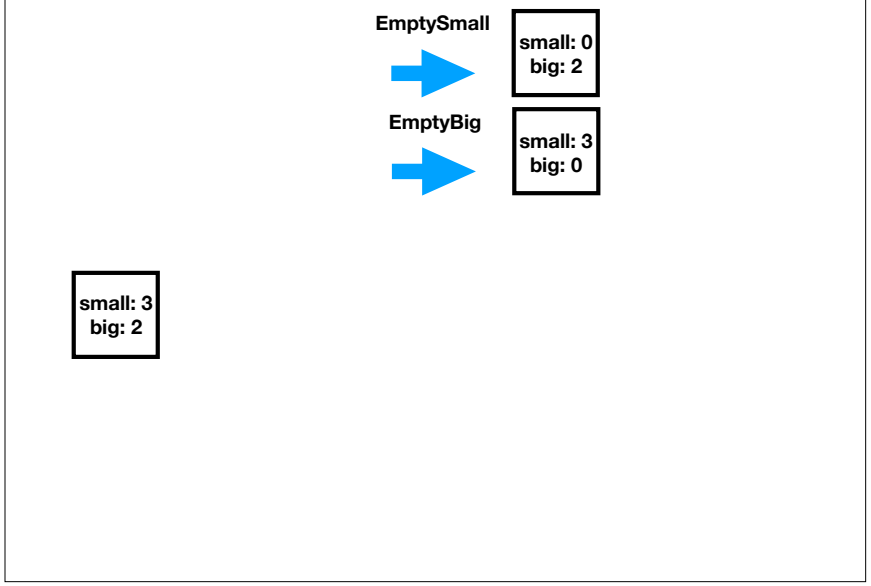
small: 3
big: 2

Possible next states



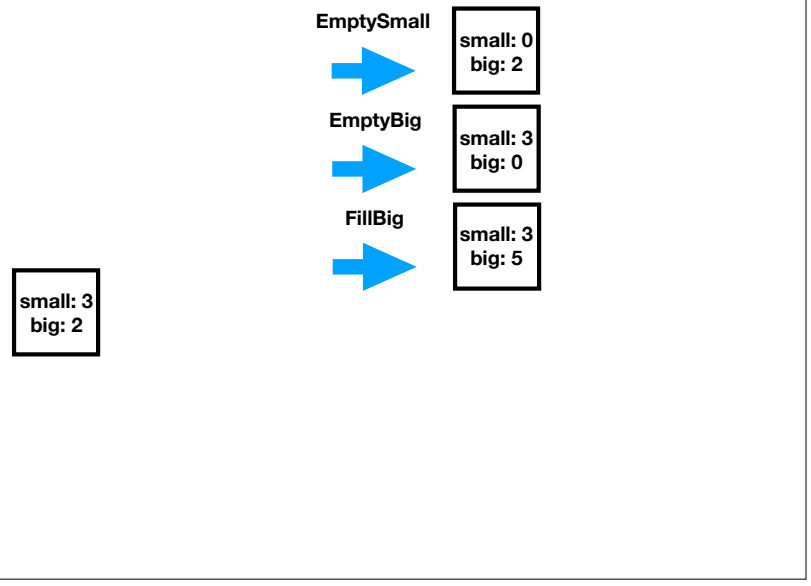
37-2

Possible next states



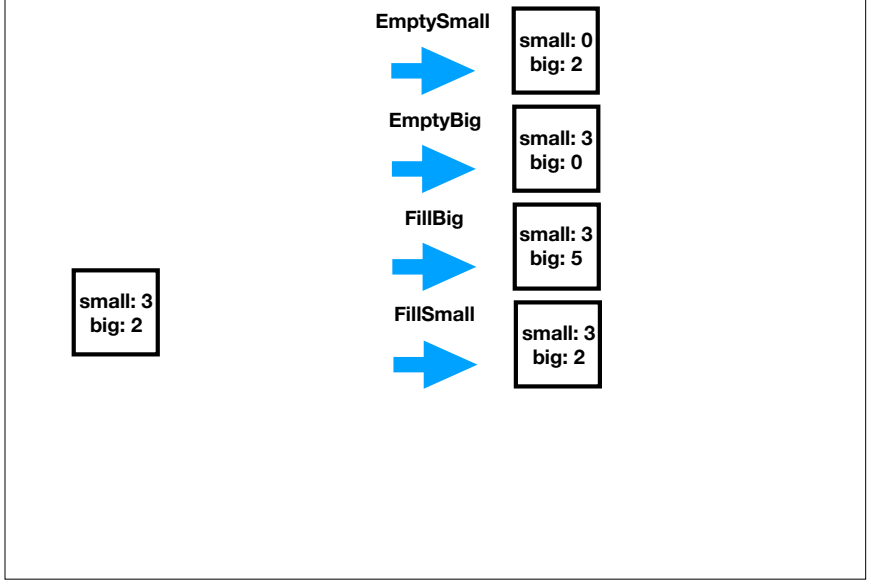
37-3

Possible next states



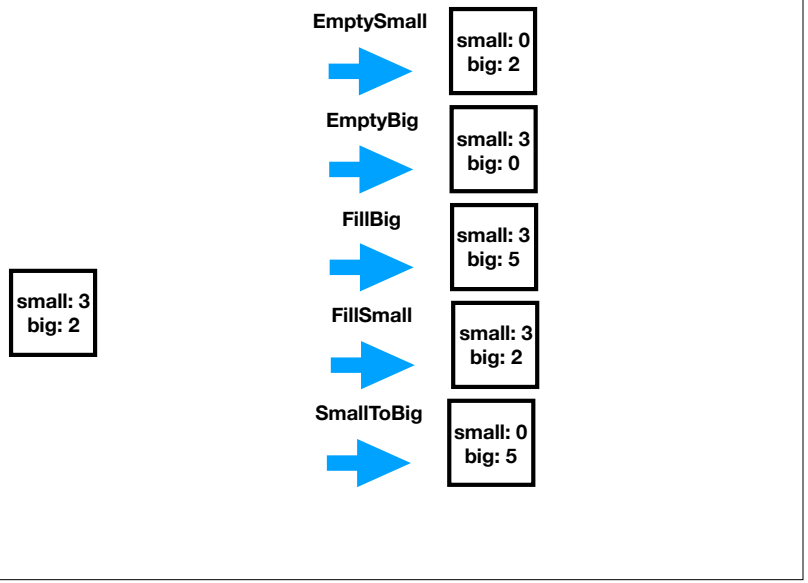
37-4

Possible next states

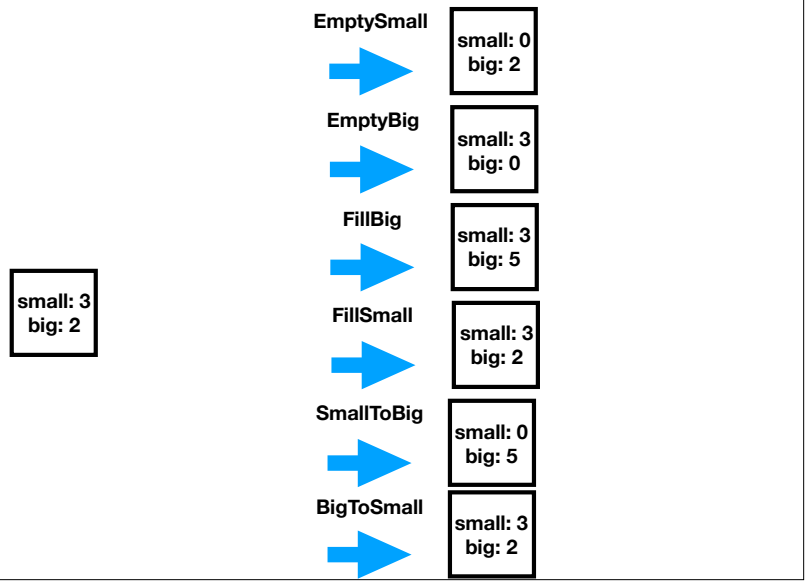


37-5

Possible next states



Possible next states




```
VARIABLES small, big

TypeOK == /\ small \in 0..3
          /\ big   \in 0..5

Init == /\ big   = 5
        /\ small = 0

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

FillBig == /\ big'   = 5
           /\ small' = small

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

EmptyBig == /\ big'   = 0
            /\ small' = small

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

BigToSmall == IF big + small =< 3
              THEN /\ big'   = 0
                   /\ small' = big + small
              ELSE /\ big'   = small - (3 - big)
                   /\ small' = 3

Next == \/ FillSmall  \/ FillBig
        \/ EmptySmall \/ EmptyBig
        \/ SmallToBig \/ BigToSmall
```


Why use state machines?

Why use state machines?

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

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:

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

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

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

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

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.

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.

Math Time!!!

“and” / “or”

“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
```

= and ≜

= and ≜

= 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 **==.**

= and ≜

= 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 **==.**

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

= and ≜

= 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 **==.**

≜ means “defined to be”. It is written as **==** in ASCII. You use it to created 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.

Exercise

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

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

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


Exercise

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

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

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


Exercise

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

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

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


Exercise

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

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

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

TLA + Syntax

TLA+ specs make heavy use of formulas like this

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


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.

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)
```


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

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 /\ or \/ symbols

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 /\ or \/ symbols

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 /\ or \/ symbols

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 /\ or \/ symbols

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 /\ or \/ symbols

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 /\ or \/ symbols

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 /\ or \/ symbols

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 /\ or \/ symbols

TLA+ Syntax

In general:

TLA+ Syntax

In general:

```
A /\ B /\ C /\ … /\ N      =      /\ A
                                  /\ B
                                  /\ C
                                  …
                                  /\ N
```


TLA+ Syntax

In general:

```
A /\ B /\ C /\ … /\ N      =      /\ A
                                  /\ B
                                  /\ C
                                  …
                                  /\ N
```

```
A \/ B \/ C \/ … \/ N      =      \/ A
                                  \/ B
                                  \/ C
                                  …
                                  \/ N
```


TLA+ Syntax

In general:

```
A /\ B /\ C /\ … /\ N      =      /\ A
                                  /\ B
                                  /\ C
                                  …
                                  /\ N
```

```
A \/ B \/ C \/ … \/ N      =      \/ A
                                  \/ B
                                  \/ C
                                  …
                                  \/ N
```

TLA+ Syntax

In general:

```
A /\ B /\ C /\ … /\ N   =   /\ A
                            /\ B
                            /\ C
                            …
                            /\ N
```

```
A \/ B \/ C \/ … \/ N   =   \/ A
                            \/ B
                            \/ C
                            …
                            \/ N
```


TLA+ Syntax

In general:

```
A /\ B /\ C /\ … /\ N   =   /\ A
                            /\ B
                            /\ C
                            …
                            /\ N
```

```
A \/ B \/ C \/ … \/ N   =   \/ A
                            \/ B
                            \/ C
                            …
                            \/ N
```


TLA+ Syntax

In general:

```
A /\ B /\ C /\ … /\ N   =   /\ A
                            /\ B
                            /\ C
                            …
                            /\ N
```

```
A \/ B \/ C \/ … \/ N   =   \/ A
                            \/ B
                            \/ C
                            …
                            \/ N
```


TLA+ Syntax

TLA+ does not allow “ambiguous” formulas:

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

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.

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)
```


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.)

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)
```

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.

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.

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.

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.

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.

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 /\ at the outermost indent level.

TLA+ Syntax

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


TLA+ Syntax

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

/\ A
```

TLA+ Syntax


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


TLA+ Syntax


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


TLA+ Syntax


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


TLA+ Syntax


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

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

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

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

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

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

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.

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

TLA+ Syntax

We’ll create indent levels for each of these

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


TLA+ Syntax

We’ll create indent levels for each of these

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

```
/\ A
```

TLA+ Syntax

We'll create indent levels for each of these

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

/\ A
/\ \/ B
```


TLA+ Syntax

We'll create indent levels for each of these

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

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


TLA+ Syntax

We'll create indent levels for each of these

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

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


TLA+ Syntax

We'll create indent levels for each of these

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

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

TLA+ Syntax

We'll create indent levels for each of these

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

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


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
```


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
```


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
```

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 .

TLA+ Syntax

What about this? How would this look?

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


TLA+ Syntax

What about this? How would this look?

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

TLA+ does not allow this. Precedence between ∨ and ∧ must be made explicit with parentheses.

TLA+ Syntax

What about this? How would this look?

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

TLA+ does not allow this. Precedence between ∨ and ∧ must be made explicit with parentheses.

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

TLA+ Syntax

What about this? How would this look?

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

TLA+ does not allow this. Precedence between ∨ and ∧ must be made explicit with parentheses.

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

Which becomes

50-4

TLA+ Syntax

What about this? How would this look?

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

TLA+ does not allow this. Precedence between ∨ and ∧ must be made explicit with parentheses.

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

Which becomes

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

50-5

TLA+ Syntax

What about this? How would this look?

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

TLA+ does not allow this. Precedence between ∨ and ∧ must be made explicit with parentheses.

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

Which becomes

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

50-6

TLA+ Syntax

What about this? How would this look?

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

TLA+ does not allow this. Precedence between ∨ and ∧ must be made explicit with parentheses.

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

Which becomes

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

50-7

TLA+ Syntax

What about this? How would this look?

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

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

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

Which becomes

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

50-8

TLA+ Syntax

What about this? How would this look?

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

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

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

Which becomes

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

50-9

TLA+ Syntax

What about this? How would this look?

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

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

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

Which becomes

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

50-10

TLA+ Syntax

What about this? How would this look?

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

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

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

Which becomes

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

50-11

Exercise

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

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

3. (A \/ (B /\ C) \/ D) /\ E /\ F /\ (G \/ H \/ (I /\ J))

Solutions

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

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

Solutions

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

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

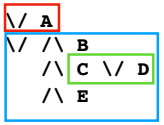
Solutions

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

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

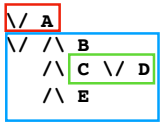
Solutions

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

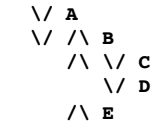


Solutions

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

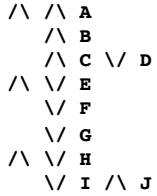


Alternate Solution



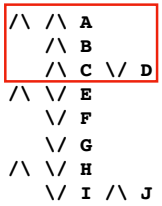
Solutions

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



Solutions

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



Solutions

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

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


Solutions

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

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


Solutions

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

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


Solutions

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

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

Solutions

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

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

Alternate Solution

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


Solutions

3. (A \/ (B /\ C) \/ D) /\ E /\ F /\ (G \/ H \/ (I /\ J))

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


Solutions

3. (A \/ (B /\ C) \/ D) /\ E /\ F /\ (G \/ H \/ (I /\ J))

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


Solutions

3. (A \/ (B /\ C) \/ D) /\ E /\ F /\ (G \/ H \/ (I /\ J))

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

Solutions

3. (A \/ (B /\ C) \/ D) /\ E /\ F /\ (G \/ H \/ (I /\ J))

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


Solutions

3. (A \/ (B /\ C) \/ D) /\ E /\ F /\ (G \/ H \/ (I /\ J))

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


Solutions

3. (A \/ (B /\ C) \/ D) /\ E /\ F /\ (G \/ H \/ (I /\ J))

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


Solutions

3. (A \/ (B /\ C) \/ D) /\ E /\ F /\ (G \/ H \/ (I /\ J))

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

Solutions

3. (A \/ (B /\ C) \/ D) /\ E /\ F /\ (G \/ H \/ (I /\ J))

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

Alternate Solution

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


Sets

Sets

- An unordered collection of unique things

Sets

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

Sets

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

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”}

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)

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()`

Sets

Sets

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

Sets

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

Sets

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

Sets

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

Sets

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

Sets

Sets

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

Sets

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

Sets

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

Sets

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

Sets

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

Sets

Sets

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

Sets

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

Sets

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

Sets

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

Sets

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

Sets

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

Sets

Sets

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

Sets

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

Sets

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

Sets

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

Sets

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

Sets

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

Sets

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

Sets

Sets

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

Sets

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

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))`**

Exercises

(TRUE or FALSE for each)

{“a”, “b”} ⊆ {“b”, “a”, “c”}

{1,2,3} ⊆ ({1,2,3} ∩ {2,3})

“a” ∈ (({“b”, “c”, “d”} ∩ {“e”, “f”}) ∪ {“a”})

(“a” ∈ {“a”, “b”}) ∨ (“b” ∈ {“c”, “d”})

Exercises

(TRUE or FALSE for each)

{“a”, “b”} ⊆ {“b”, “a”, “c”}

{1,2,3} ⊆ ({1,2,3} ∩ {2,3})

“a” ∈ (({“b”, “c”, “d”} ∩ {“e”, “f”}) ∪ {“a”})

(“a” ∈ {“a”, “b”}) ∨ (“b” ∈ {“c”, “d”})

Exercises

(TRUE or FALSE for each)

{“a”, “b”} ⊆ {“b”, “a”, “c”} TRUE

{1,2,3} ⊆ ({1,2,3} ∩ {2,3})

“a” ∈ (({“b”, “c”, “d”} ∩ {“e”, “f”}) ∪ {“a”})

(“a” ∈ {“a”, “b”}) ∨ (“b” ∈ {“c”, “d”})

Exercises

(TRUE or FALSE for each)

{“a”, “b”} ⊆ {“b”, “a”, “c”} TRUE

{1,2,3} ⊆ ({1,2,3} ∩ {2,3}) FALSE

“a” ∈ (({“b”, “c”, “d”} ∩ {“e”, “f”}) ∪ {“a”}) TRUE

(“a” ∈ {“a”, “b”}) ∨ (“b” ∈ {“c”, “d”})

Exercises

(TRUE or FALSE for each)

{“a”, “b”} ⊆ {“b”, “a”, “c”} TRUE

{1,2,3} ⊆ ({1,2,3} ∩ {2,3}) FALSE

“a” ∈ (({“b”, “c”, “d”} ∩ {“e”, “f”}) ∪ {“a”}) TRUE

(“a” ∈ {“a”, “b”}) ∨ (“b” ∈ {“c”, “d”}) TRUE

“There exists”

“There exists”

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

“There exists”

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

“There exists”

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

“There exists”

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

“There exists”

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

“There exists”

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

“There exists”

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

“There exists”

- ∃ means “there exists”, written as **\E** in ASCII
- The usual form is ∃ x ∈ 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

“There exists”

- ∃ means “there exists”, written as **\E** in ASCII
- The usual form is ∃ x ∈ 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
  - ∃ x∈{1,2,3,4} : x > 3 is TRUE

“There exists”

- ∃ means “there exists”, written as **\E** in ASCII
- The usual form is ∃ x ∈ 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
  - ∃ x∈{1,2,3,4} : x > 3 is TRUE
  - ∃ x∈{1,2,3,4} : x > 5 is FALSE

“There exists”

“There exists”

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

is roughly equivalent to the following Python expressions

“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
```


“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
```


“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
```

“For all”

“For all”

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

“For all”

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

“For all”

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

“For all”

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

“For all”

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

“For all”

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

“For all”

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

“For all”

- ∀ means “for all”, written as **\A** in ASCII
- The usual form is ∀ x ∈ 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

“For all”

- ∀ means “for all”, written as **\A** in ASCII
- The usual form is ∀ x ∈ 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
  - ∀ x∈{1,2,3,4} : x > 3 is FALSE

“For all”

- ∀ means “for all”, written as **\A** in ASCII
- The usual form is ∀ x ∈ 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
  - ∀ x∈{1,2,3,4} : x > 3 is FALSE
  - ∀ x∈{1,2,3,4} : x > 0 is TRUE

“For all”

“For all”

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

is roughly equivalent to the following Python expressions

“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”

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

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
```


“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
```

Exercises

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

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

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

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

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

Exercises

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Set constructors

Set constructors

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

Set constructors

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

{ x ∈ {1,2,3,4,5} : x > 3 } = {4,5}

Set constructors

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

{ x ∈ {1,2,3,4,5} : x > 3 } = {4,5}

This is equivalent to the following Python expressions:

Set constructors

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

{ x ∈ {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 constructors

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

{ x ∈ {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]))
```

Set constructors

Set constructors

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

{ x * 2 : x ∈ {1,2,3,4,5}} = {2,4,6,8,10}

Set constructors

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

{ x * 2 : x ∈ {1,2,3,4,5}} = {2,4,6,8,10}

This is roughly equivalent to the following Python expressions:

Set constructors

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

{ x * 2 : x ∈ {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 constructors

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

{ x * 2 : x ∈ {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]))
```


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}**).

Solutions

Solutions

1. { x ∈ {1,2,3,4,5} : ∧ x > 1 ∧ x < 5 } **or** { x ∈ {1,2,3,4,5} : x > 1 ∧ x < 5 }

Solutions

1. { x ∈ {1,2,3,4,5} : ∧ x > 1 ∧ x < 5 } **or** { x ∈ {1,2,3,4,5} : x > 1 ∧ x < 5 }

2. { x ∈ {1,2,3,4,5} : x*3 > 10}

70-3

Solutions

1. { x ∈ {1,2,3,4,5} : ∧ x > 1 ∧ x < 5 } **or** { x ∈ {1,2,3,4,5} : x > 1 ∧ x < 5 }

2. { x ∈ {1,2,3,4,5} : x*3 > 10}

3. { x+x-1 : x ∈ {1,2,3,4,5}}

70-4

71

Back to State Machines

72-1

Back to State Machines

Recall that a state machine is defined by three things:

Back to State Machines

Recall that a state machine is defined by three things:

1. The **variables**

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

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**)

A small C program

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

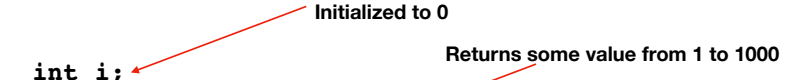

A small C program


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


A small C program


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

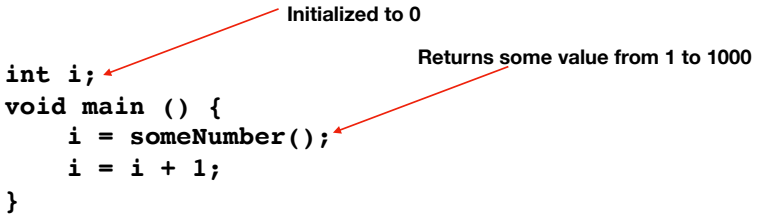

A small C program




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

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

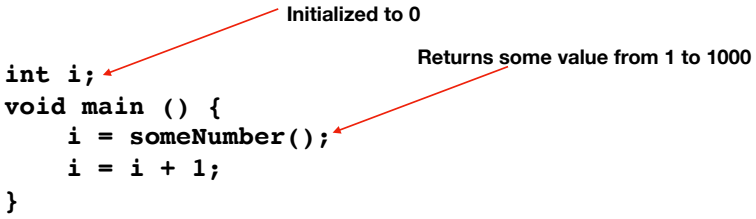
A small C program



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

- Variables: **i**

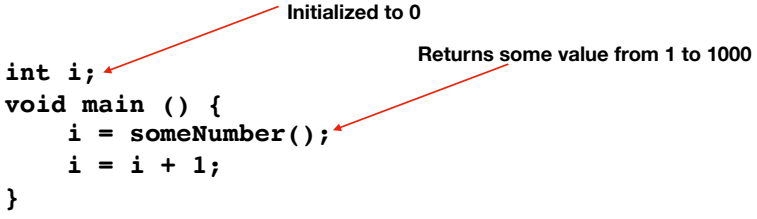
A small C program



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

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

A small C program



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!

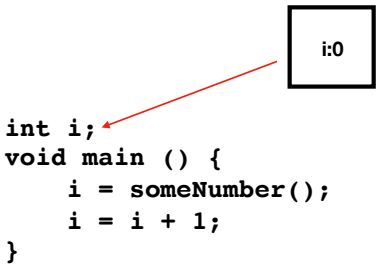
A small C program - example behaviour

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

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

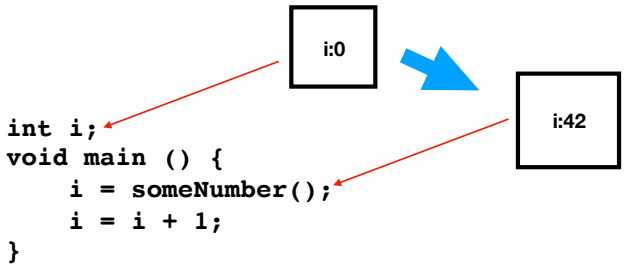
A small C program - example behaviour

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



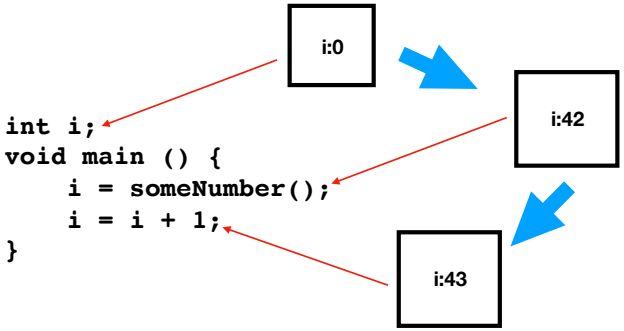
A small C program - example behaviour

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



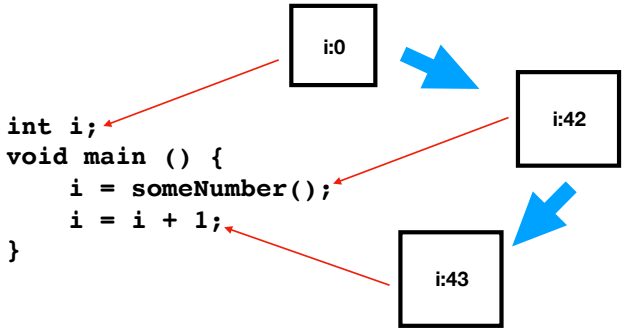
A small C program - example behaviour

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



A small C program - example behaviour

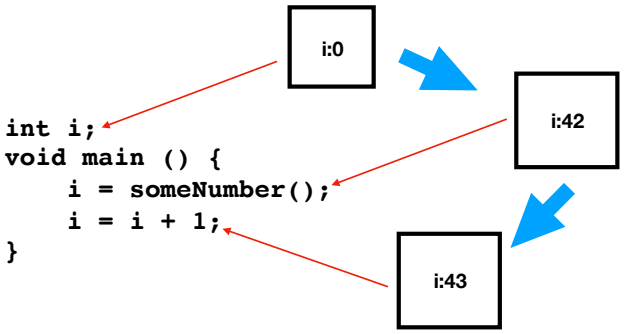
Assume that for this particular *behaviour,* **`someNumber()`** returns **42**.



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

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”.

A small C program - example behaviour

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

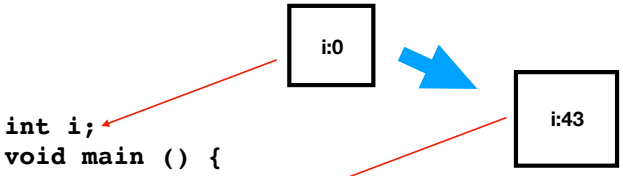
```
int i;
void main () {
    i = someNumber();
    i = i + 1;
}
```


A small C program - example behaviour

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

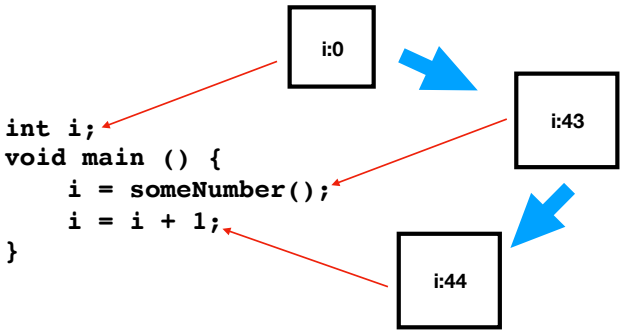
A small C program - example behaviour

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



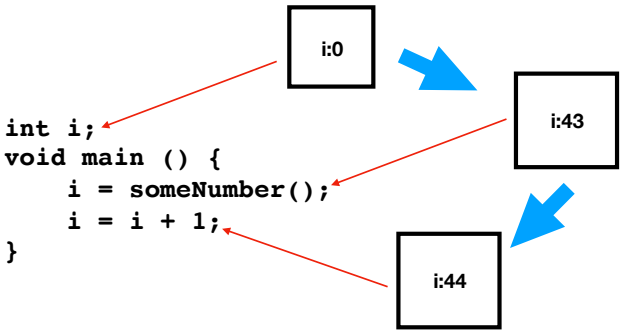
A small C program - example behaviour

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



A small C program - example behaviour

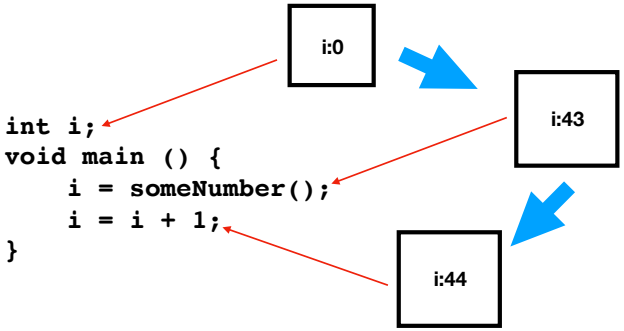
Now let's look at a behaviour where **`someNumber()`** returned **43**.



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

A small C program - example behaviour

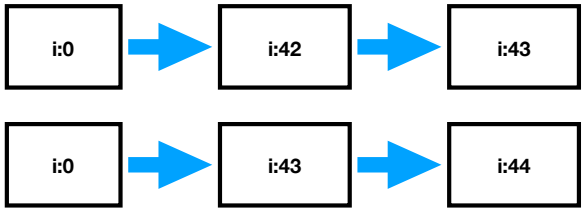
Now let's look at a behaviour where **`someNumber()`** returned **43**.



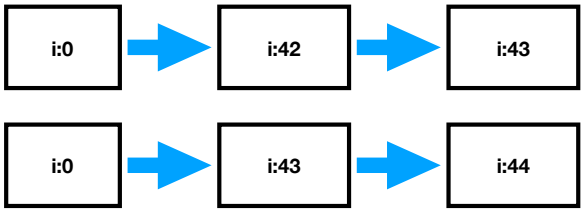
In this case, a state of **[i:43]** has a next state of **[i:44]**.

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

43



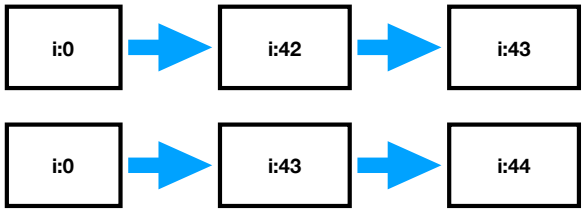
43



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

76-2

43

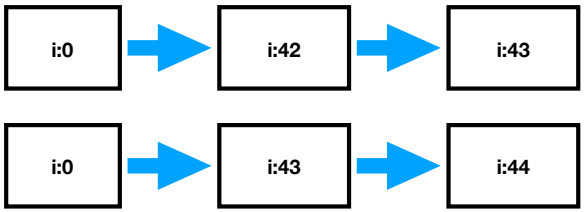


Thus representing the state of the program with just the variable **i** is not sufficient.

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**

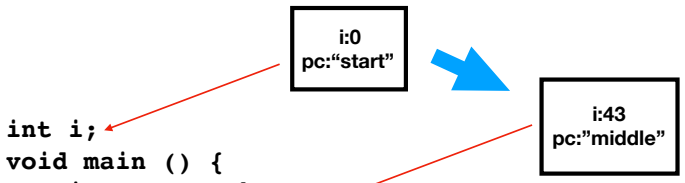
We need a second variable, **pc**, to represent "control state", i.e. where in the program we are


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

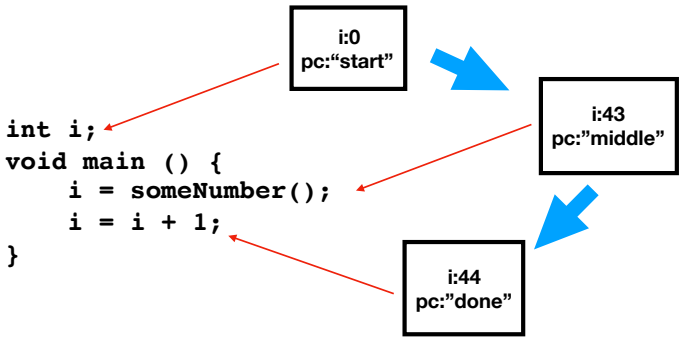



```
int i;
void main () {
    i = someNumber();
    i = i + 1;
}
```


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




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


A small C program

Let's make this a bit more formal

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


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;
}
```


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;
}
```


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
```

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
```


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
```


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
```


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
```

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
```


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
```

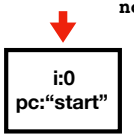

A small C program

```
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"

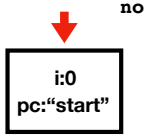
A small C program

```
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
```



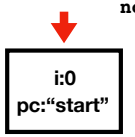
A small C program

```
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
```



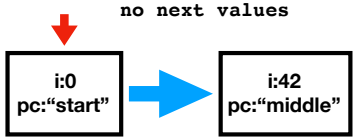
A small C program

```
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
```



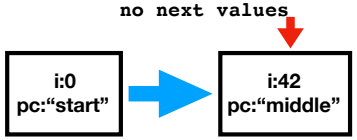
A small C program

```
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
```



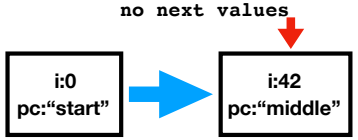
A small C program

```
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
```



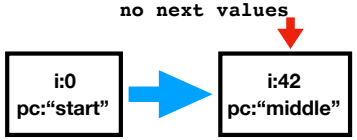
A small C program

```
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
```



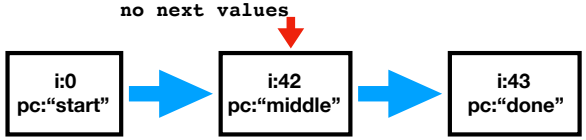
A small C program

```
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
```



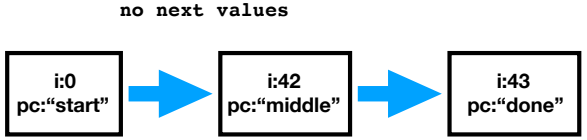
A small C program

```
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
```



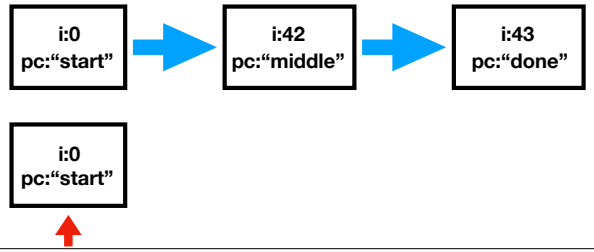
A small C program

```
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
```



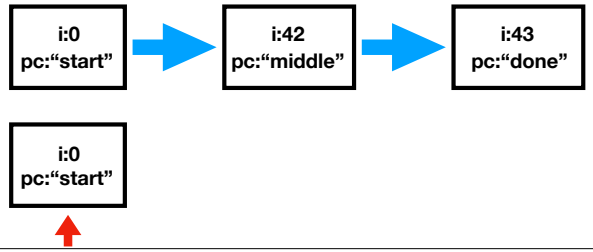
A small C program

```
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
```



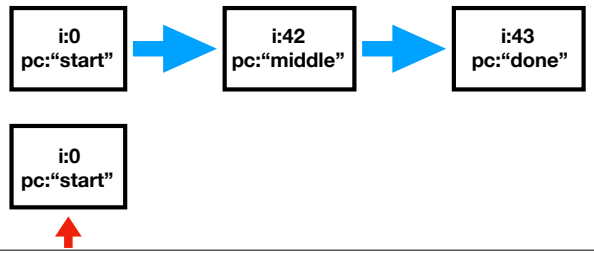
A small C program

```
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
```



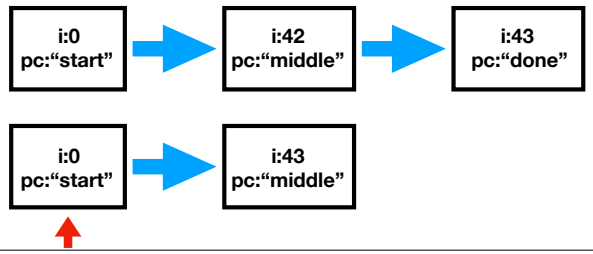
A small C program

```
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
```



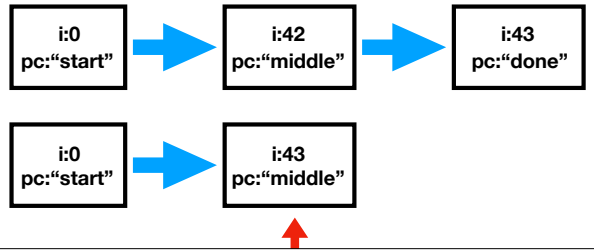
A small C program

```
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
```



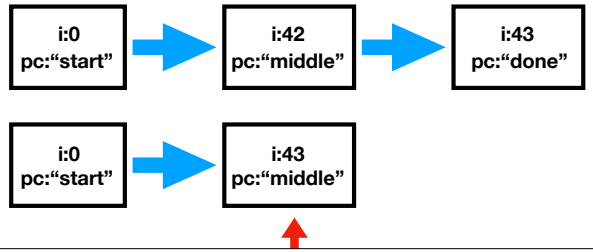
A small C program

```
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
```



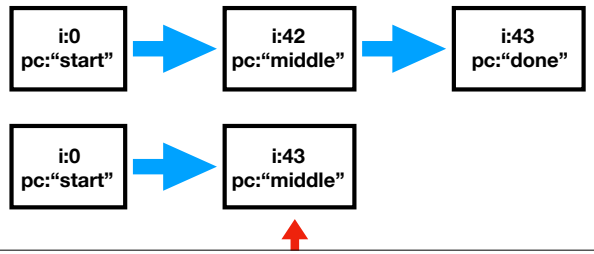
A small C program

```
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
```



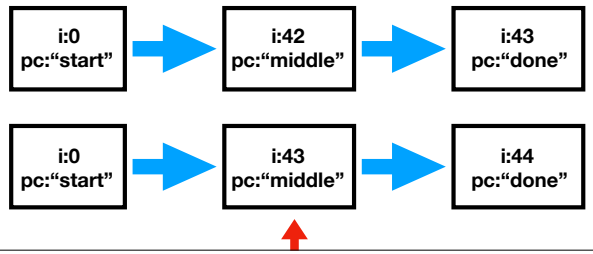
A small C program

```
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
```



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
```



Improving the next state formula

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
```


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
```


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

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

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
```


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
```


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

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

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

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
```


= doesn't mean assignment!!!

= 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"

= 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
```


= 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
```


= 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
```

= doesn't mean assignment!!!

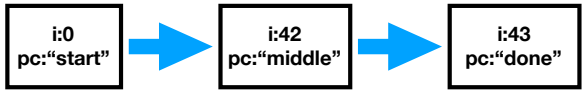
This means:
"If in the current state pc is equal to "start", then a valid next state is any state in which i is any value between 1 and 1000, and pc is the string "middle"

```
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.

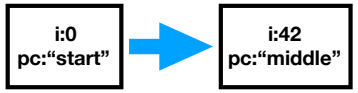
State pairs

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



State pairs

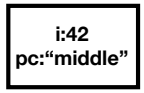
```
if pc = "start"
  then
    i' ∈ 1..1000
    pc' = "middle"
  else if pc = "middle"
    then
      i' = i + 1
      pc' = "done"
    else
      no next values
```



Current state **Next state**

State pairs

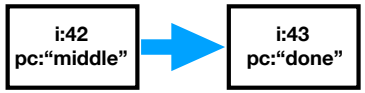
```
if pc = "start"
  then
    i' ∈ 1..1000
    pc' = "middle"
  else if pc = "middle"
    then
      i' = i + 1
      pc' = "done"
    else
      no next values
```



Current state **Next state**

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

State pairs

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:

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**.

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.

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**.

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).

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!

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
```


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

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

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

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
```


A little bit more formal

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

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

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
```

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

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
```


Looking pretty good…

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

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
```

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?

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

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

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

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' ∈ 1..1000) /\ (pc' = "middle")** in the next state

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' ∈ 1..1000) /\ (pc' = "middle")** in the next state

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' ∈ 1..1000) /\ (pc' = "middle")** in the next state

Or in other words:

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' ∈ 1..1000) /\ (pc' = "middle")** in the next state

Or in other words:

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

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' ∈ 1..1000) /\ (pc' = "middle")** in the next state

Or in other words:

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

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' ∈ 1..1000) /\ (pc' = "middle")** in the next state

Or in other words:

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

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' ∈ 1..1000) /\ (pc' = "middle")** in the next state

Or in other words:

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

Case 2

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


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

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

Case 2

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

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

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

Case 2

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

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

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

Case 2

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

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

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

Or in other words:

Case 2

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

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

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

Or in other words:

```
(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
```

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")))
```


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")))
```

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.)

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
```

Comparison

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


Comparison

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

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


Notes

Notes

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

Notes

- 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.

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.

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”.

An aside

An aside

- Specifying at this level is incredibly rare.

An aside

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

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.

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.

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.

Next State

Next State

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


Next State

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

- This formula defines all the valid state pairs.

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'** ".

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?

Behaviours

Behaviours

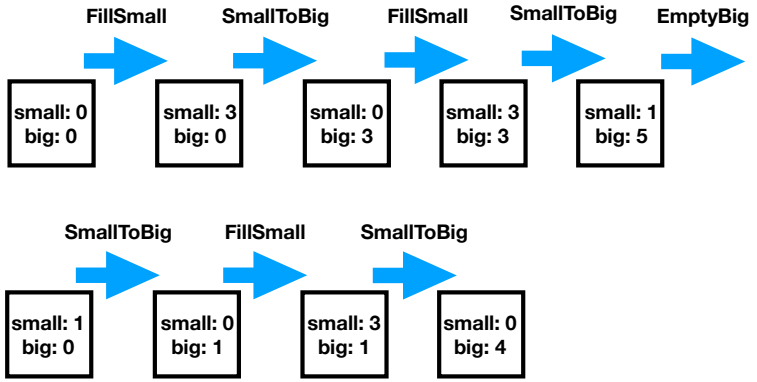
- Think back to our Die Hard example

Behaviours

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

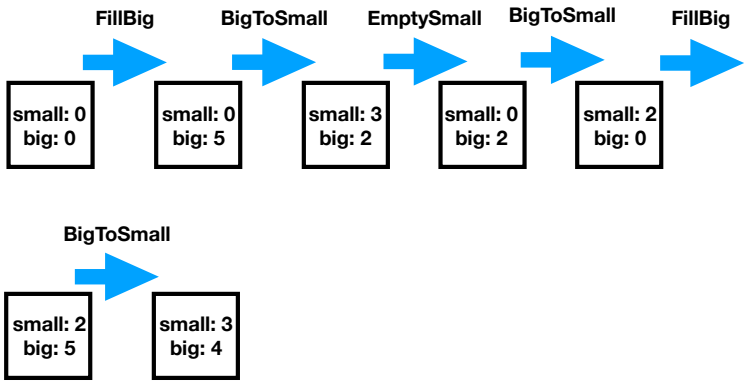
Behaviours

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



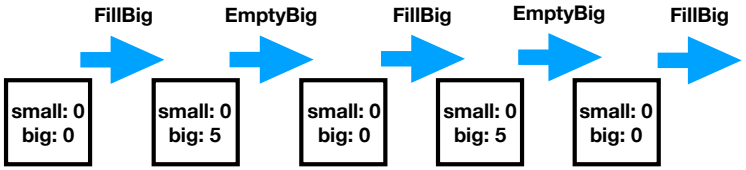
Behaviours

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



Behaviours

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



Behaviours

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

- An INFINITE number of behaviours are possible

Behaviours

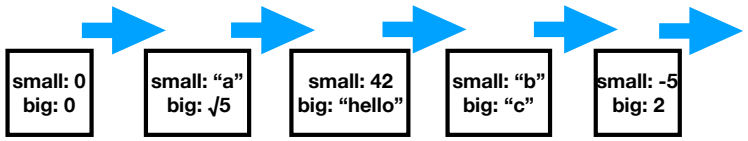
- Think back to our Die Hard example
- We showed three possible behaviours

- 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**

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**



Behaviours

Behaviours

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

110-2

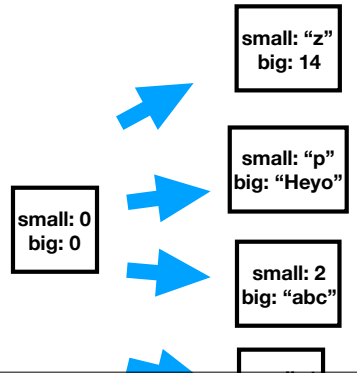
Behaviours

- 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+.

Behaviours

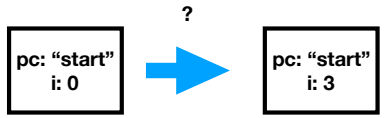
- The universe presents an infinite combination of behaviours, where an infinite number of **values** can be assigned to our **variables**.
- From any **state**, an infinite number of **next states** are possible.
- The point of our **formula** is to restrict which of those **state pairs** are permitted.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

Behaviours

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

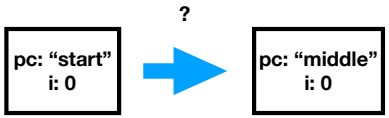
Behaviours

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



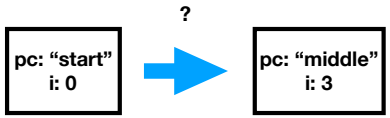
Behaviours

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



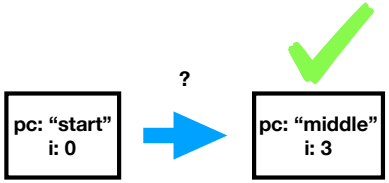
Behaviours

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



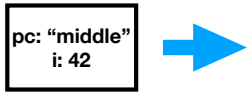
Behaviours

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



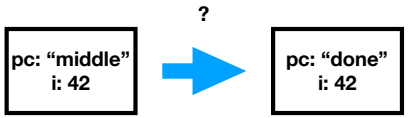
Behaviours

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



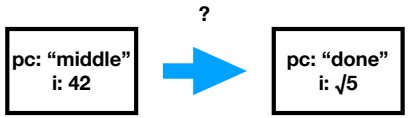
Behaviours

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



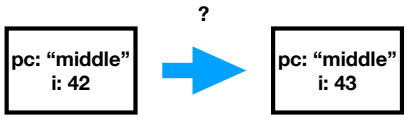
Behaviours

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



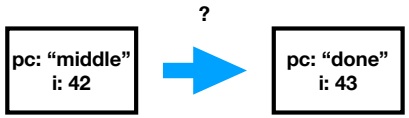
Behaviours

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



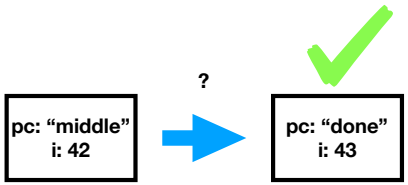
Behaviours

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



Behaviours

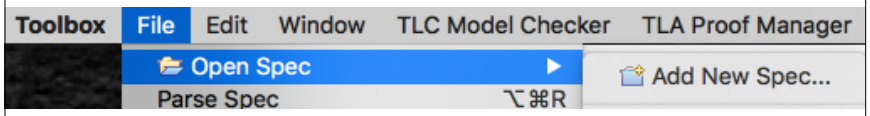
```
\/ /\ pc = "start"
   /\ i' ∈ 1..1000
   /\ pc' = "middle"
\/ /\ pc = "middle"
   /\ i' = i + 1
   /\ pc' = "done
```

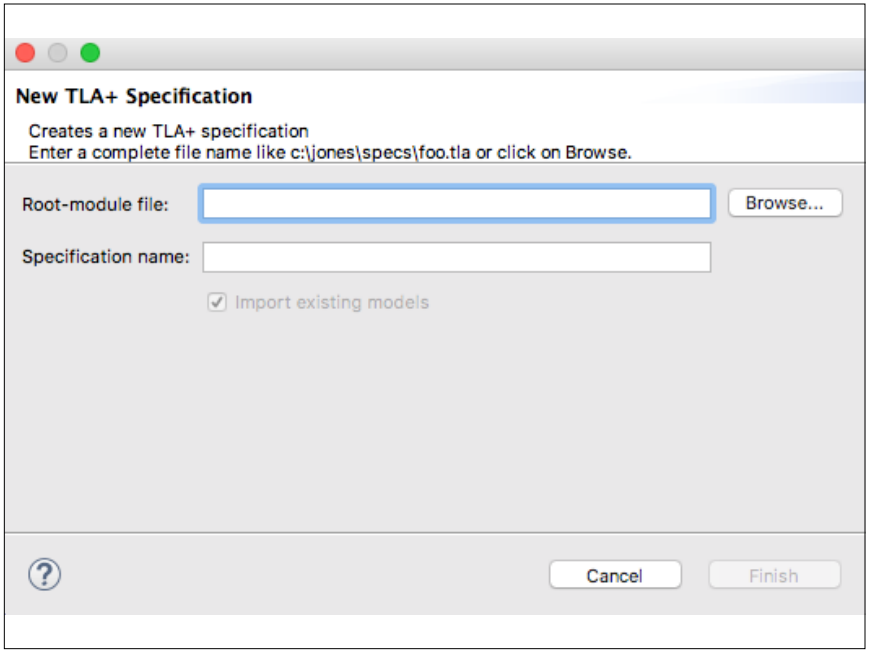


Final Spec for our C Program

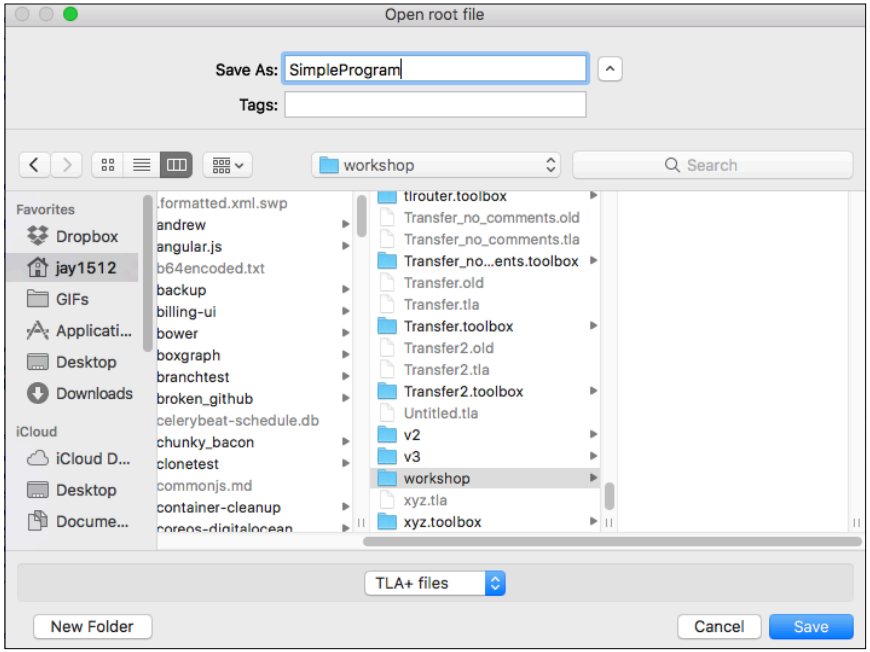
Final Spec for our C Program

```
---- 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
```

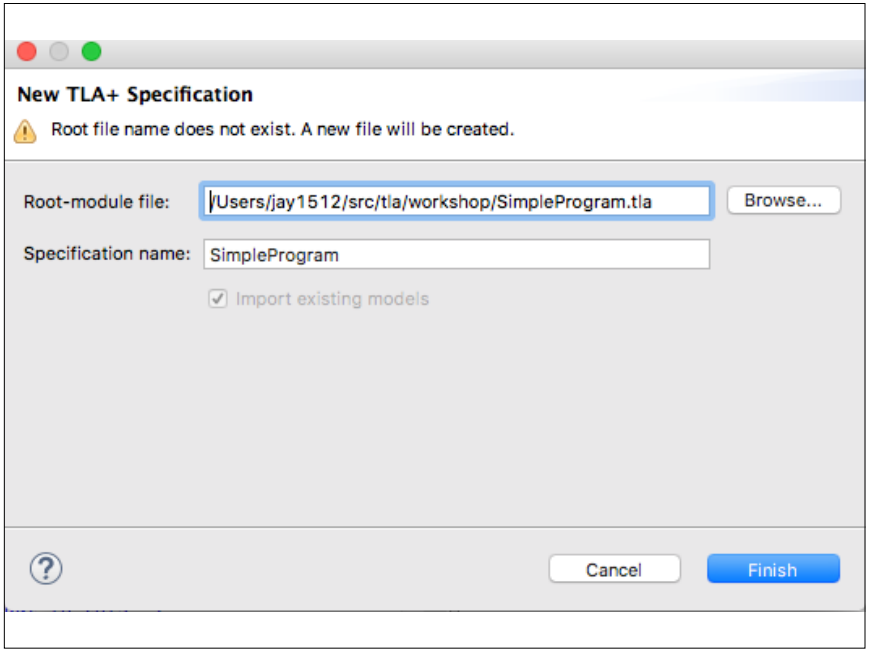





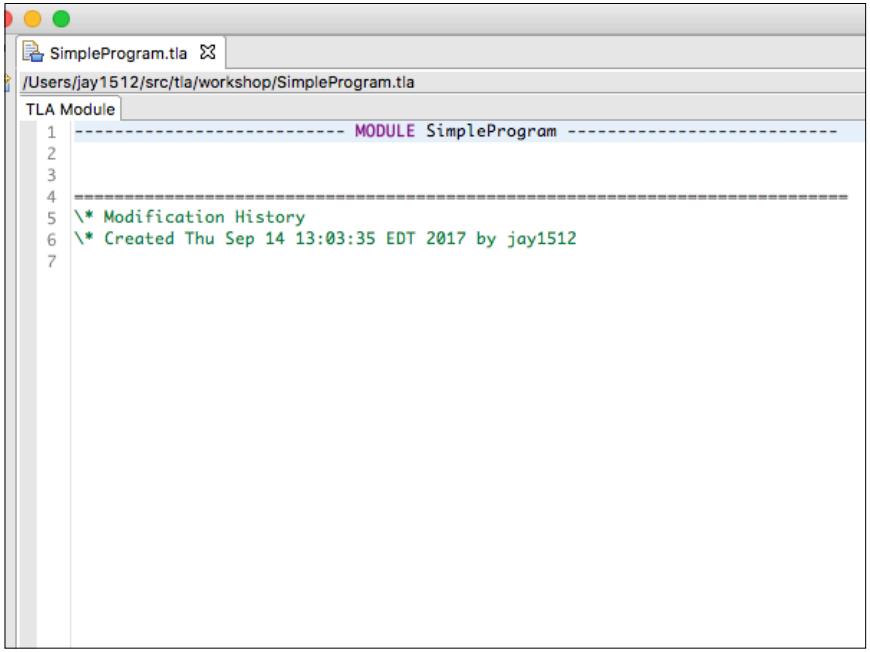
115



116



117



118

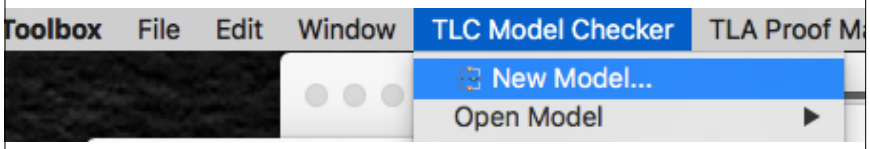
/Users/jay1512/src/tla/workshop/SimpleProgram.tla

TLA Module

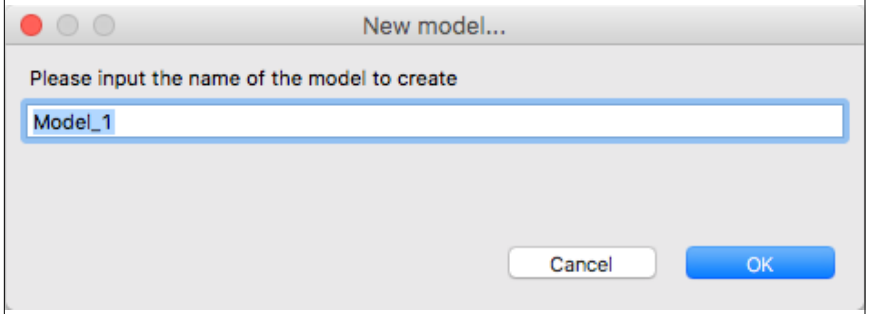
```
1  ---------------------------- MODULE SimpleProgram -------------
2  EXTENDS Integers
3  VARIABLES i, pc
4
5  Init == (pc="start") /\ (i=0)
6
7  Next ==
8        \/ /\ pc = "start"
9           /\ i' \in 1..1000
10          /\ pc' = "middle"
11       \/ /\ pc = "middle"
12          /\ i' = i + 1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

In GitHub at `specs/SimpleProgram.tla`

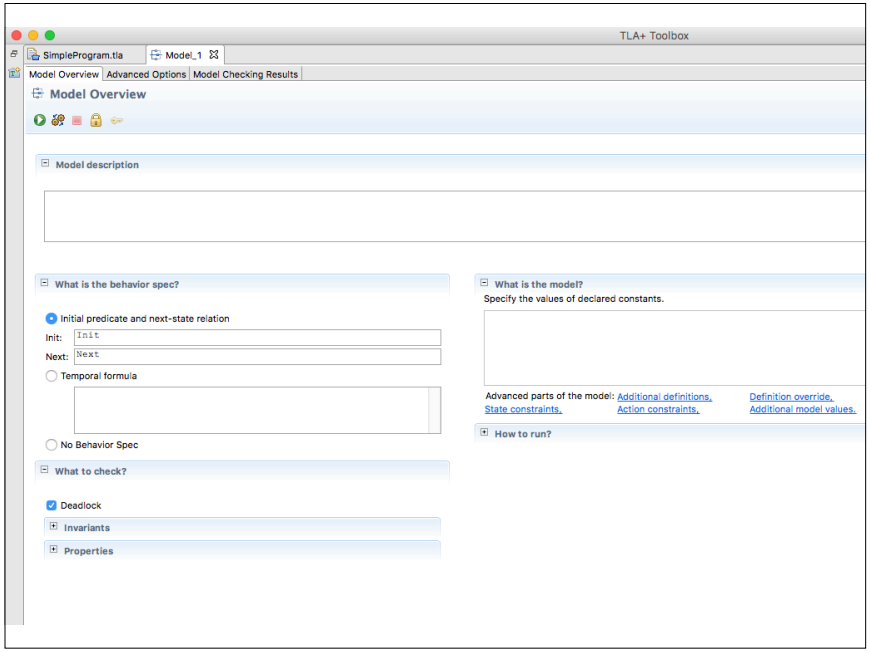
119



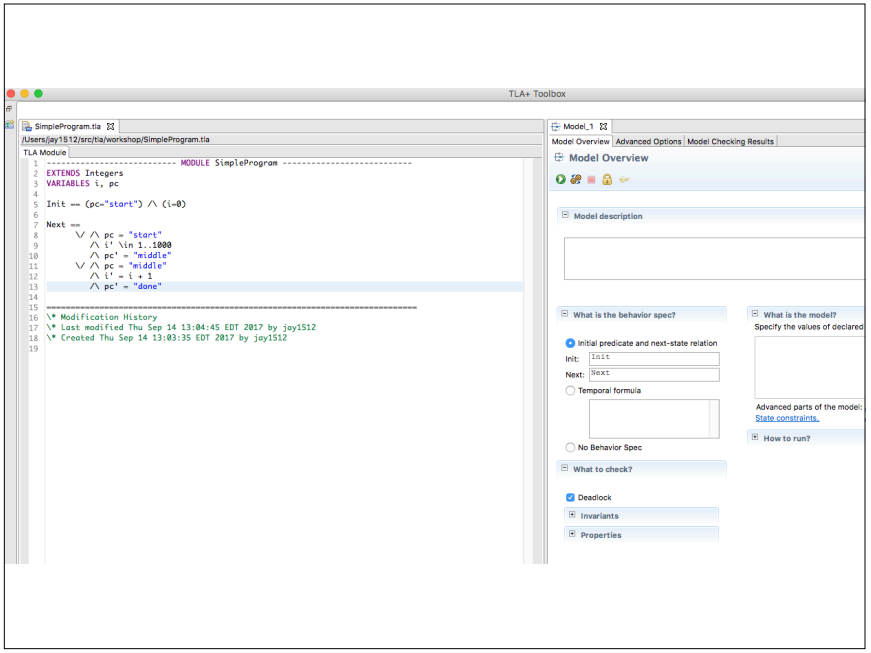
120



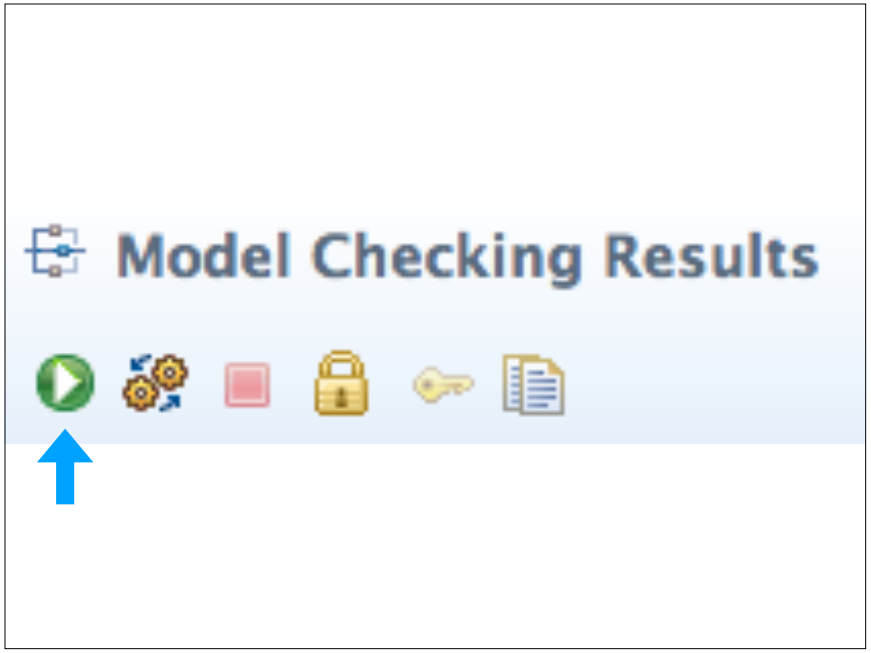
121



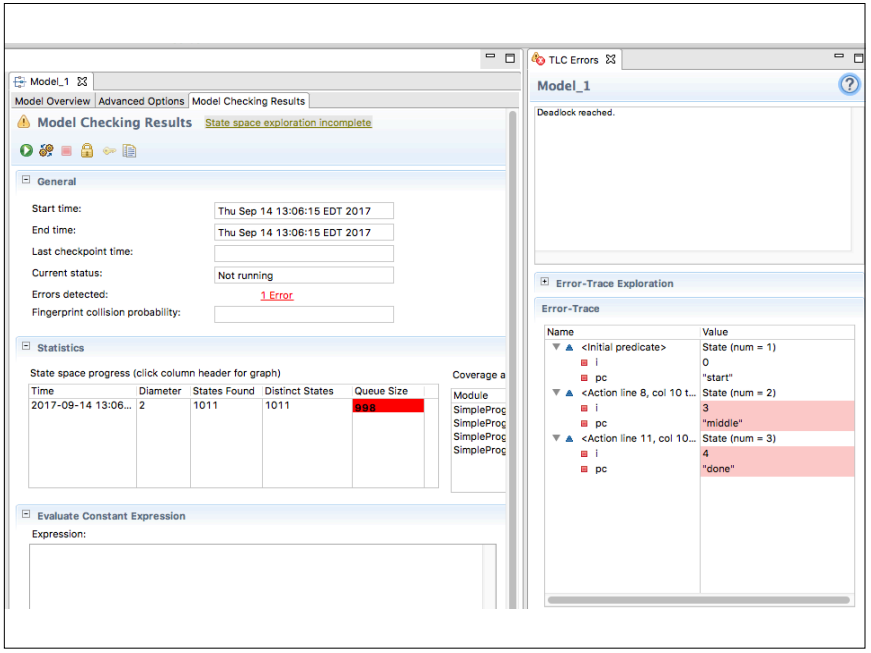
122



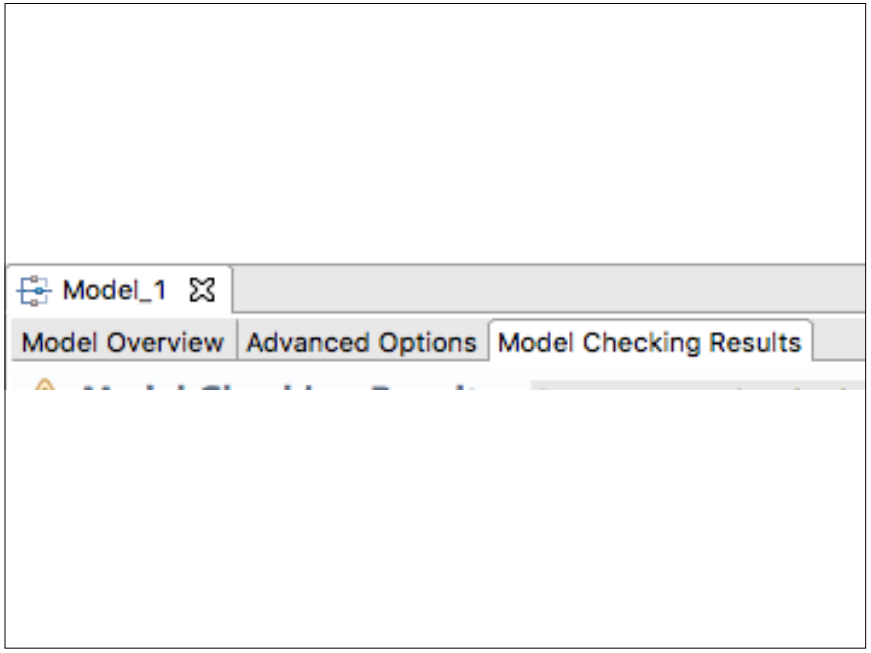
123



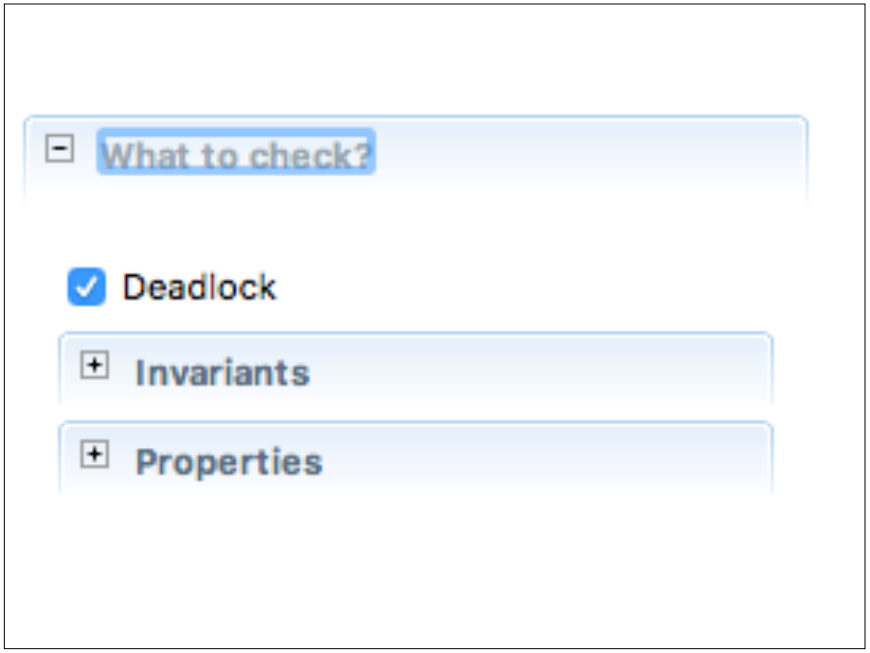
124



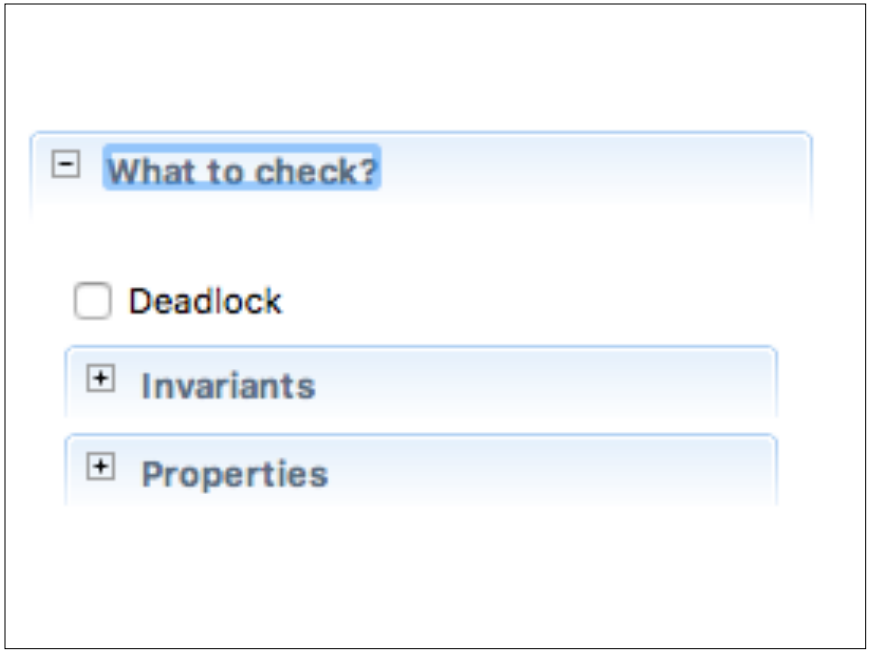
125



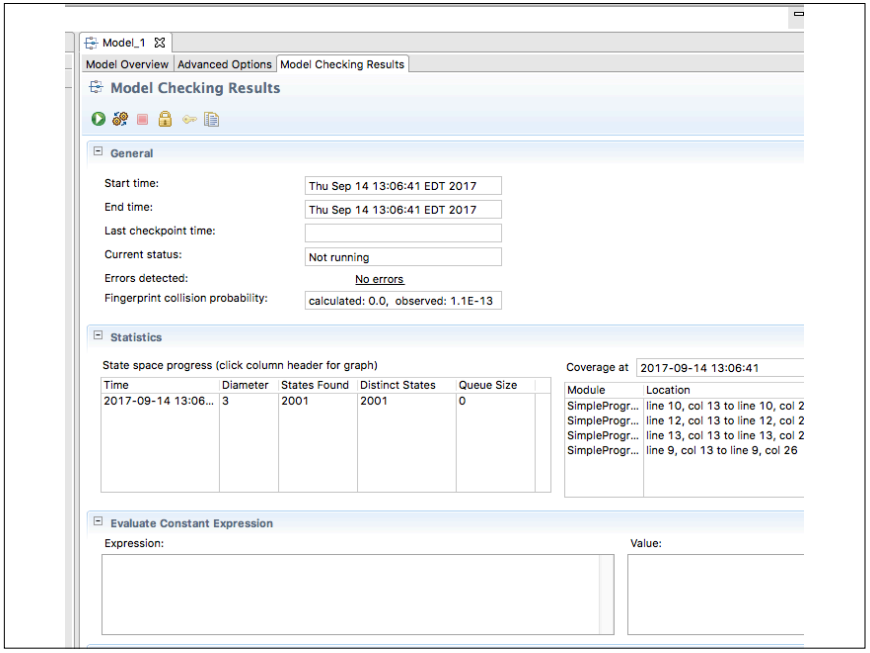
126



127



128



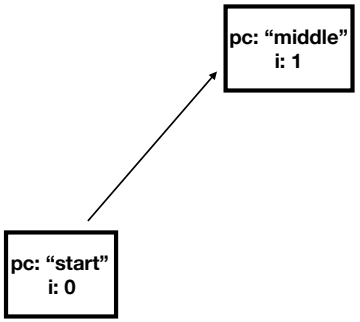
129

All Possible State Transitions

pc: “start”
i: 0

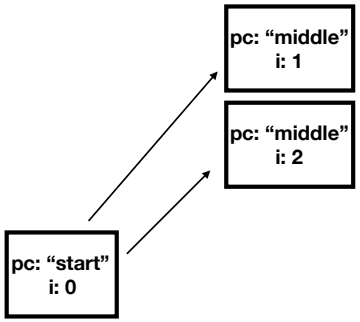
130-1

All Possible State Transitions



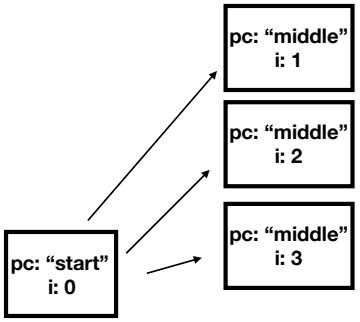
130-2

All Possible State Transitions



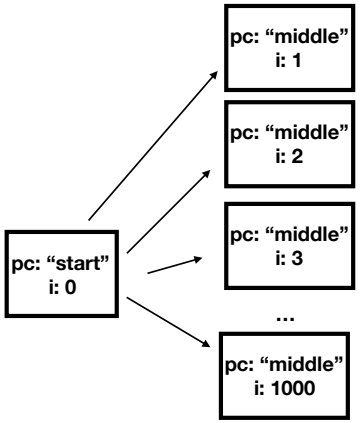
130-3

All Possible State Transitions



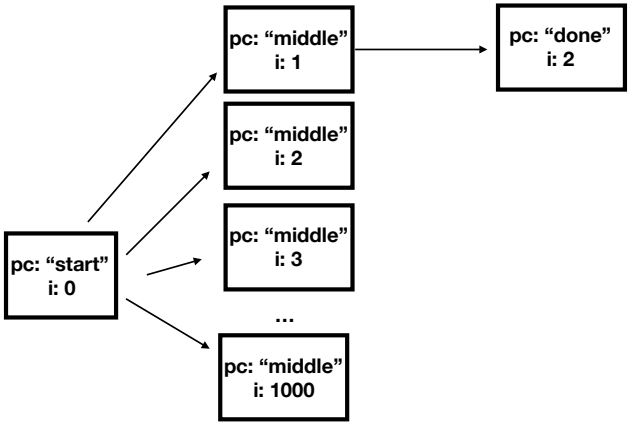
130-4

All Possible State Transitions



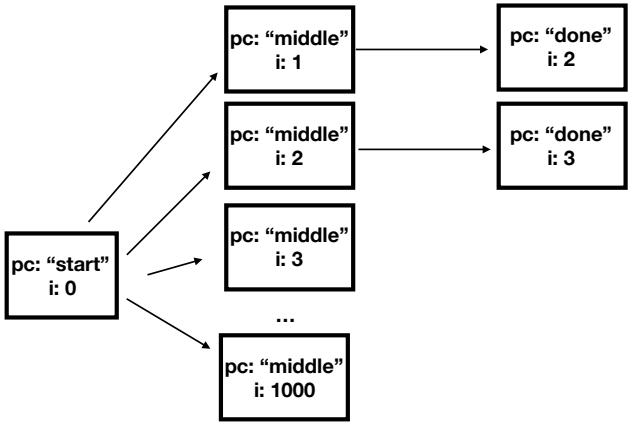
130-5

All Possible State Transitions



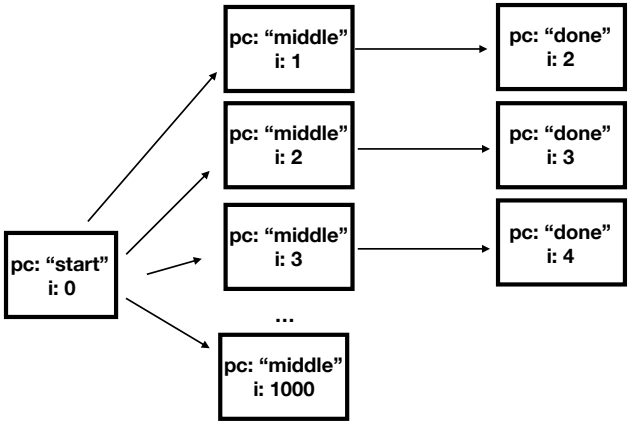
130-6

All Possible State Transitions



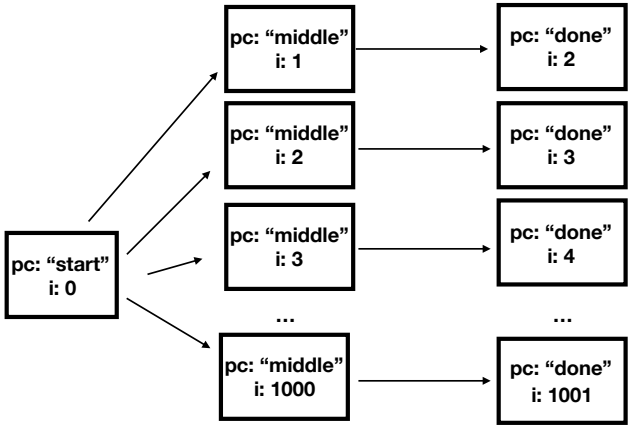
130-7

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
```


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
```


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
```


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"
```

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"
```


Sub-definitions

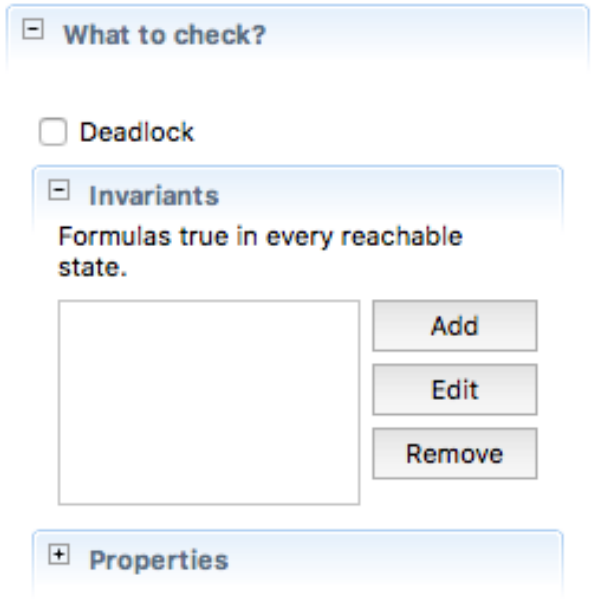
When programming, we don't stuff all our code into a single subroutine. We split it up!
TLA+ allows for the same thing

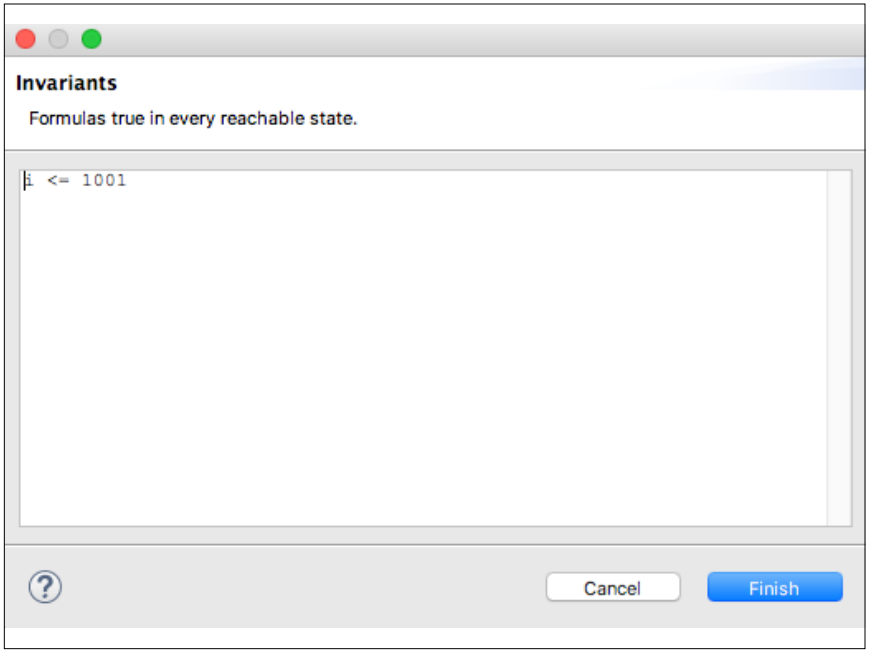
```
Next ==
      \/ /\ pc = "start"
         /\ i' \in 1..1000
         /\ pc' = "middle"
      \/ /\ pc = "middle"
         /\ i' = i + 1
         /\ pc' = "done"

Pick ==
      /\ pc = "start"
      /\ i' \in 1..1000
      /\ pc' = "middle"

Add ==
      /\ pc = "middle"
      /\ i' = i + 1
      /\ pc' = "done"

Next == Pick \/ Add
```



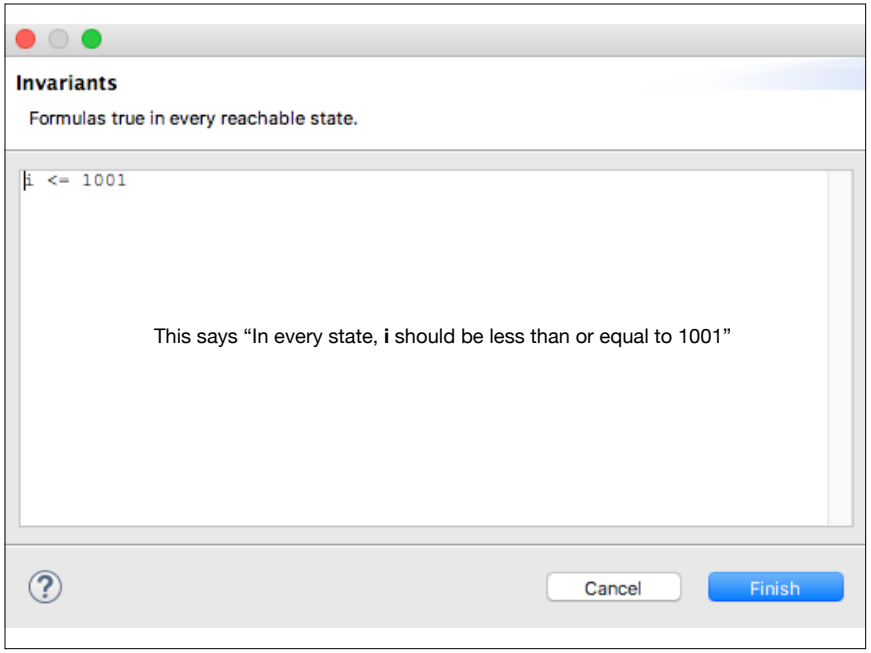
```
EXTENDS Integers
VARIABLES i, pc

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





134-1



134-2

Run the model again, everything should still be fine

135

Change the Add action

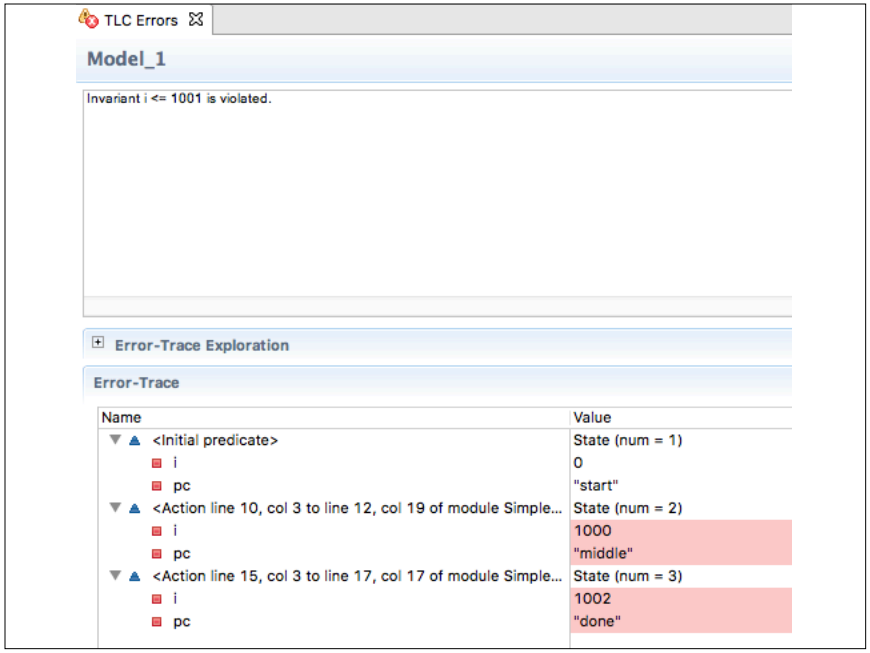
```
Add ==
  /\ pc = "middle"
  /\ i' = i + 2
  /\ pc' = "done"
```

136-1

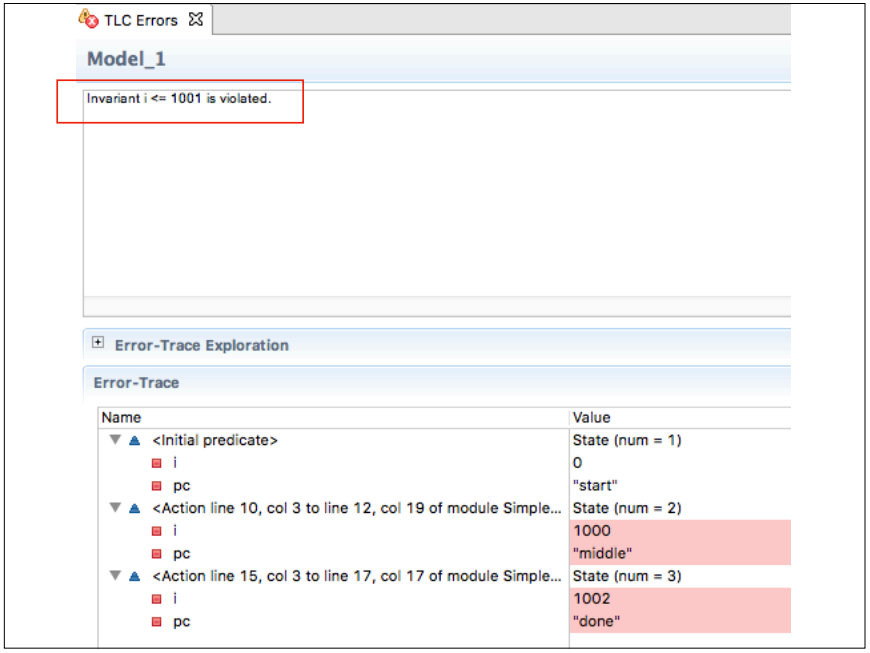
Change the Add action

```
Add ==
   /\ pc = "middle"
   /\ i' = i + 2
   /\ pc' = "done"
```

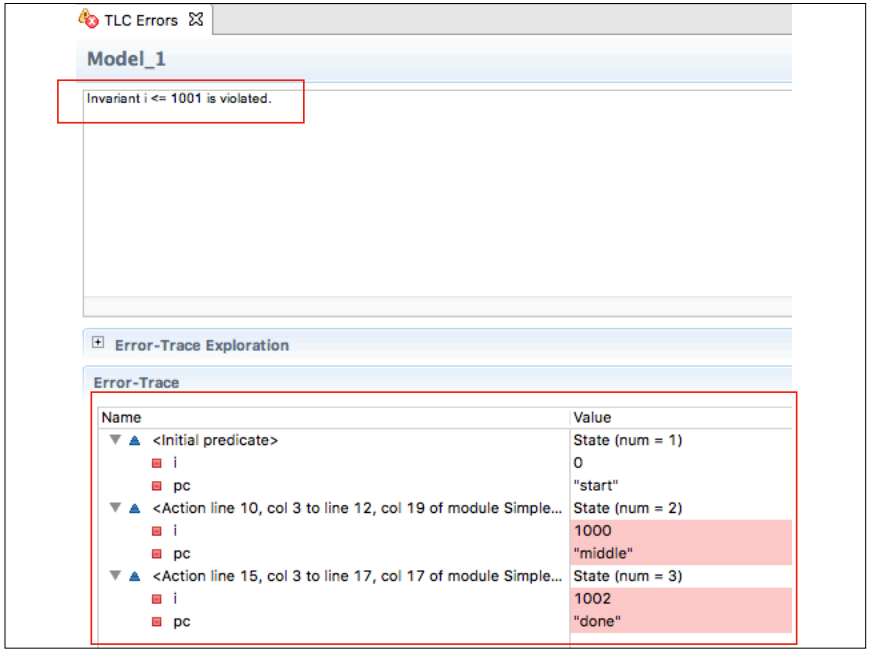
136-2



137-1

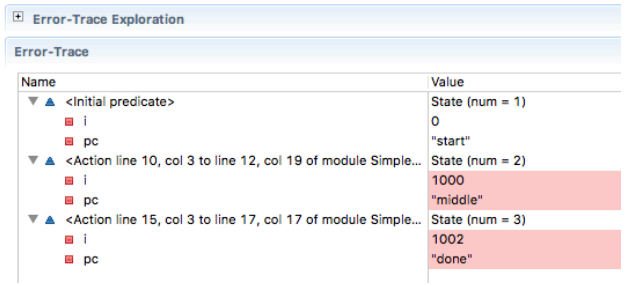


137-2



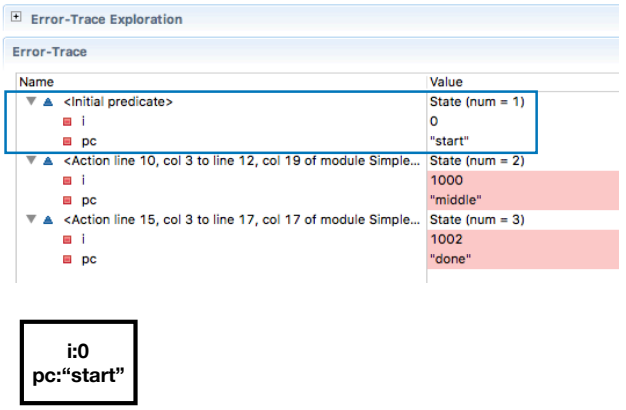
137-3

Error Trace



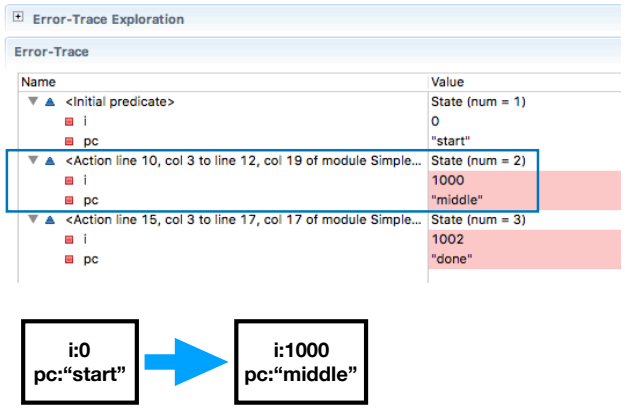
138-1

Error Trace



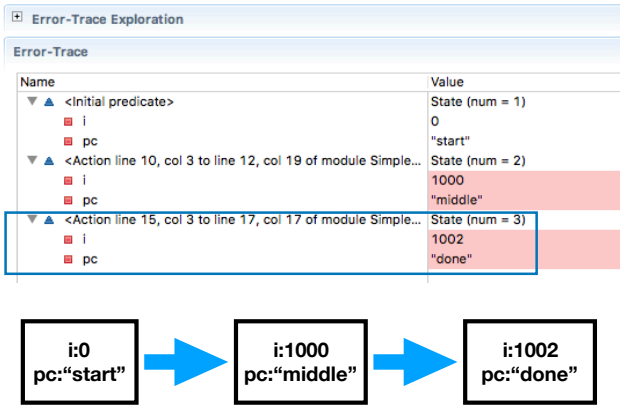
138-2

Error Trace



138-3

Error Trace



138-4

Die Hard Variables

```
VARIABLES big, small
```


Die Hard Initial States

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


Die Hard Next States

Die Hard Next States

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

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**

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**

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**

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**

Die Hard Next States

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


FillSmall

FillSmall

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


FillSmall

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


EmptySmall

EmptySmall

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

EmptySmall

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


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**

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

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
```

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
```

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

SmallToBig

Putting it all together

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


SmallToBig

Putting it all together

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


SmallToBig

Putting it all together

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

SmallToBig

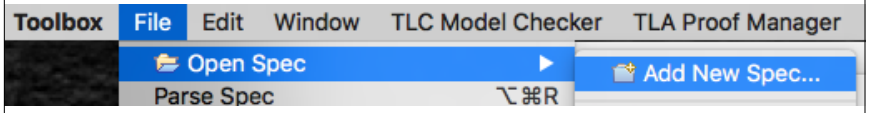
Putting it all together

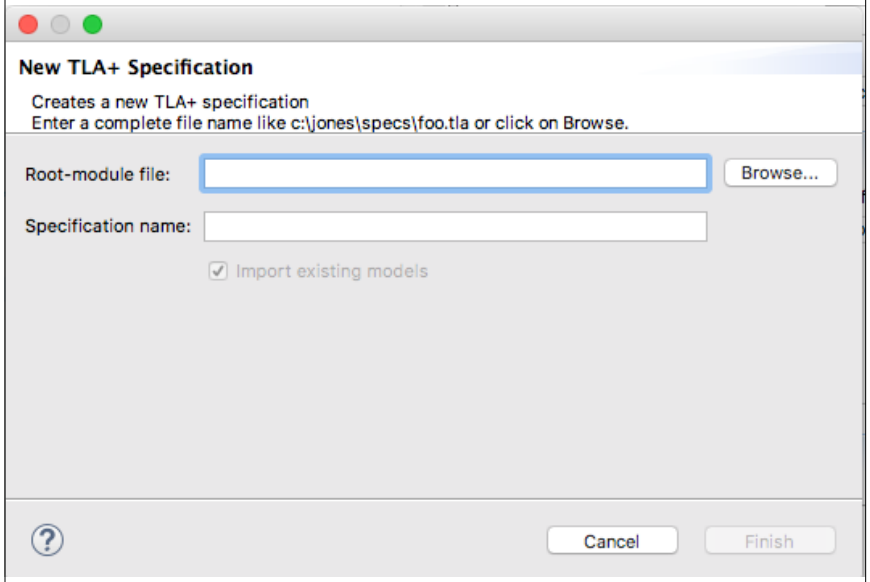
```
SmallToBig == IF big + small < 5
              THEN /\ big' = big + small
                   /\ small' = 0
              ELSE /\ small' = small - (5 - big)
```

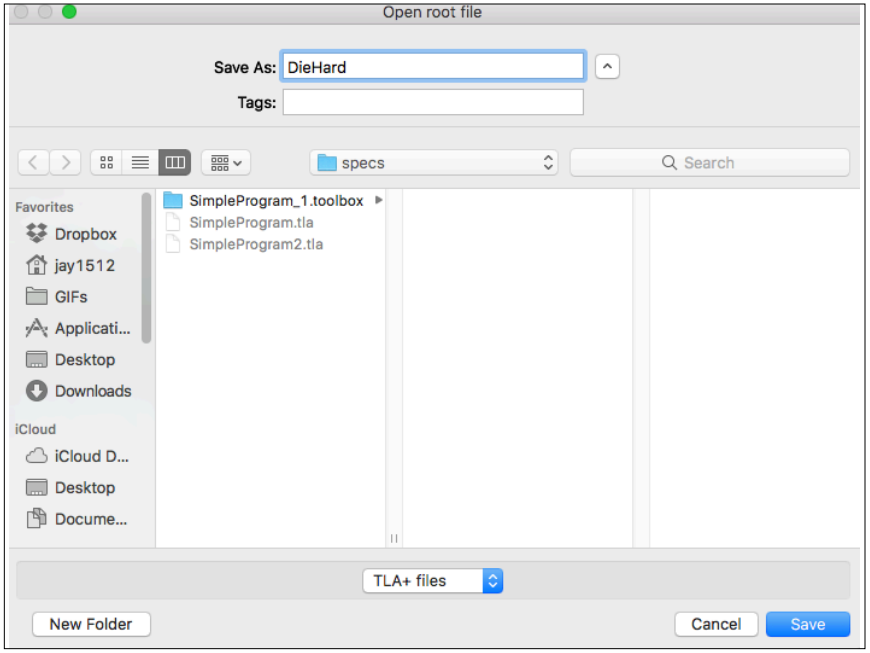

SmallToBig

Putting it all together

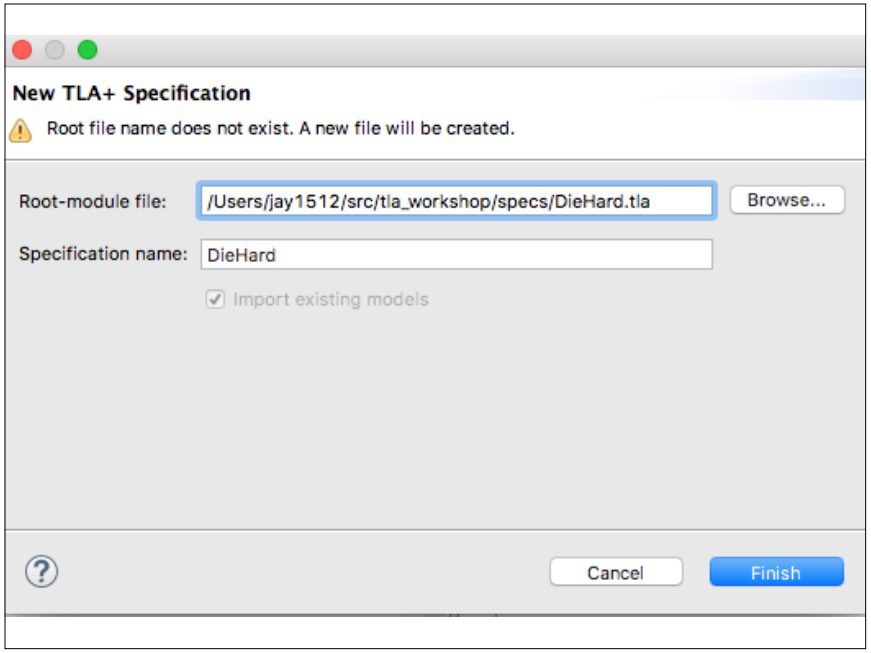
```
SmallToBig == IF big + small < 5
              THEN /\ big' = big + small
                   /\ small' = 0
              ELSE /\ small' = small - (5 - big)
                   /\ big' = 5
```

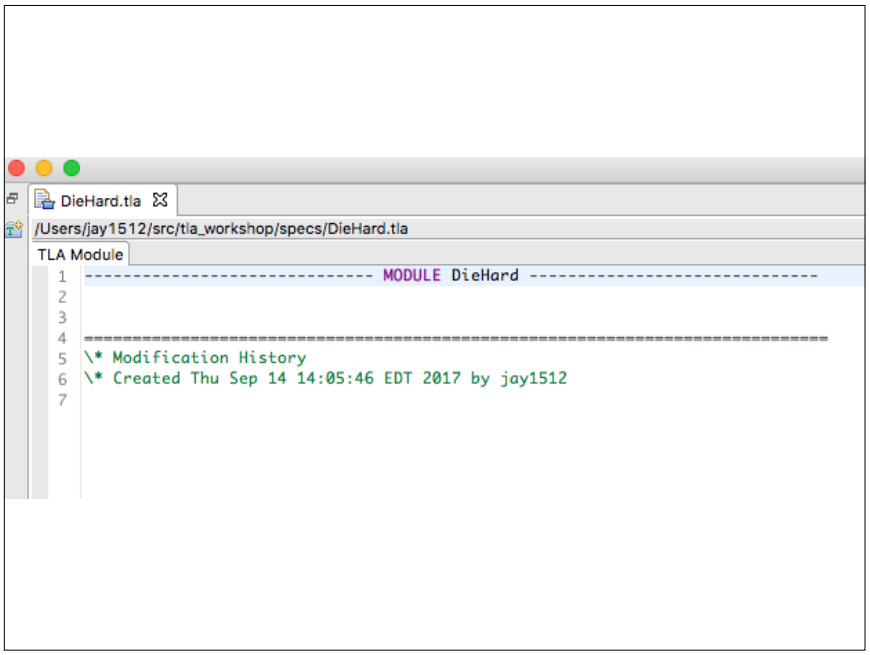




152



153

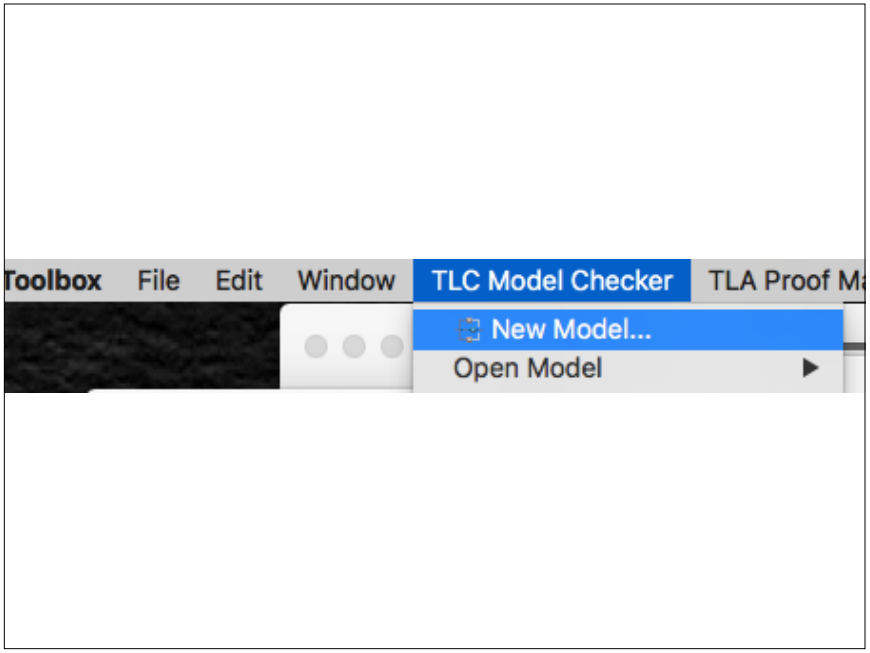


154

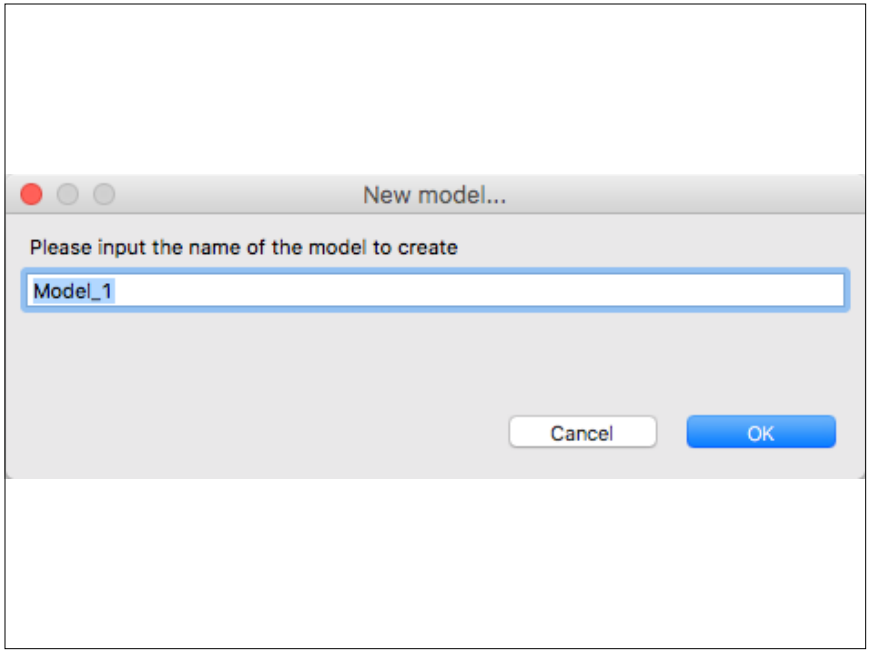
```
---------- MODULE DieHard ----------
EXTENDS Integers

VARIABLES small, big

TypeOK == /\ small \in 0..3
          /\ big   \in 0..5

Init == /\ big   = 0
        /\ small = 0

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

FillBig ==

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

EmptyBig ==

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

BigToSmall ==

Next == \/ FillSmall
        \/ FillBig
        \/ EmptySmall
        \/ EmptyBig
        \/ SmallToBig
        \/ BigToSmall
```

specs/DieHard_incomplete.tla

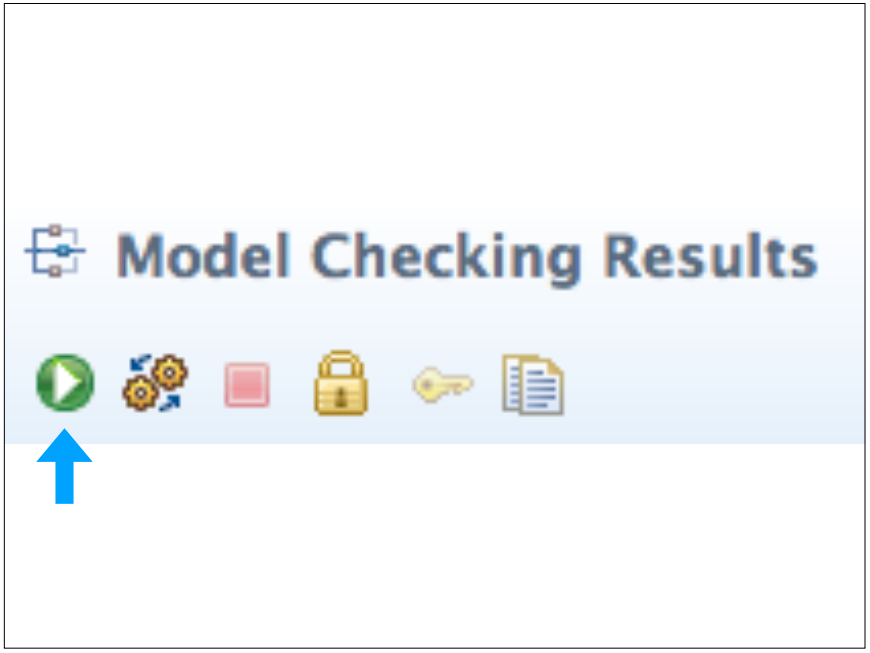
155



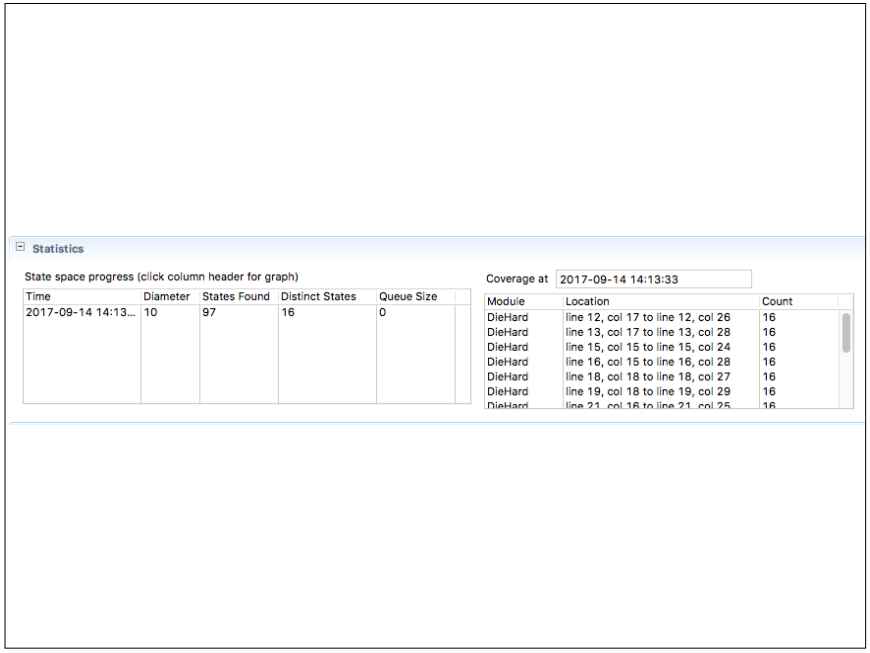
156



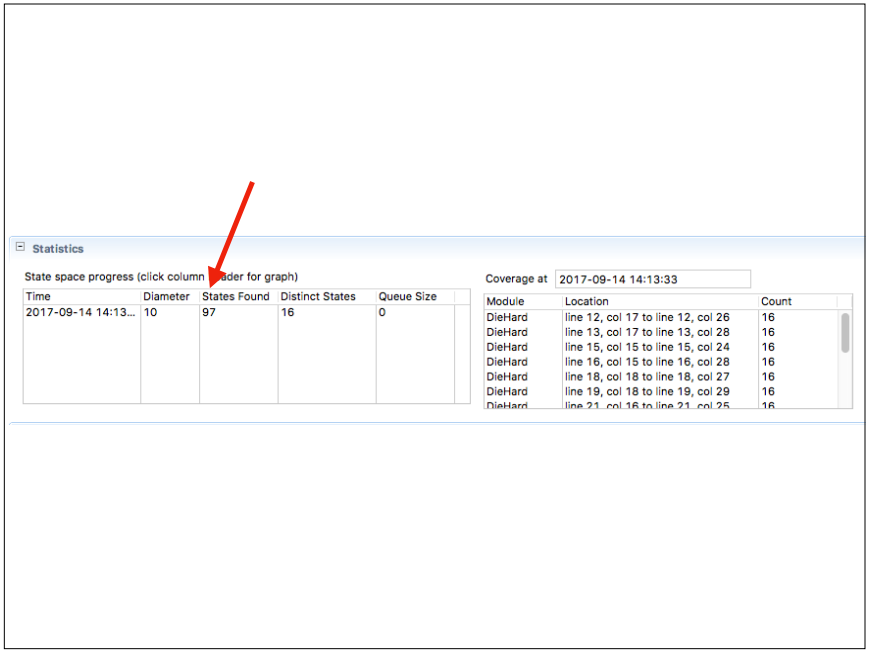
157



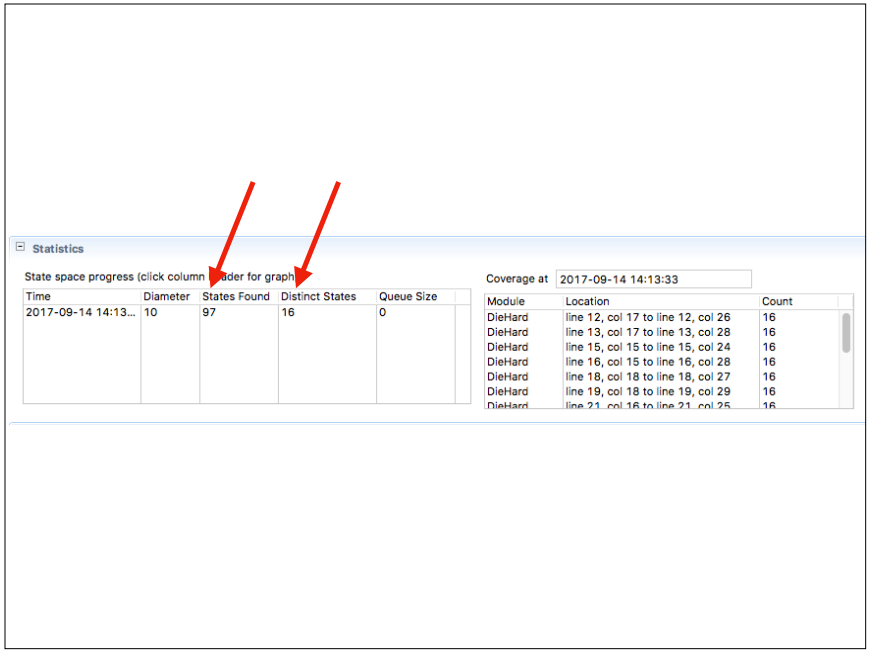
158



159-1



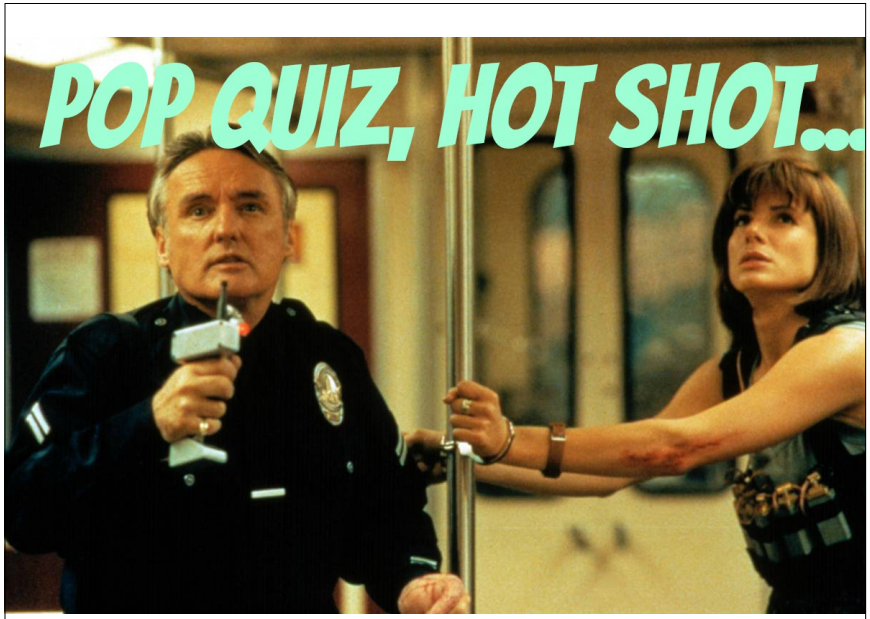
159-2



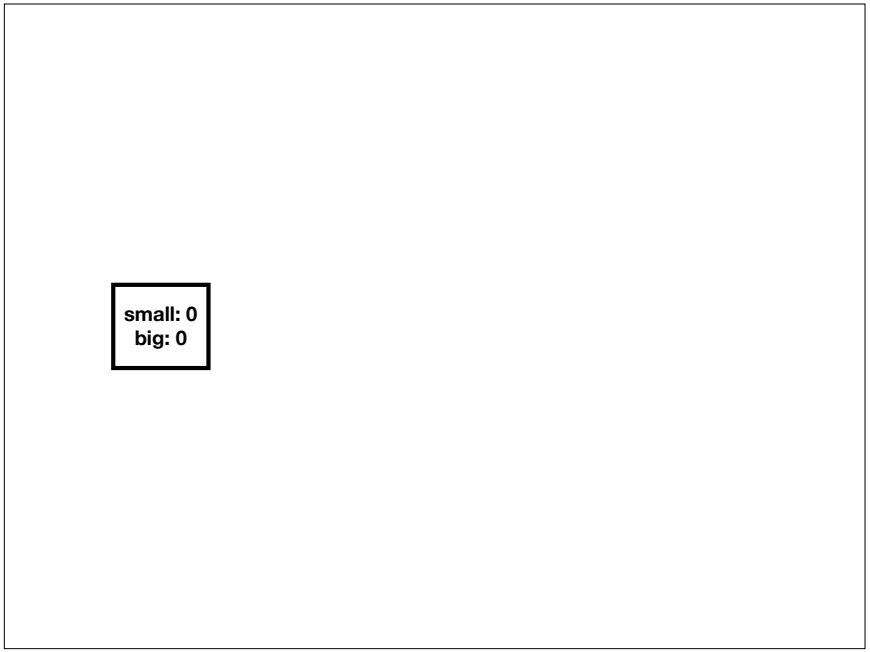
159-3

```
EXTENDS Integers
VARIABLES small, big

TypeOK == /\ small \in 0..3
          /\ big   \in 0..5

Init == /\ big   = 0
        /\ small = 0

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

FillBig == /\ big'   = 5
           /\ small' = small

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

EmptyBig == /\ big'   = 0
            /\ small' = small

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

BigToSmall == IF big + small =< 3
              THEN /\ big'   = 0
                   /\ small' = big + small
              ELSE /\ big'   = small - (3 - big)
                   /\ small' = 3

Next == \/ FillSmall
        \/ FillBig
        \/ EmptySmall
        \/ EmptyBig
        \/ SmallToBig
        \/ BigToSmall
```

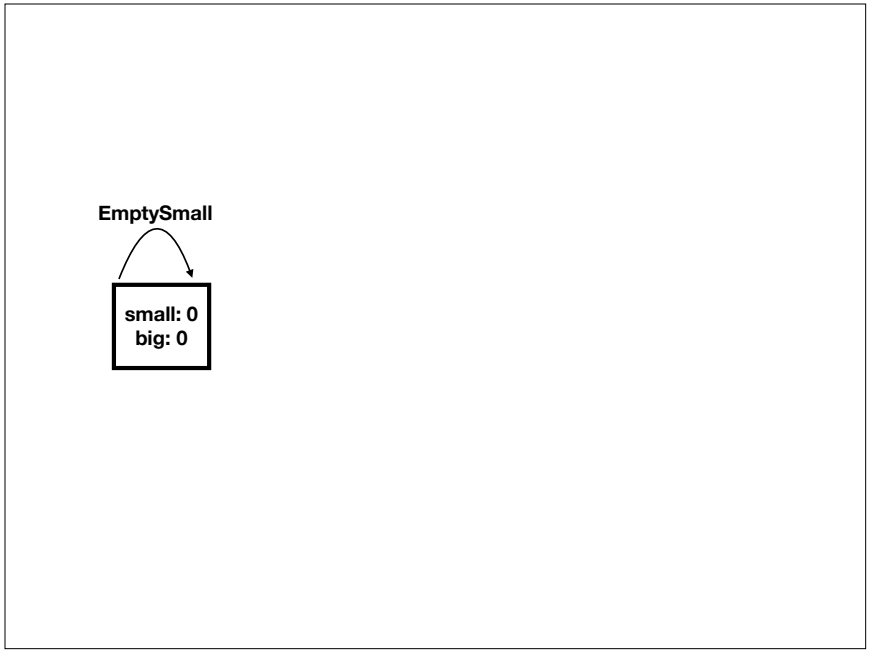
160



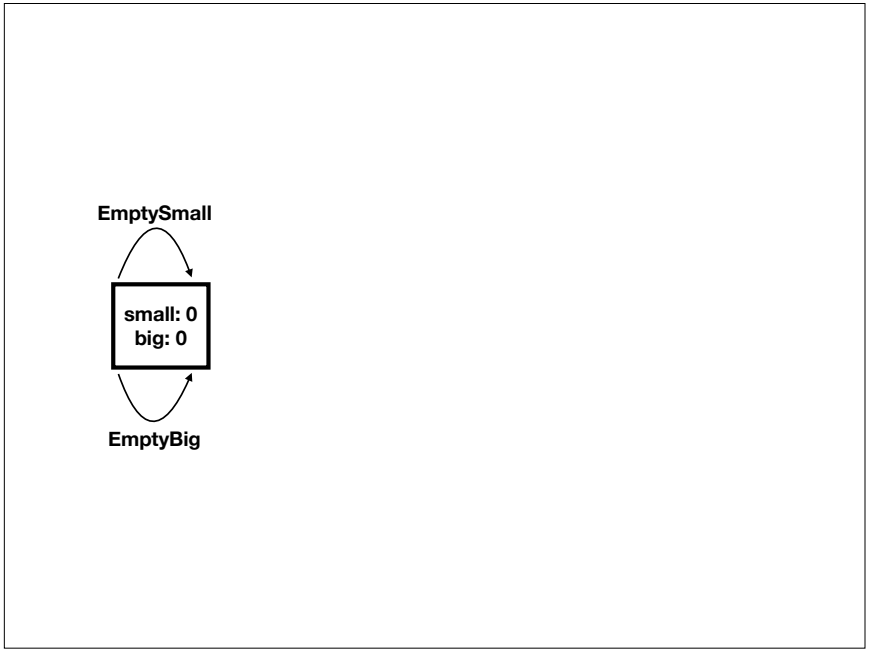
161



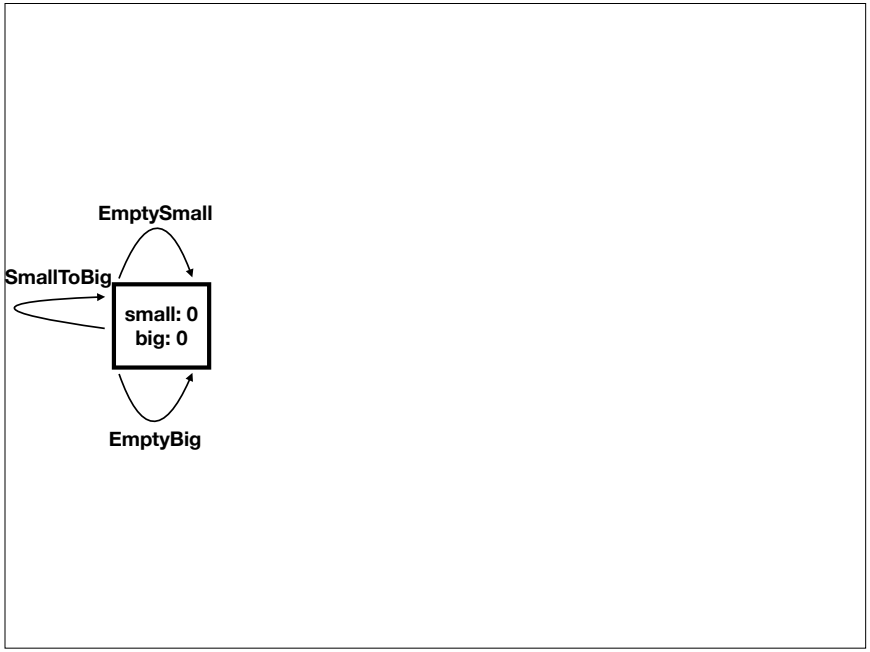
162-1



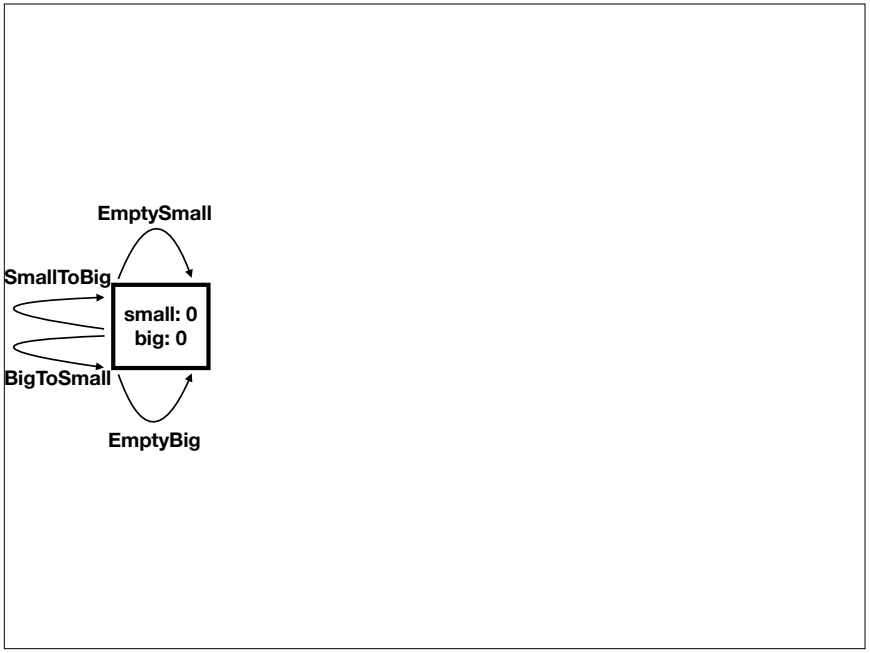
162-2



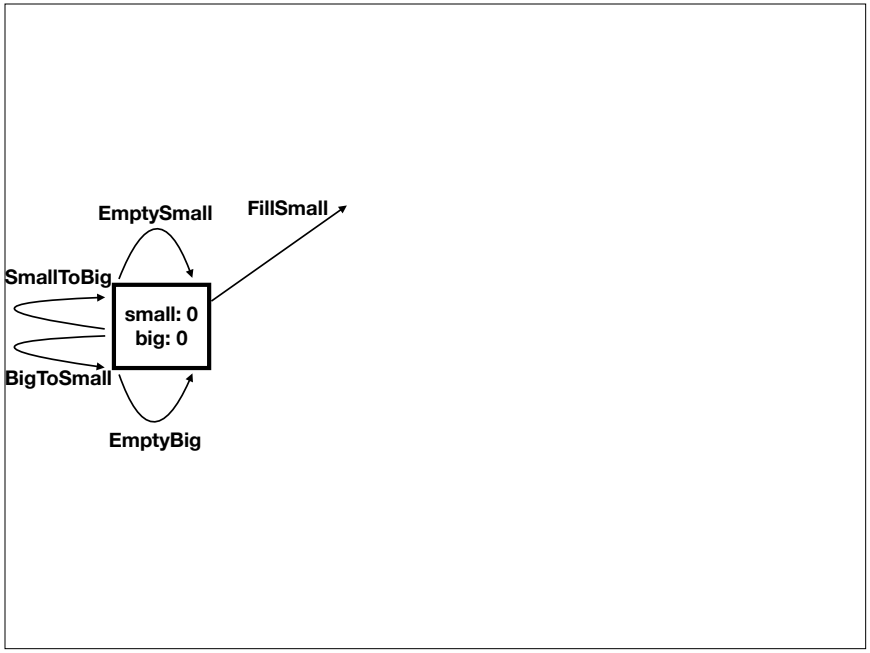
162-3



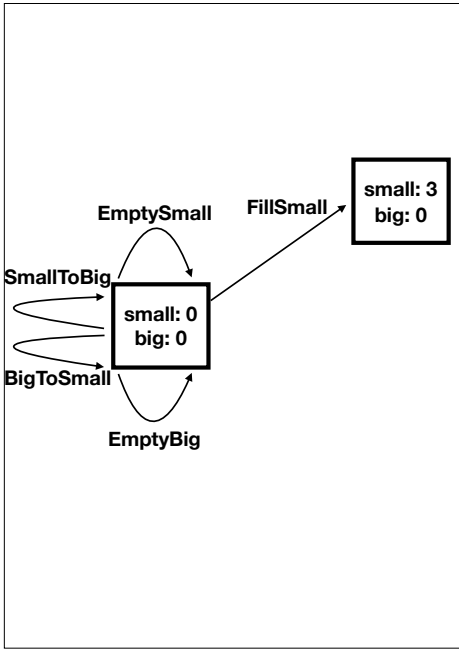
162-4



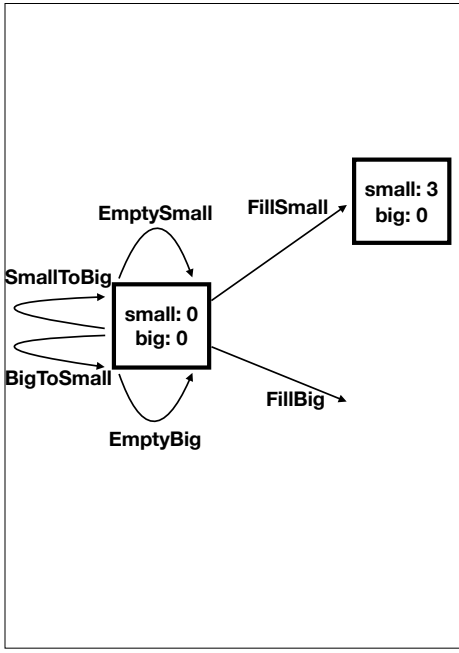
162-5



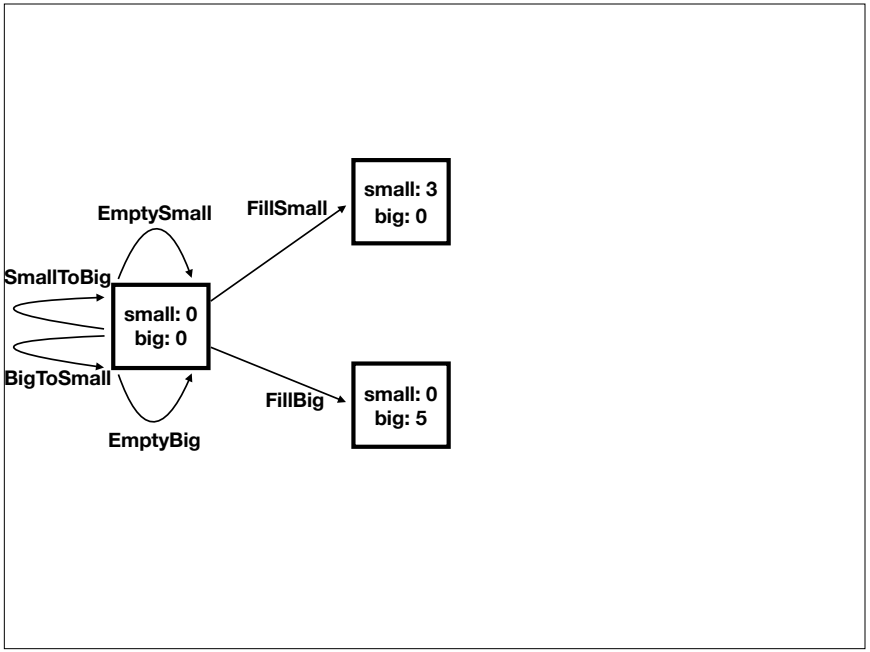
162-6



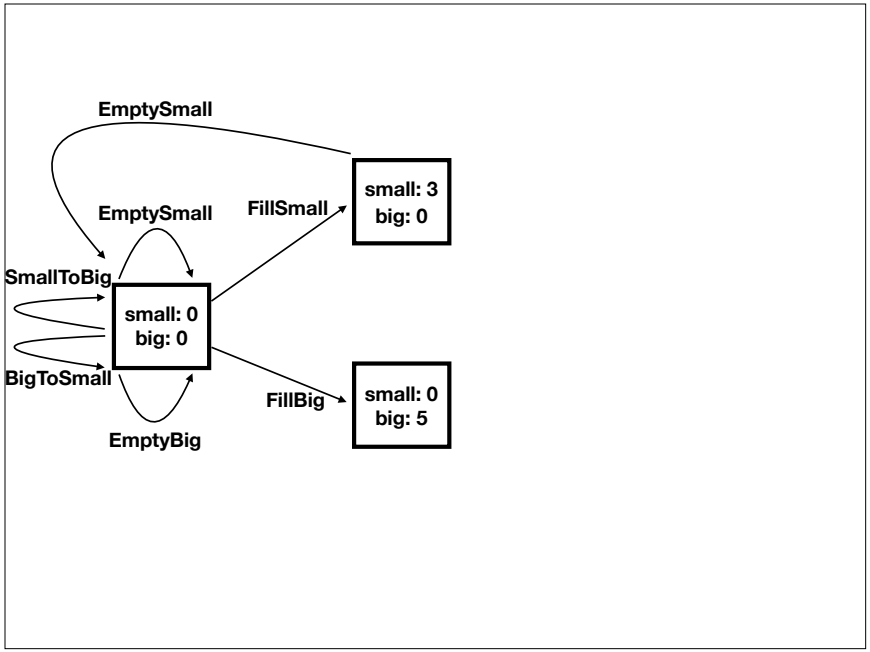
162-7



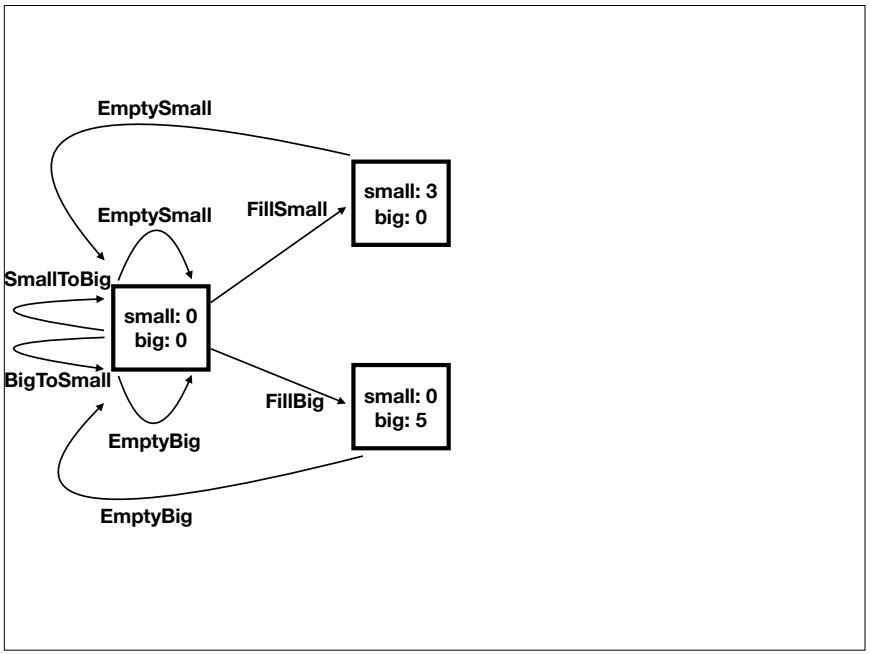
162-8



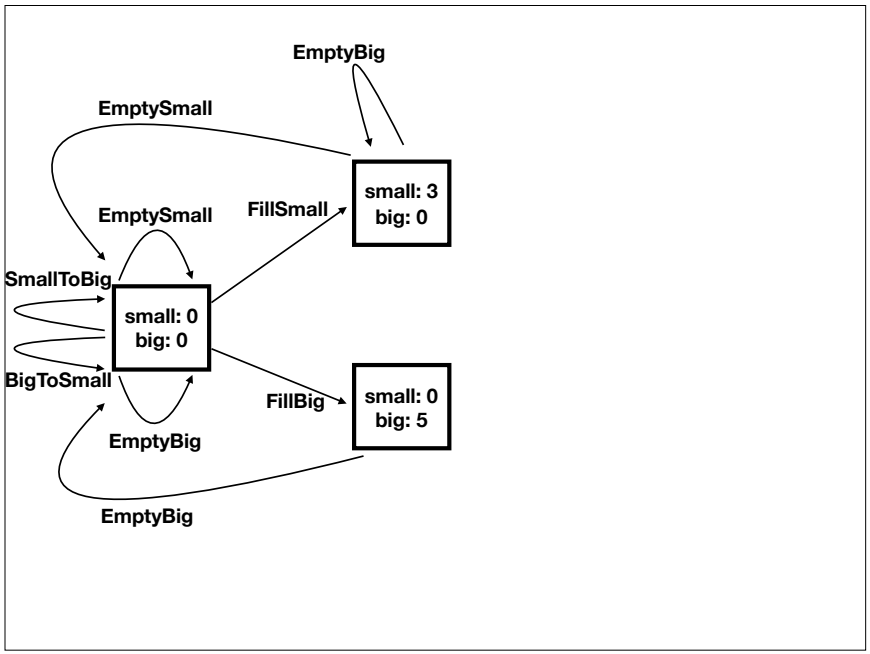
162-9



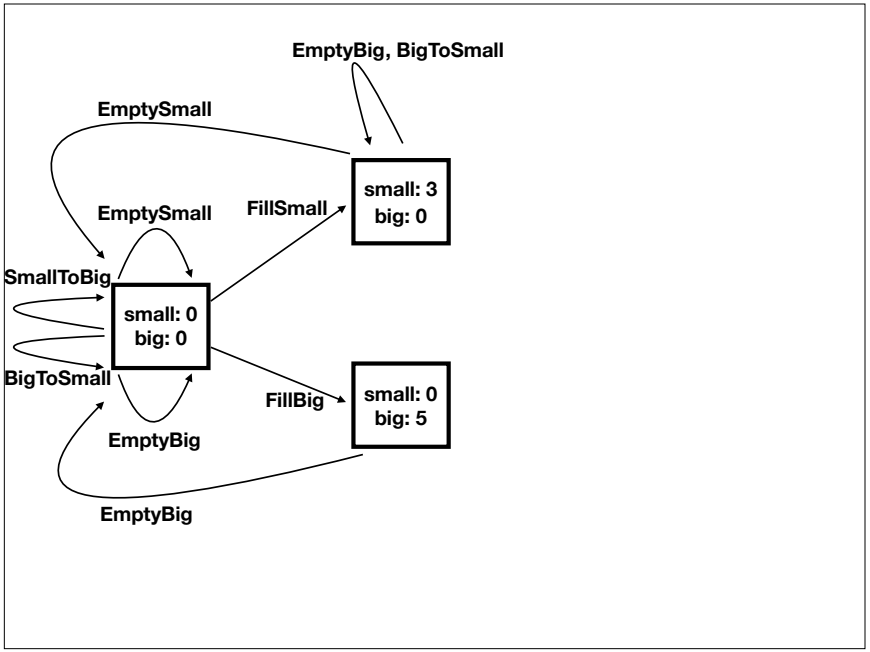
162-10



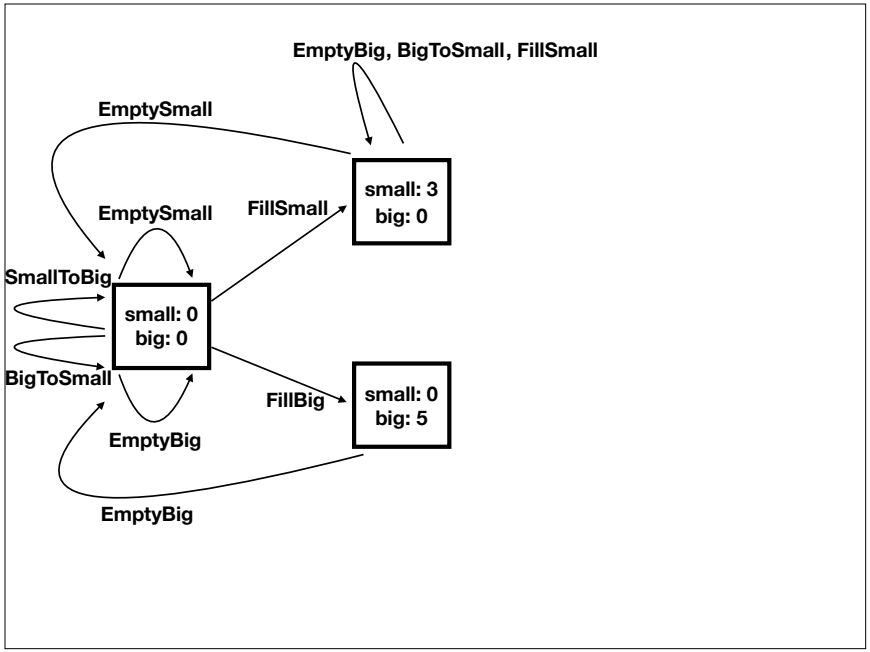
162-11



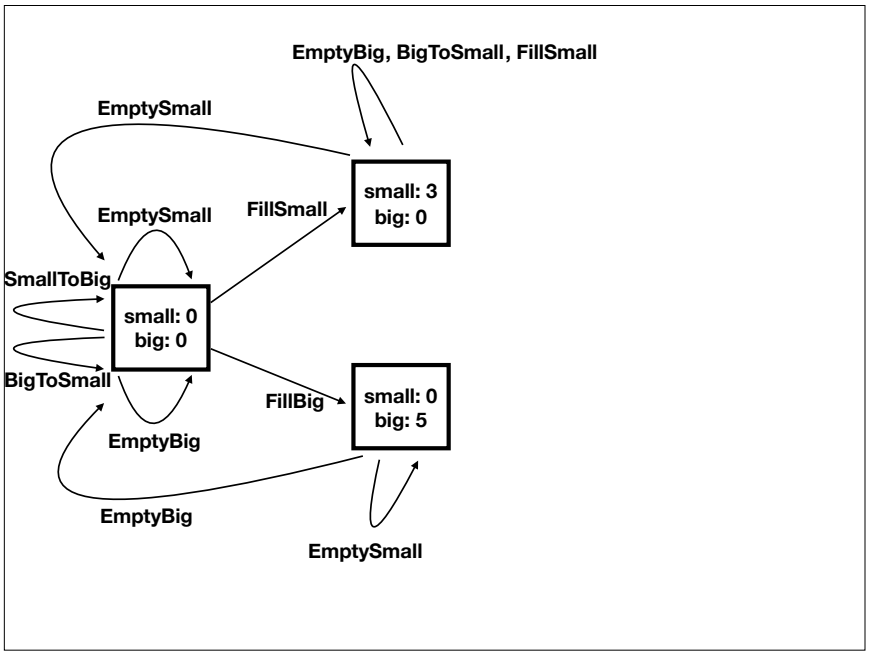
162-12



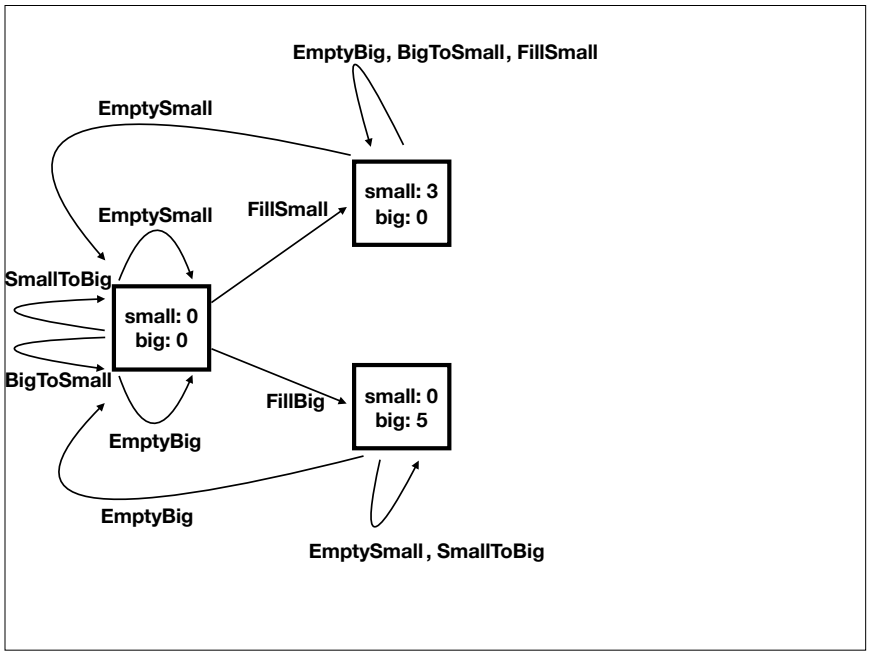
162-13



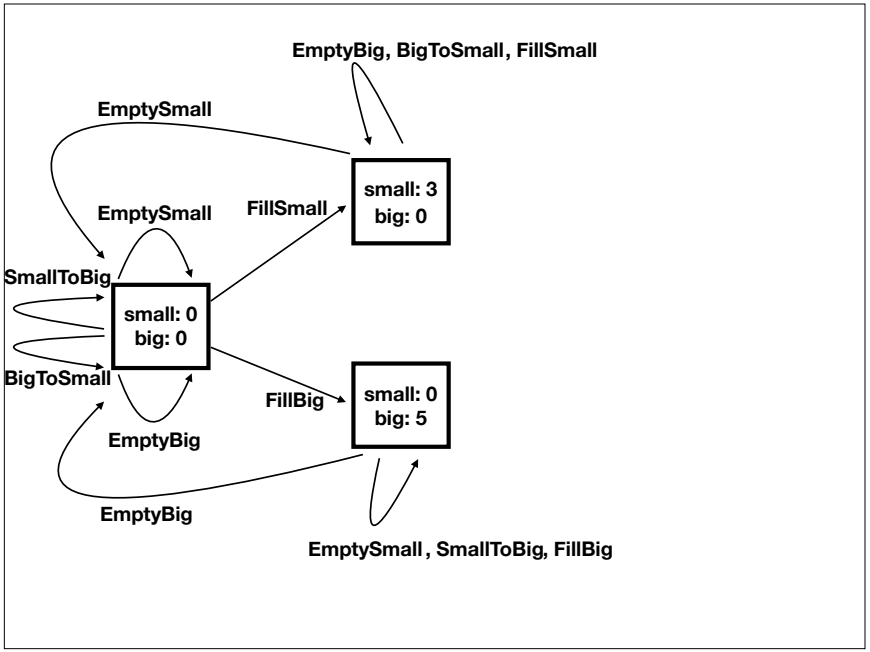
162-14



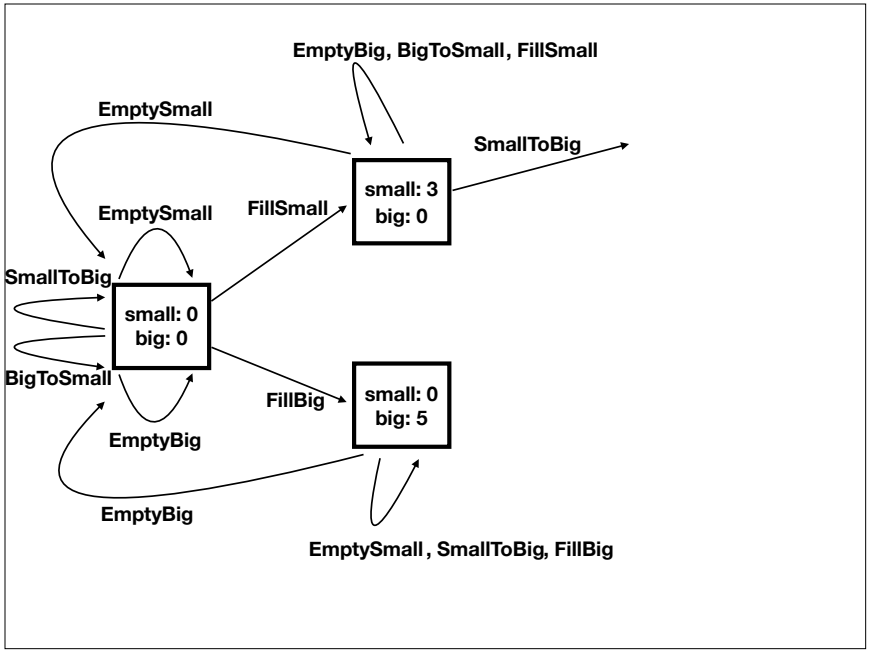
162-15



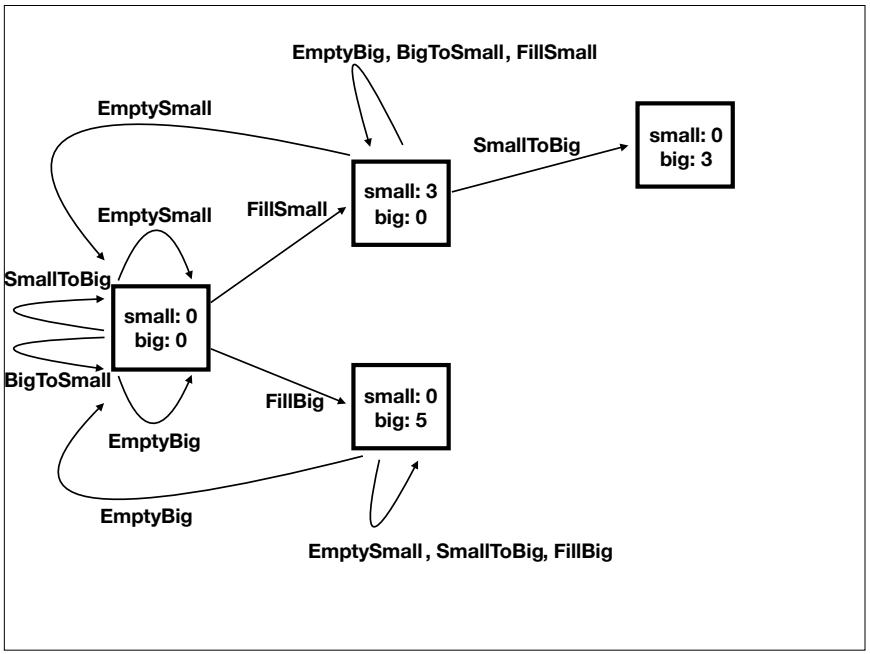
162-16



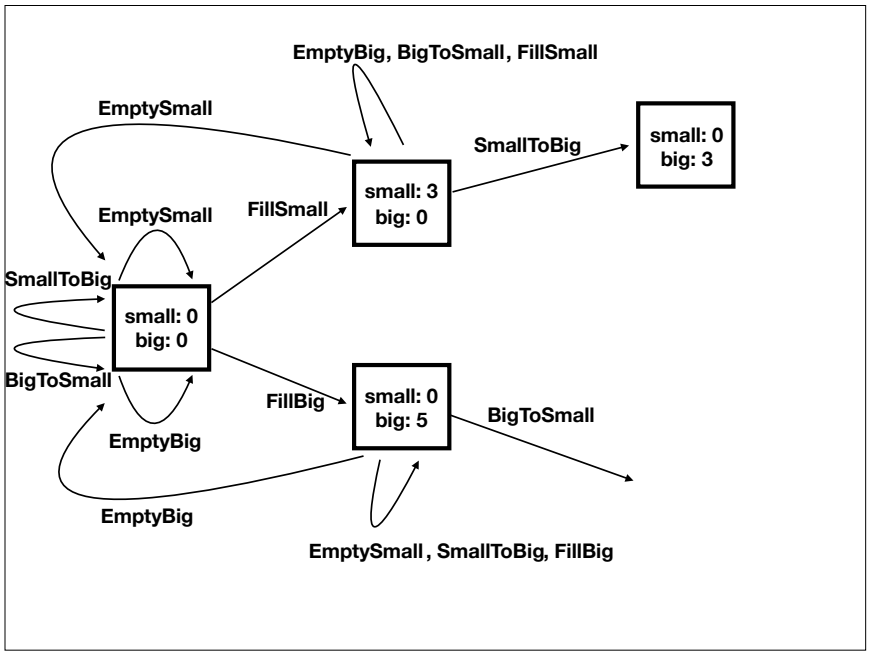
162-17



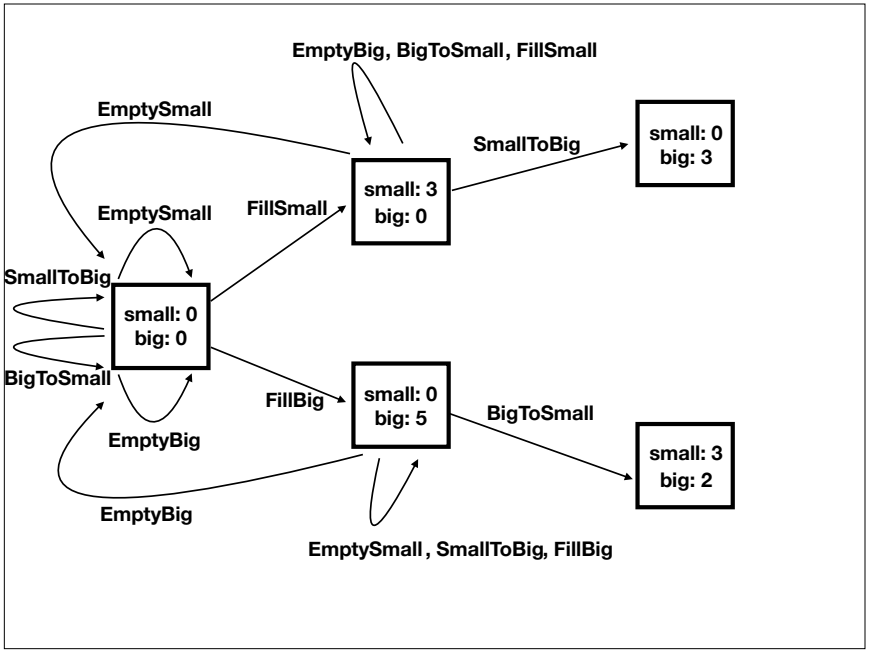
162-18



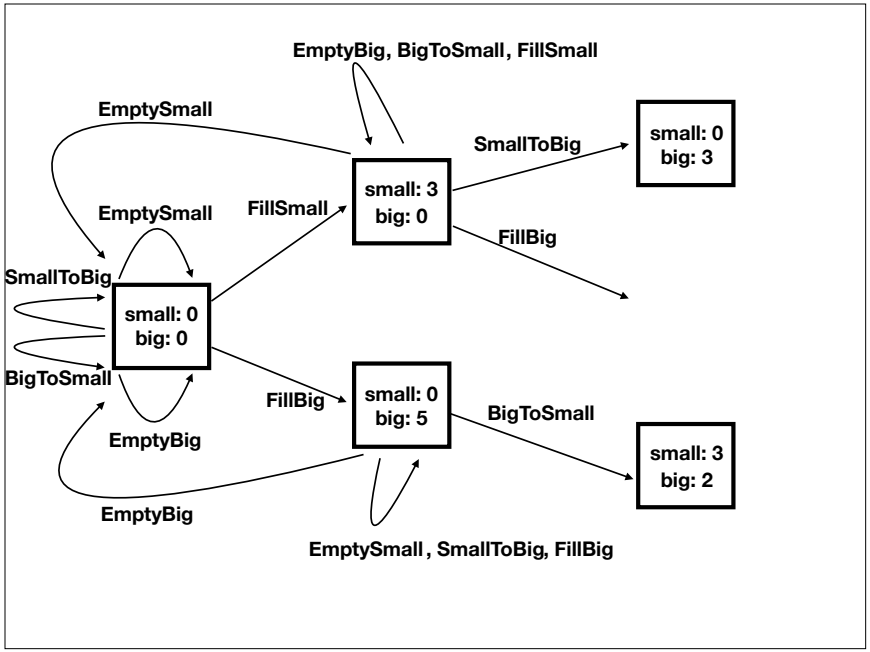
162-19



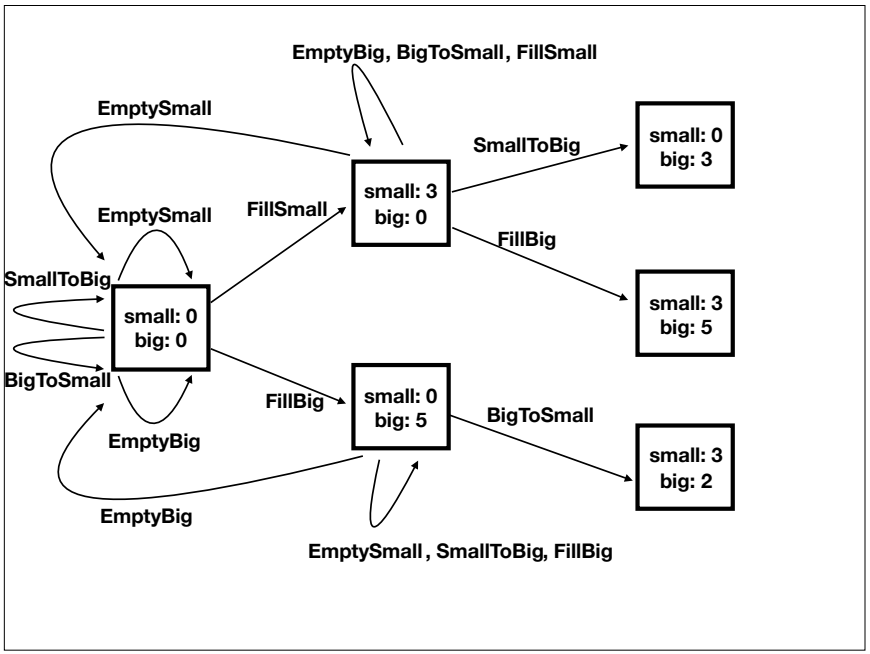
162-20



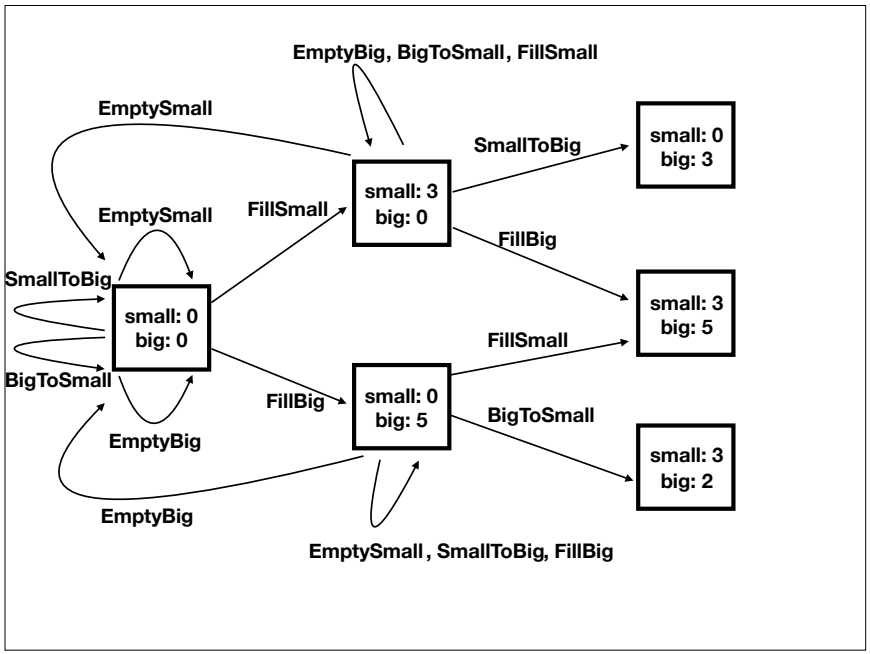
162-21



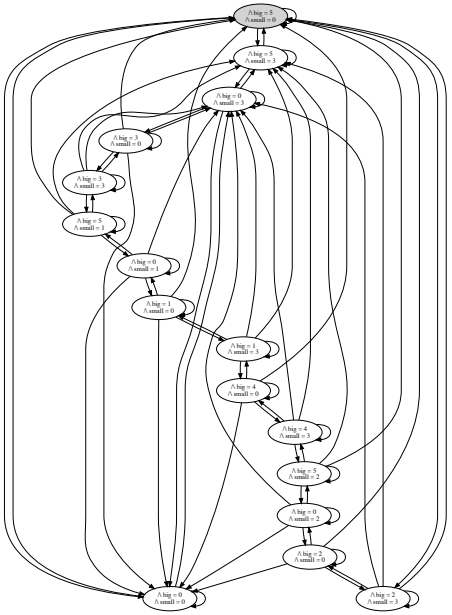
162-22



162-23



162-24



163

Die Hard

164-1

Die Hard

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

164-2

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**.

164-3

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?

Invariants

Invariants

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

Invariants

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

Invariants

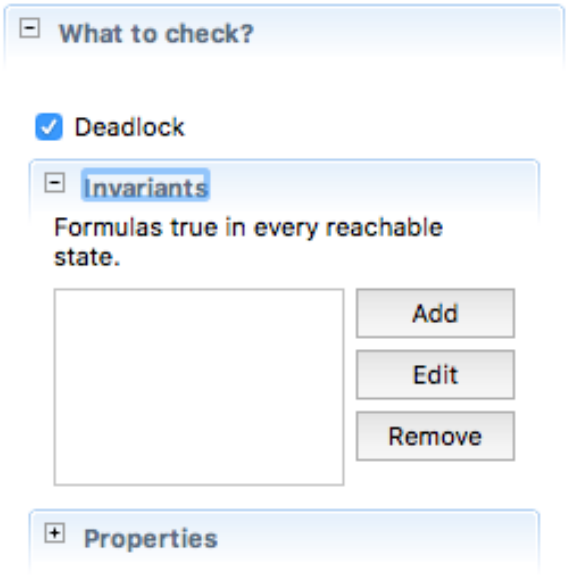
- It's time to explore the true power of TLA+ and TLC
- An invariant is "**something that should always be true**"
  - Or "**something that should never occur**"

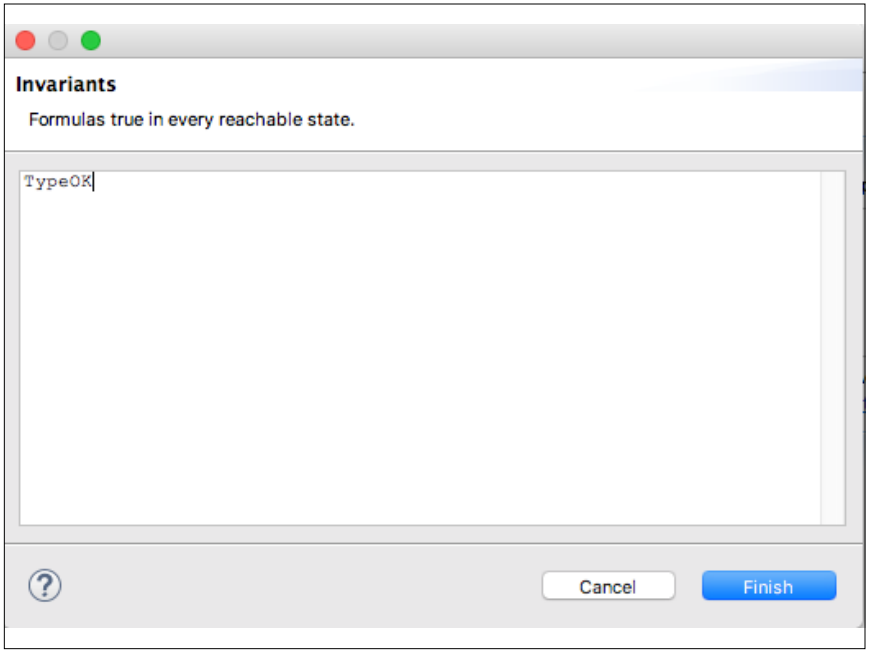
Invariants

- 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

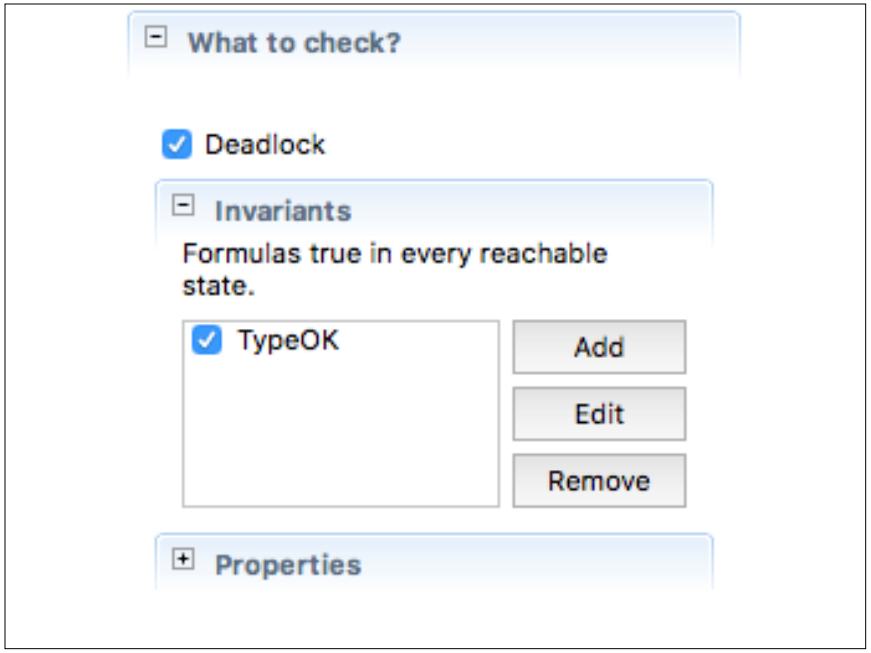
Invariants

- 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

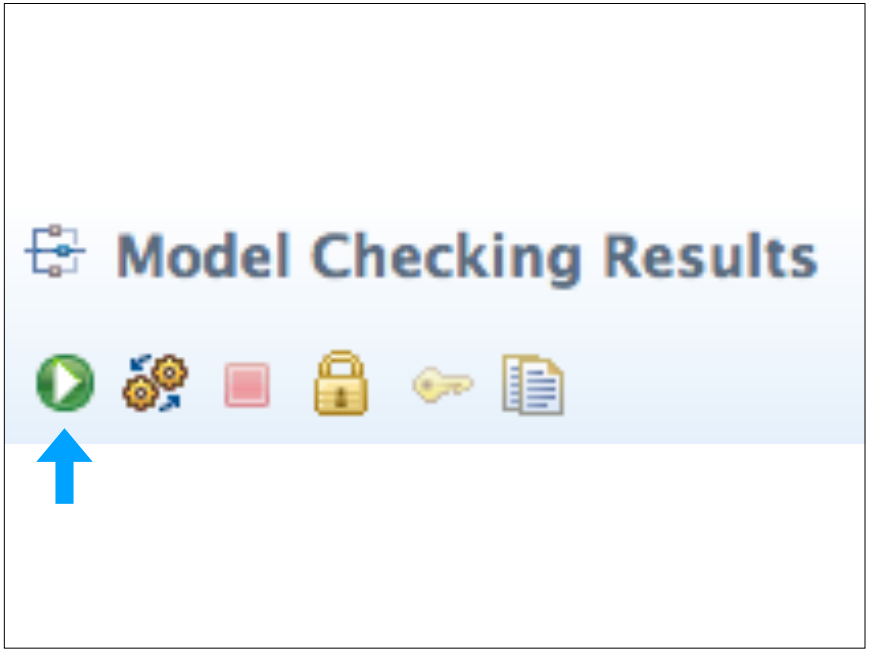




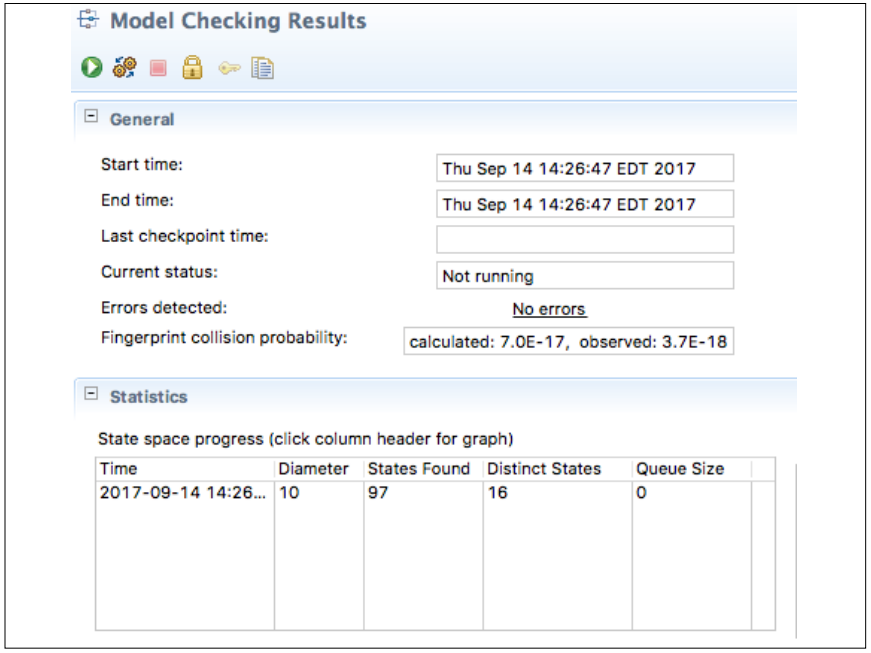
167



168

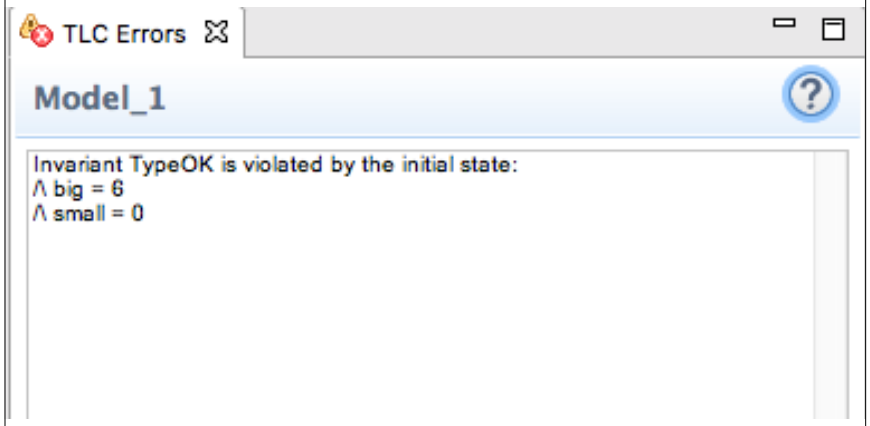


169



170

```
Init == /\ big   = 6
        /\ small = 0
```

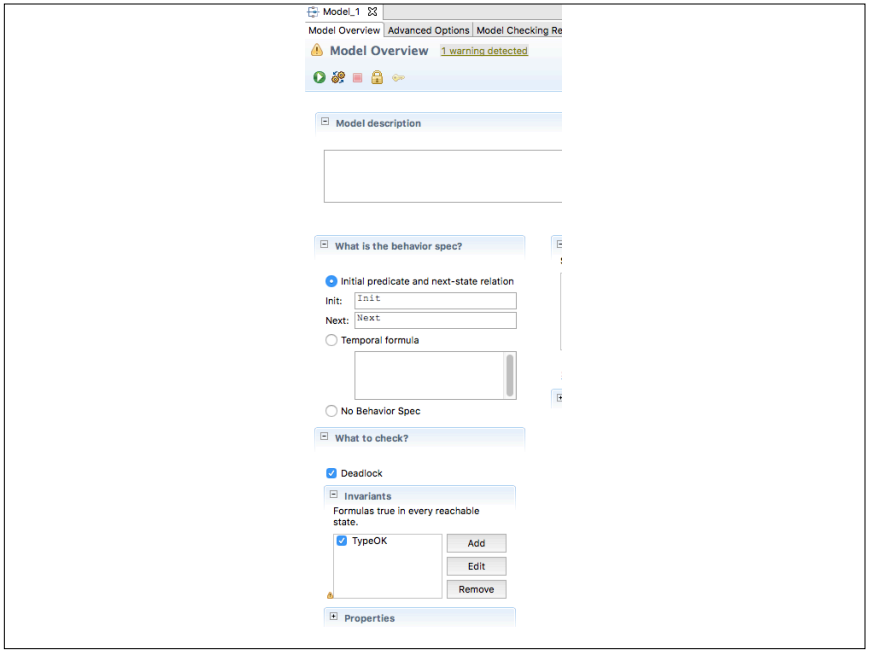



TypeOK

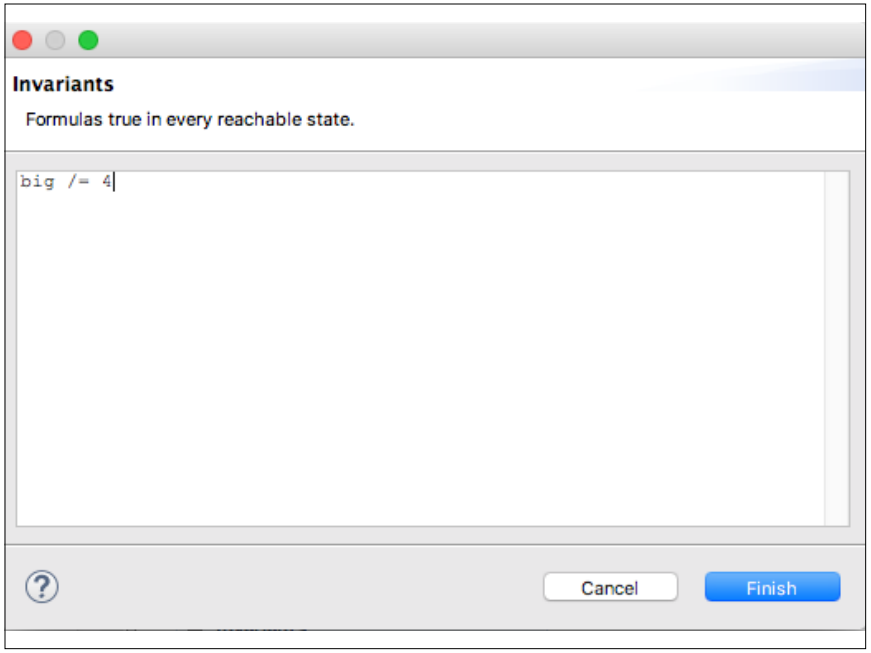
```
TypeOK == /\ small \in 0..3
          /\ big   \in 0..5
```



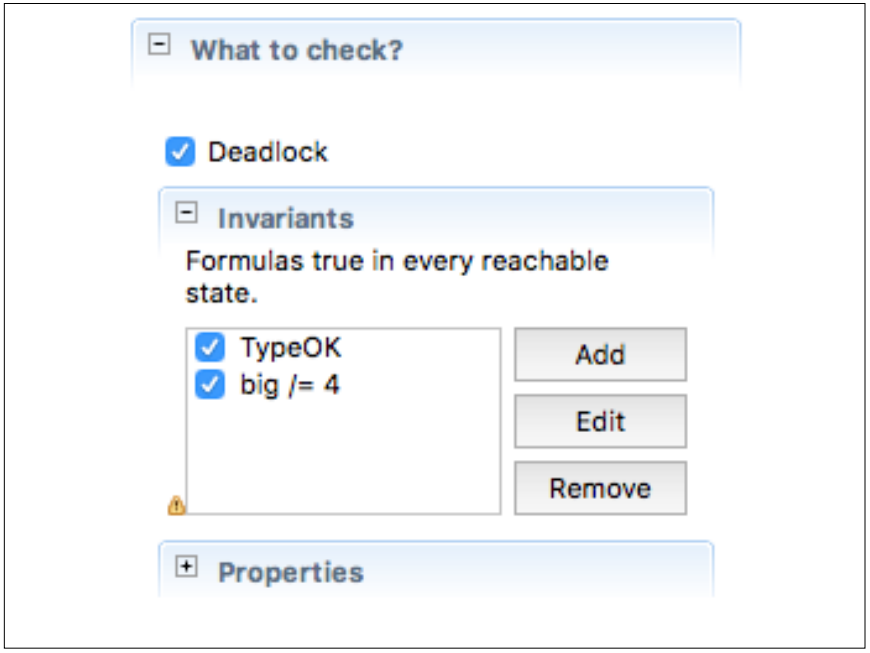
```
Init == /\ big   = 5
        /\ small = 0
```



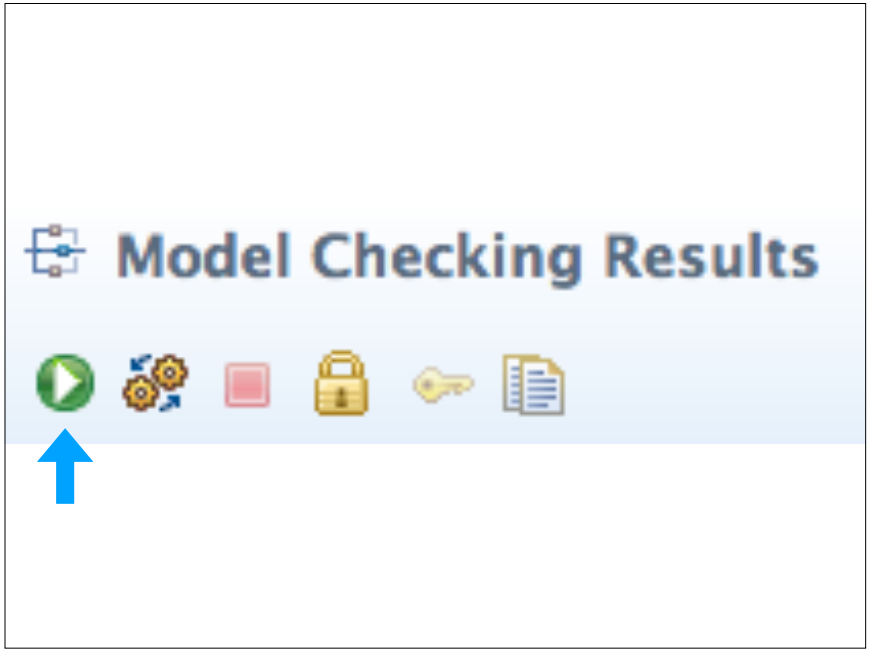
175



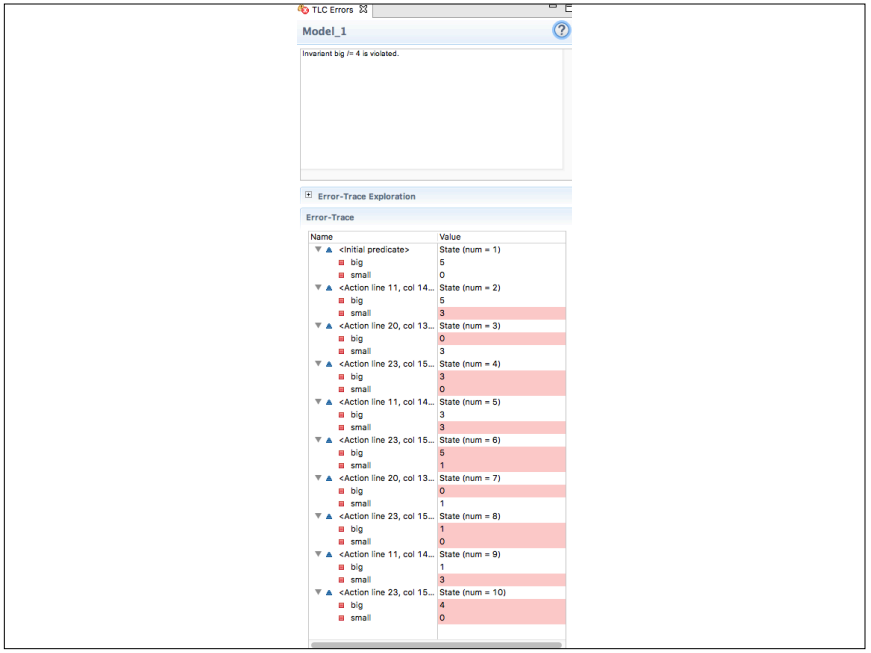
176



177

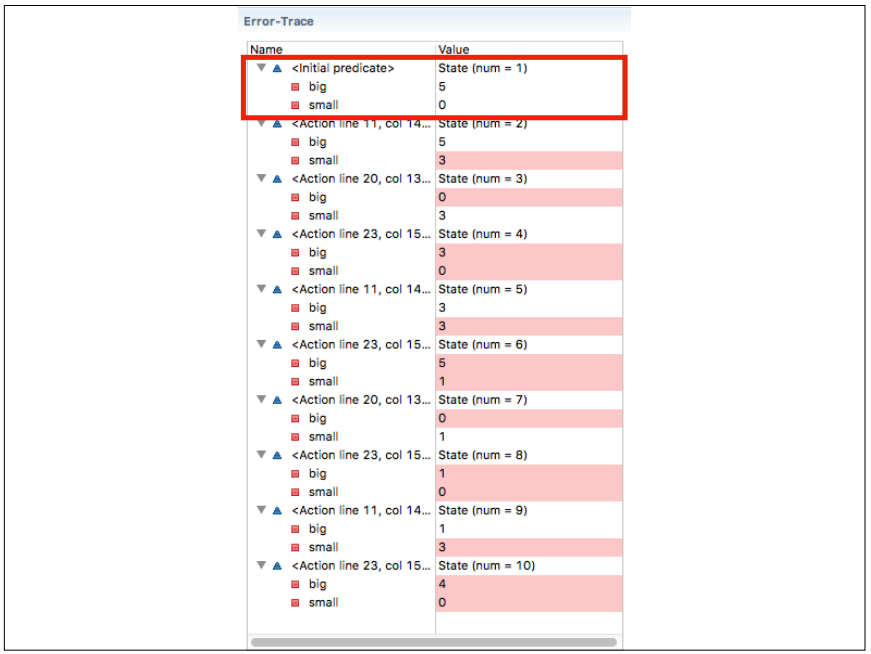


178

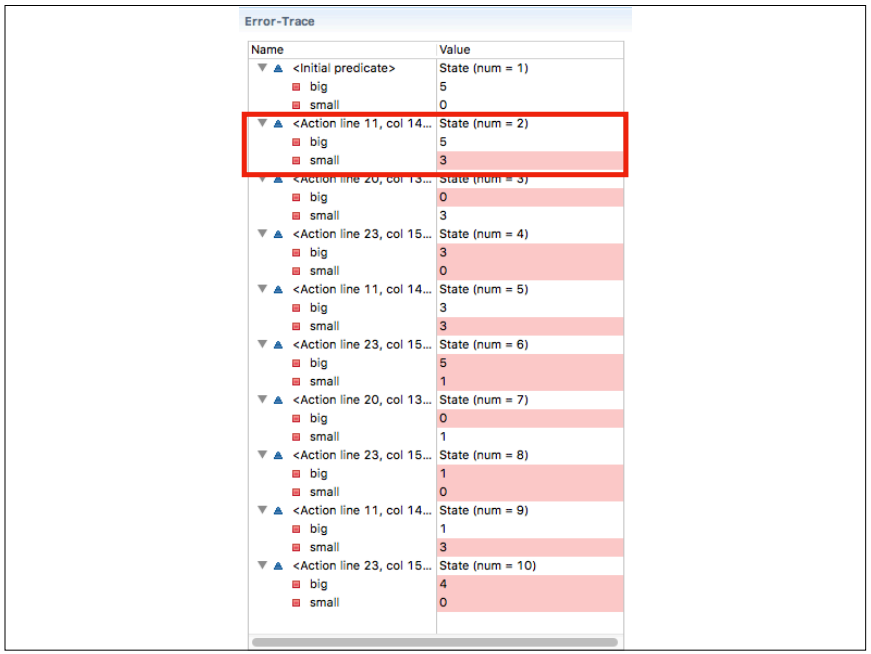


179

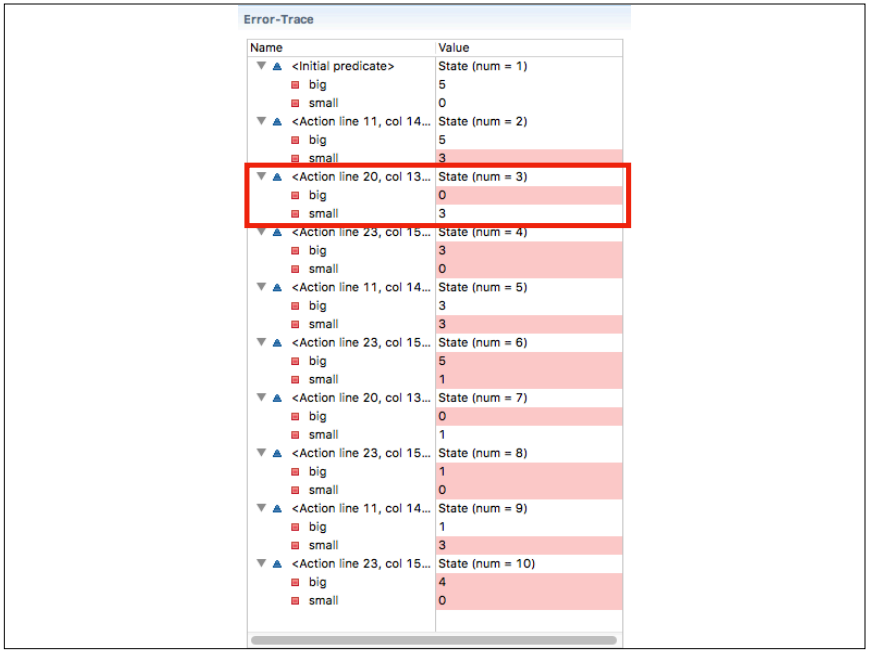
180-1



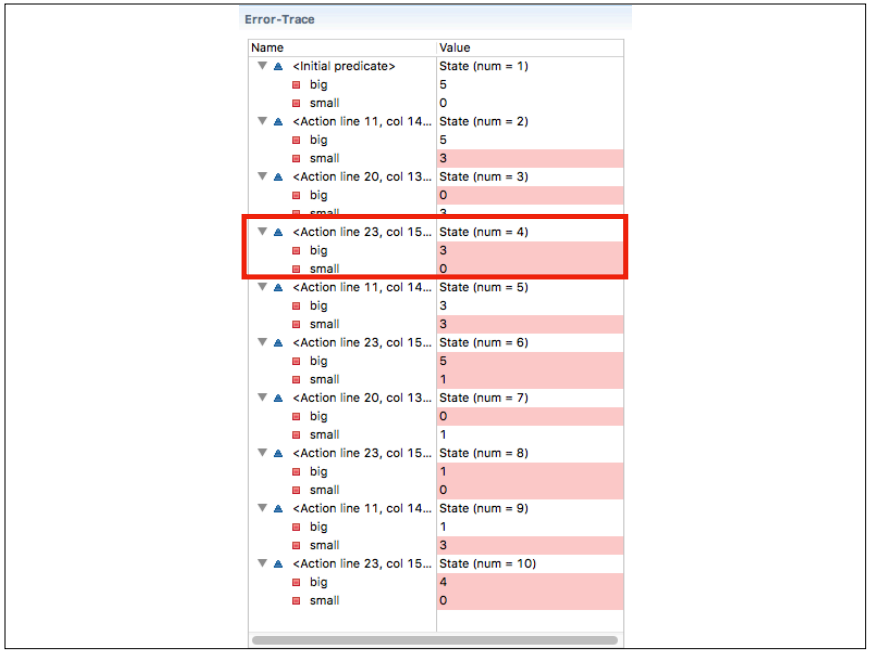
180-2



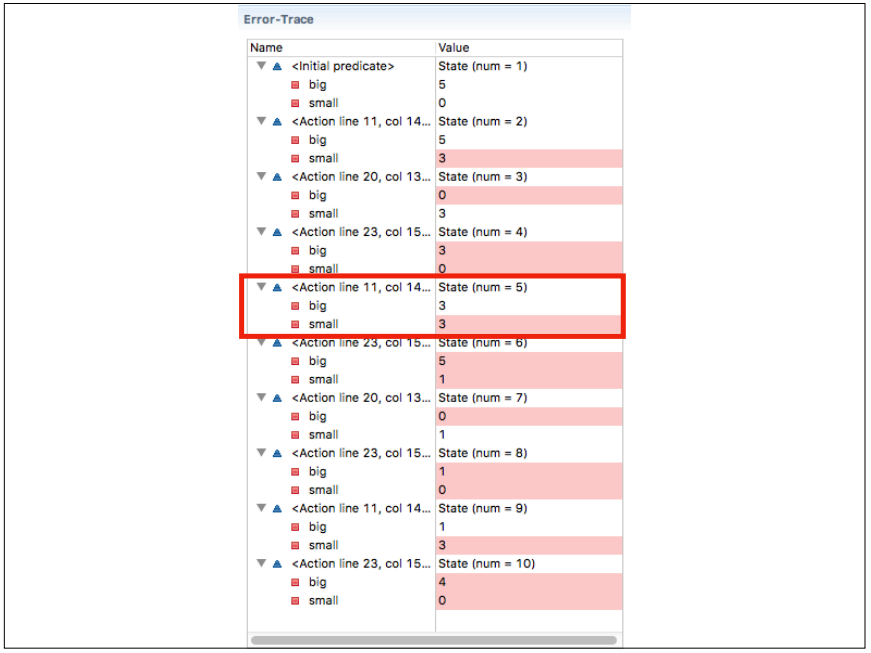
180-3



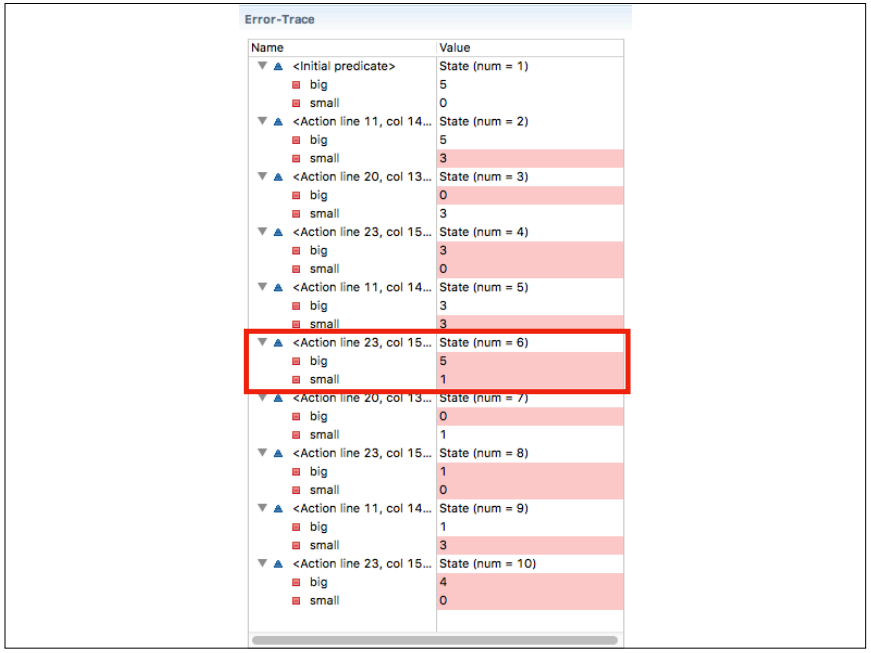
180-4



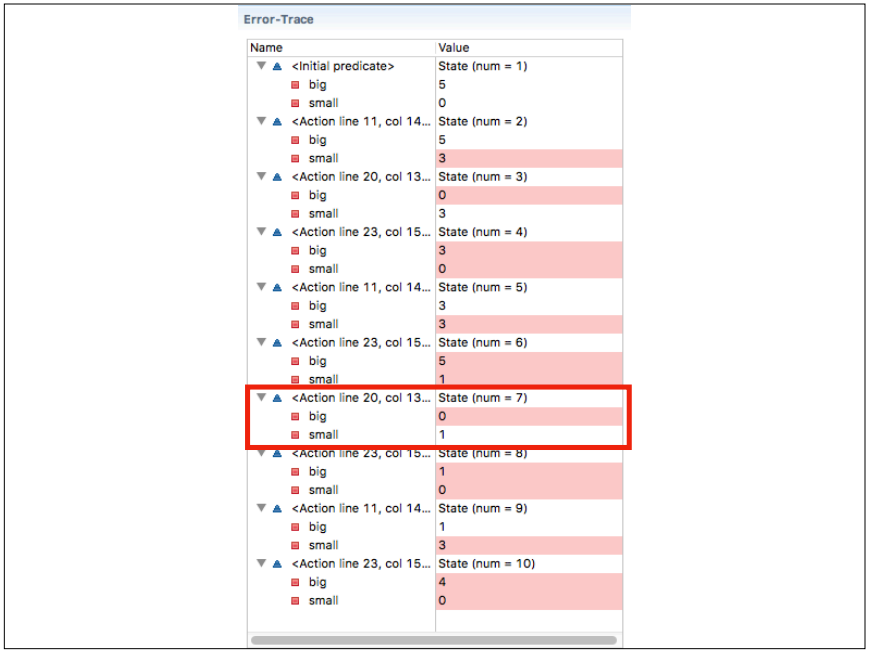
180-5



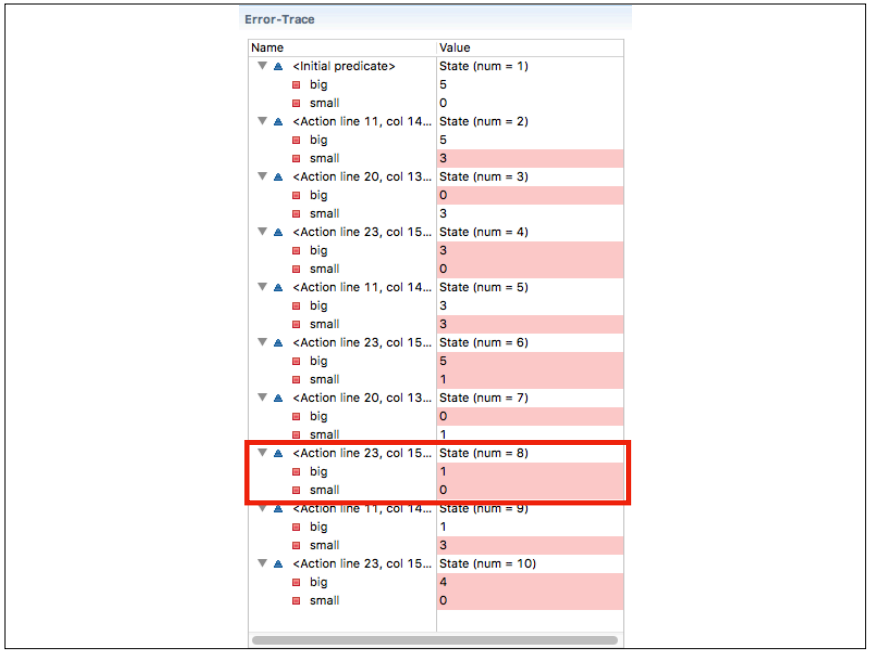
180-6



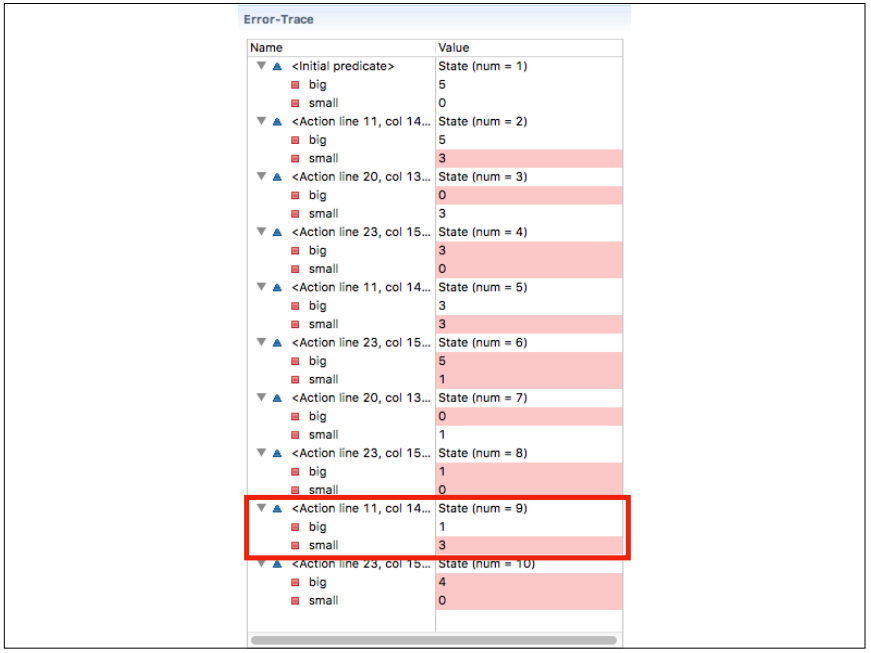
180-7



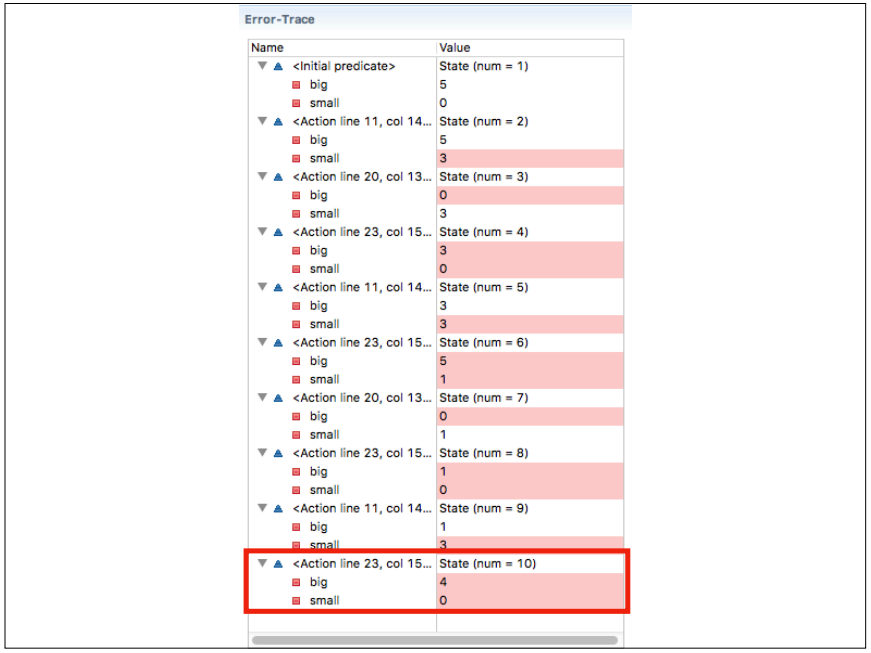
180-8



180-9



180-10



180-11

Woah

Woah

- This is amazingly powerful.

Woah

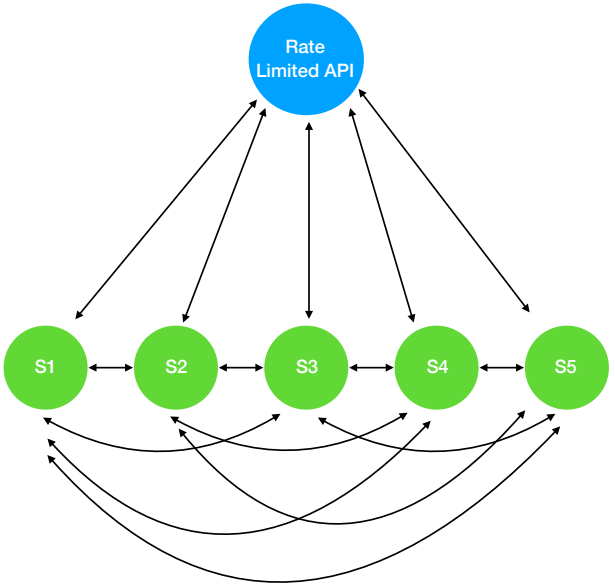
- This is amazingly powerful.
- TLC tries every single possible combination of states (i.e. behaviour/execution), and checks invariants for every state in every behaviour.

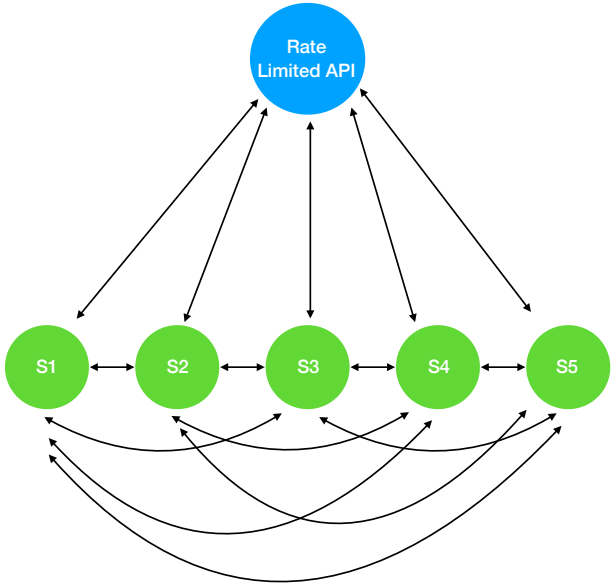
Woah

- 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.

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.





https://learntla.com/concurrency/example/

AWS Fault-Tolerant Replication Algorithm

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.

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.

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.

Transaction Commit

Transaction Commit

- Consider a replicated database.

Transaction Commit

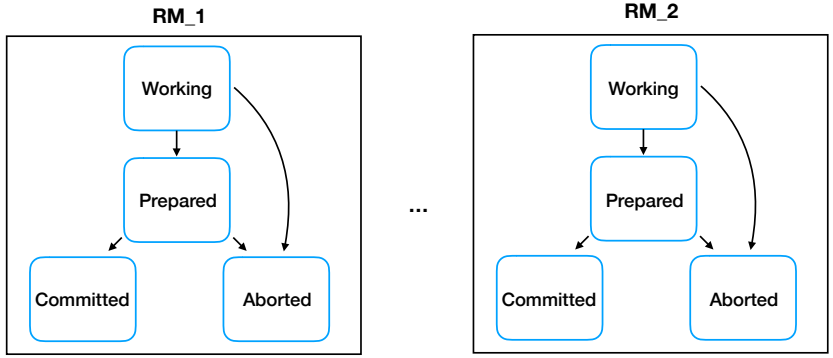
- Consider a replicated database.
- Multiple “resource managers,” each carrying a full replica.

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.

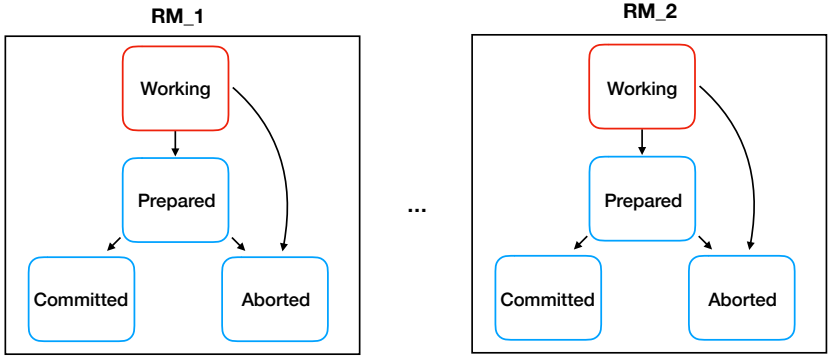
Resource Managers

Resource Managers



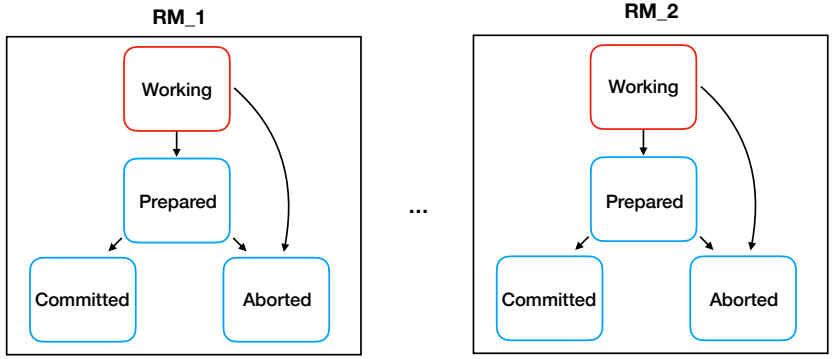
186

Resource Managers



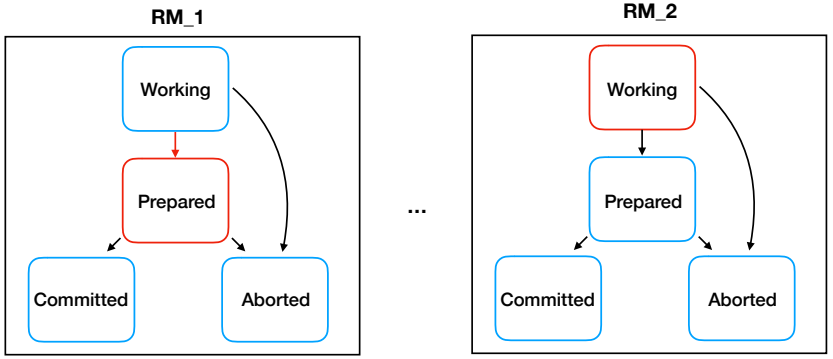
187

Resource Managers



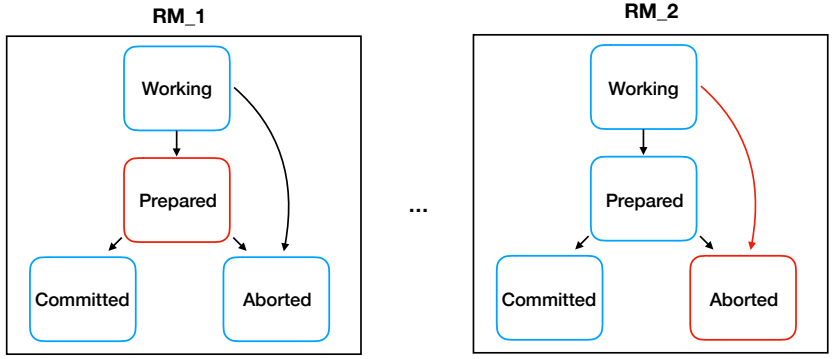
188

Resource Managers

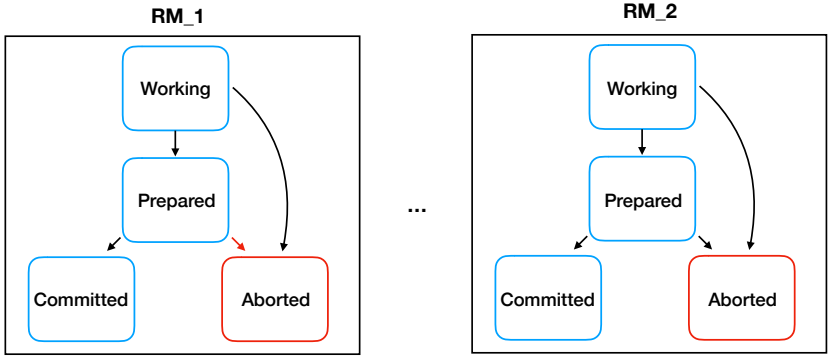


189

Resource Managers



Resource Managers



Resource Managers

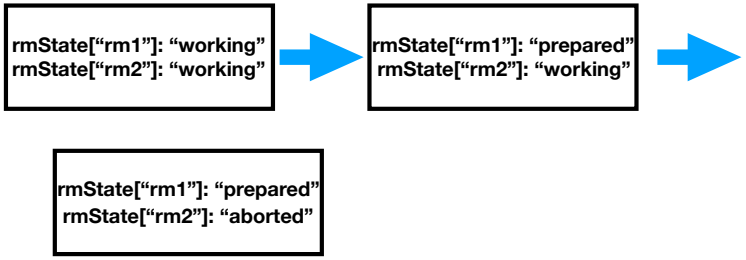
rmState["rm1"]: "working"
rmState["rm2"]: "working"

Resource Managers

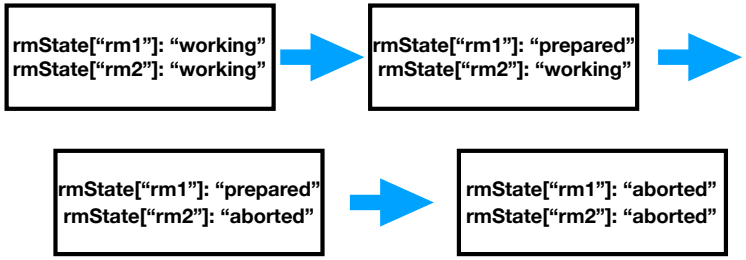
rmState["rm1"]: "working"
rmState["rm2"]: "working"

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

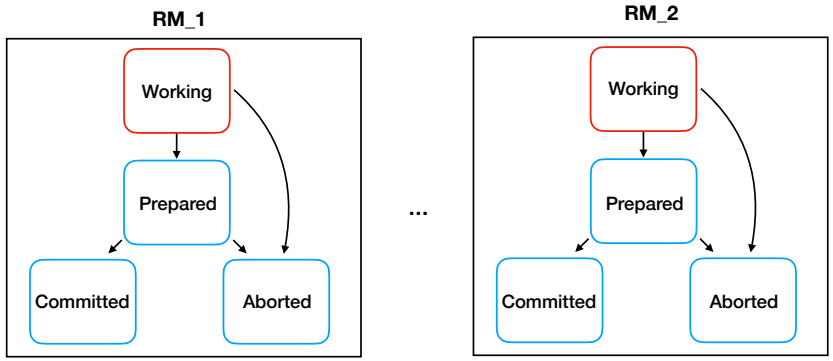
Resource Managers



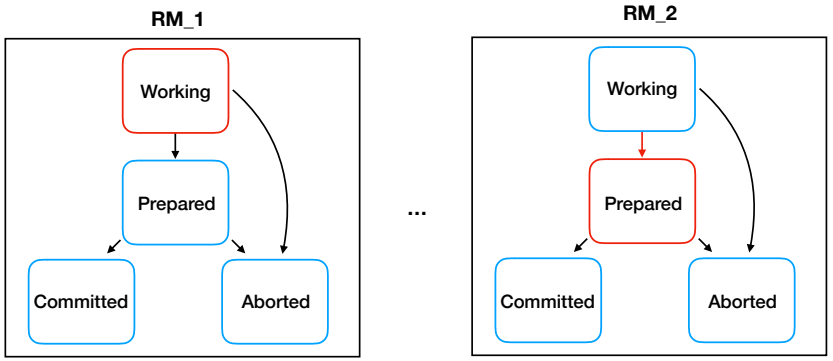
Resource Managers



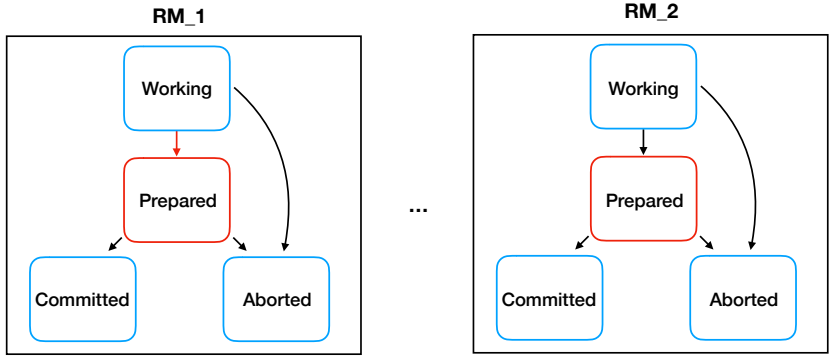
Resource Managers



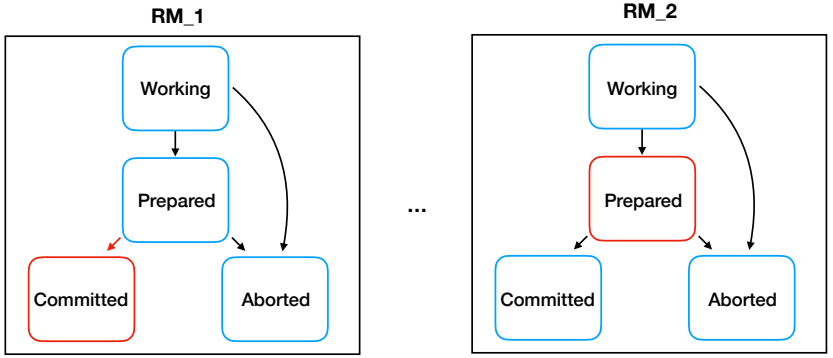
Resource Managers



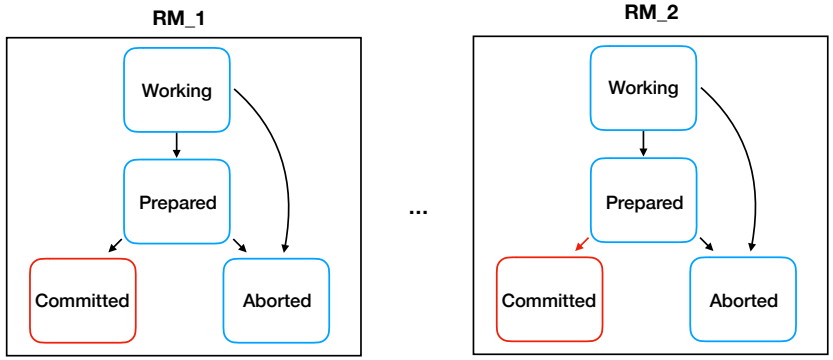
Resource Managers



Resource Managers



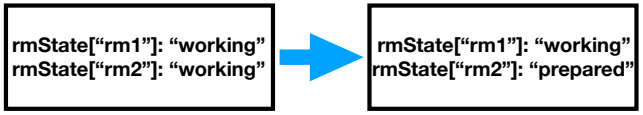
Resource Managers



Resource Managers

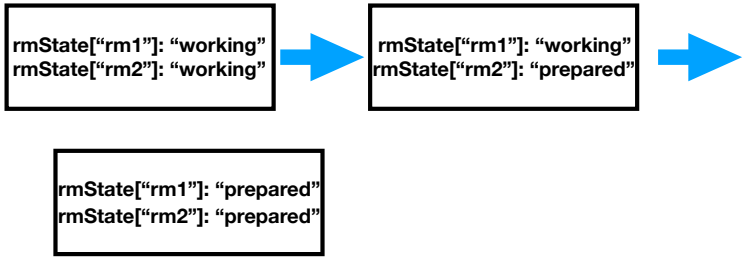
rmState[“rm1”]: “working”
rmState[“rm2”]: “working”

Resource Managers



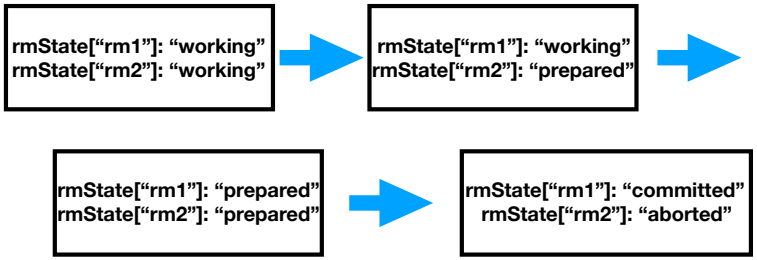
198-2

Resource Managers



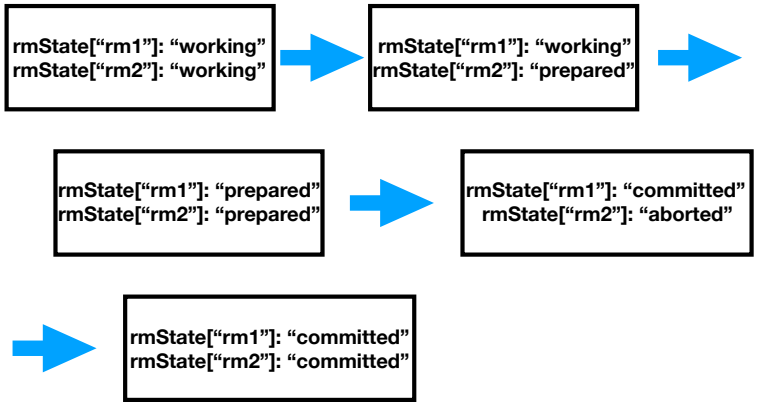
198-3

Resource Managers



198-4

Resource Managers



198-5

Resource Managers

Resource Managers

- We just looked at two possible behaviours.

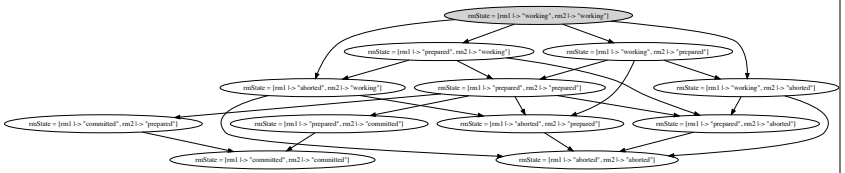
Resource Managers

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

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.

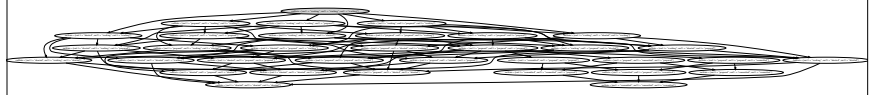
2 Resource Managers



12 Distinct States

200

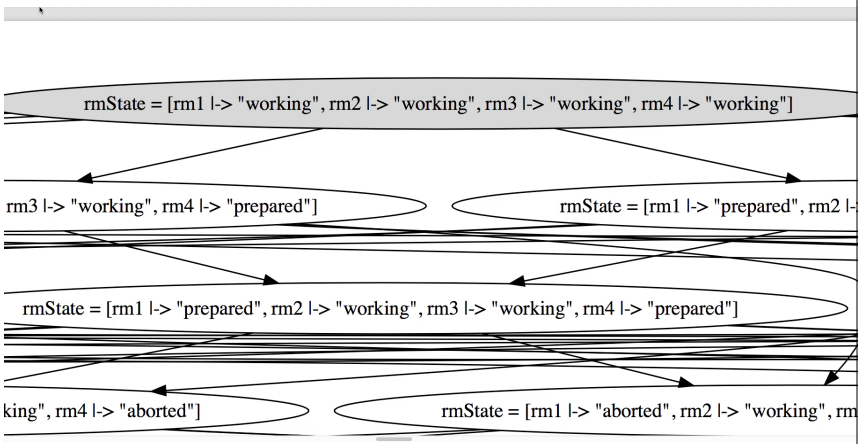
3 Resource Managers



34 Distinct States

201

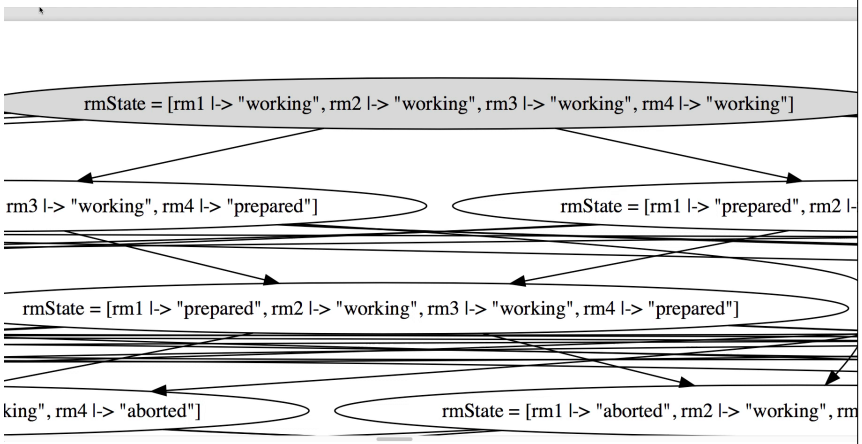
4 Resource Managers



96 Distinct States

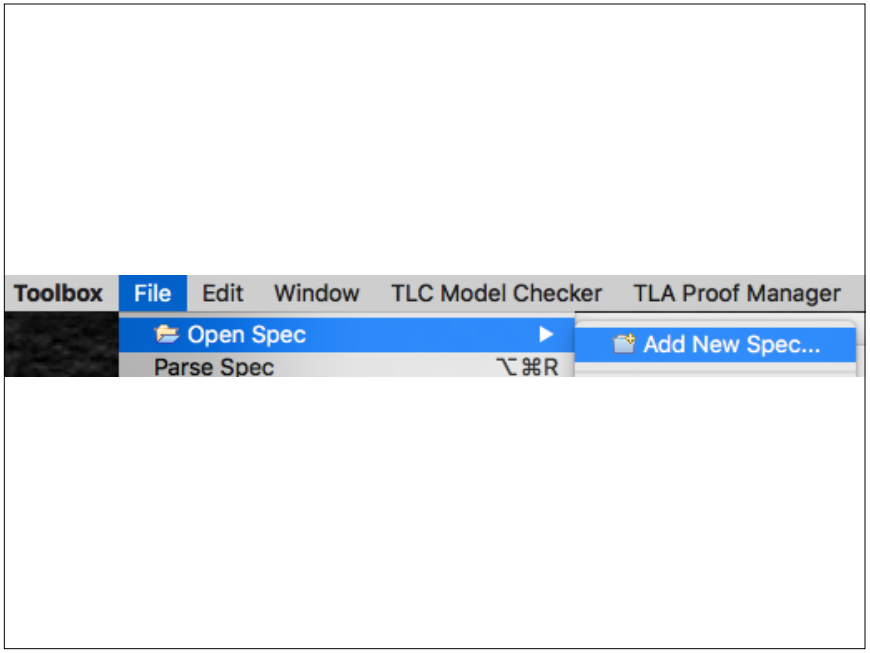
202-1

4 Resource Managers

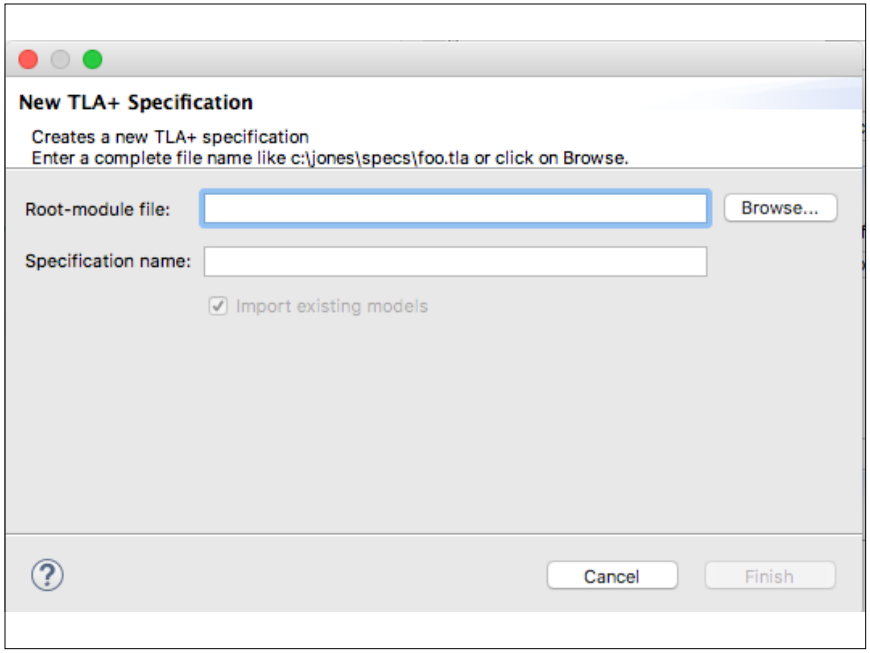


96 Distinct States

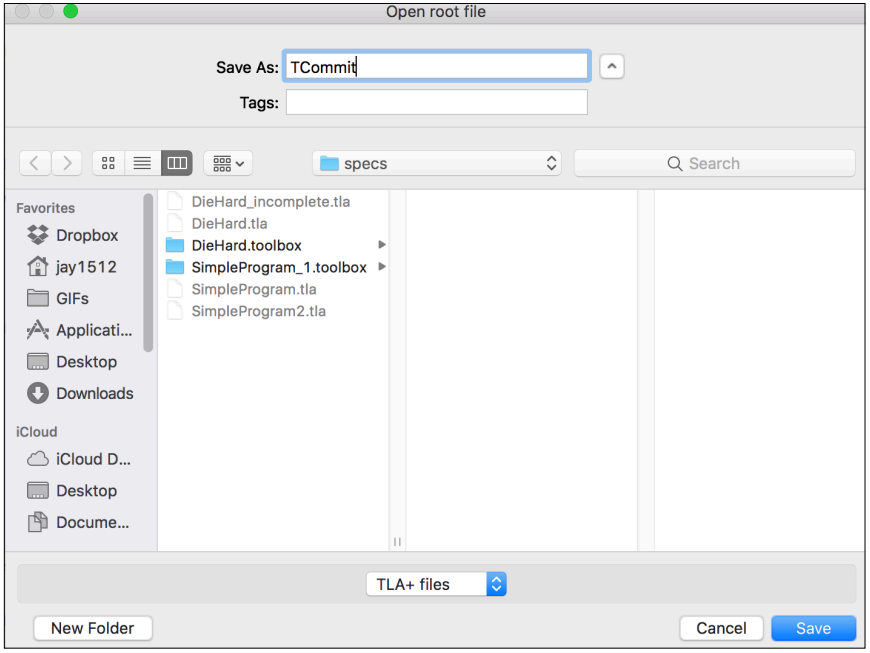
202-2



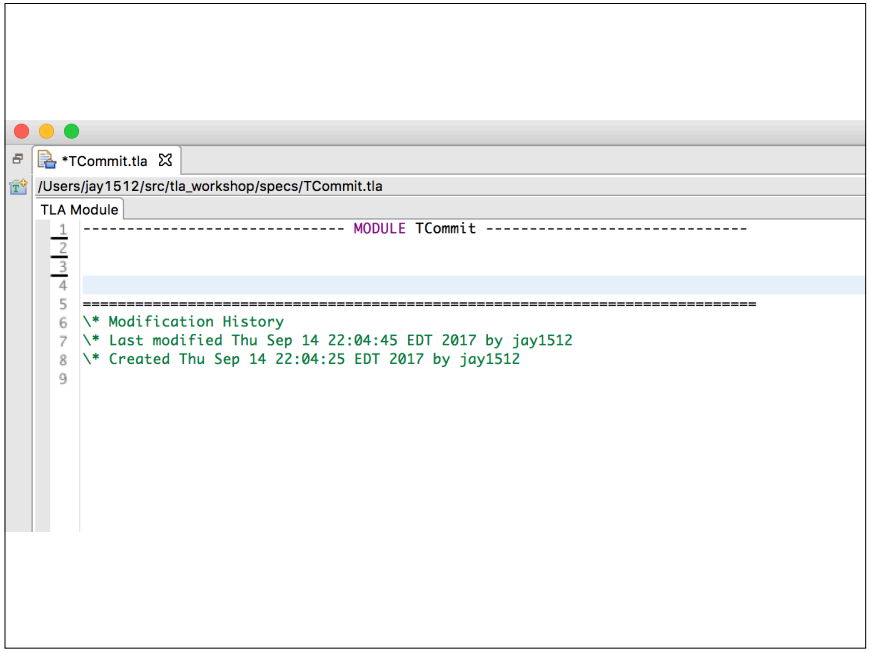
203



204



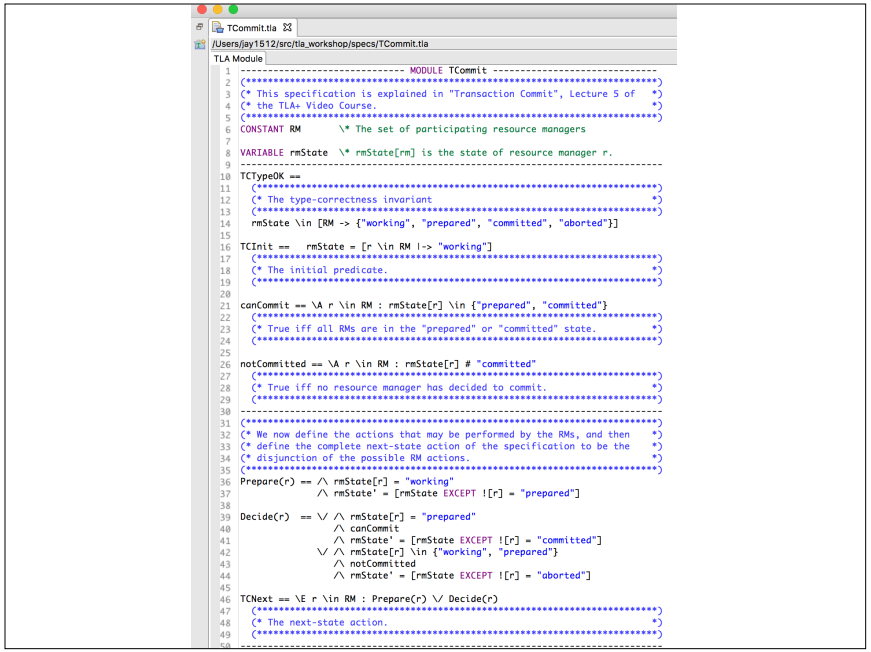
205



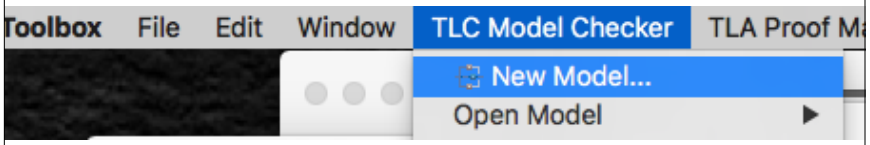
206

specs/TCommit.tla

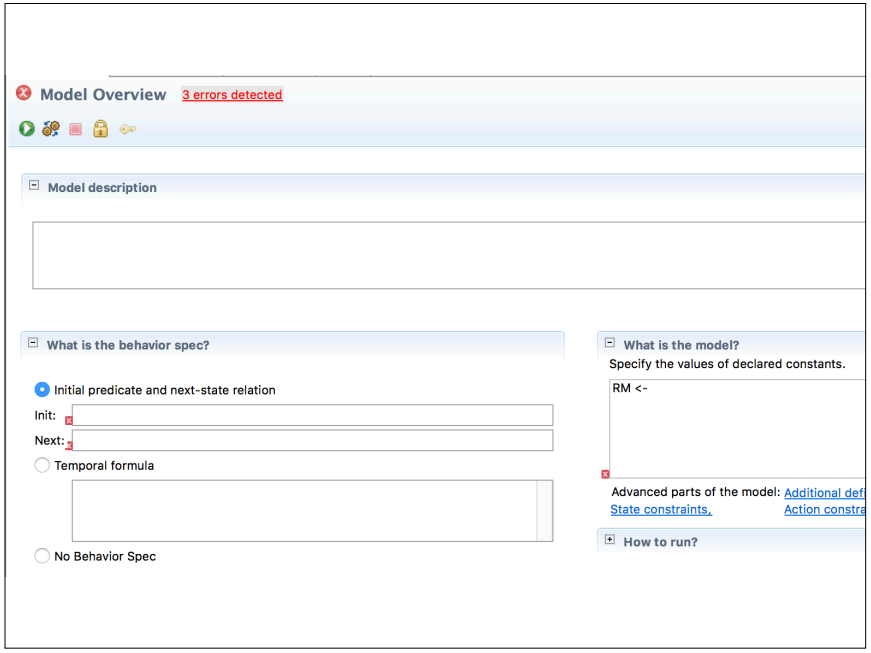
207



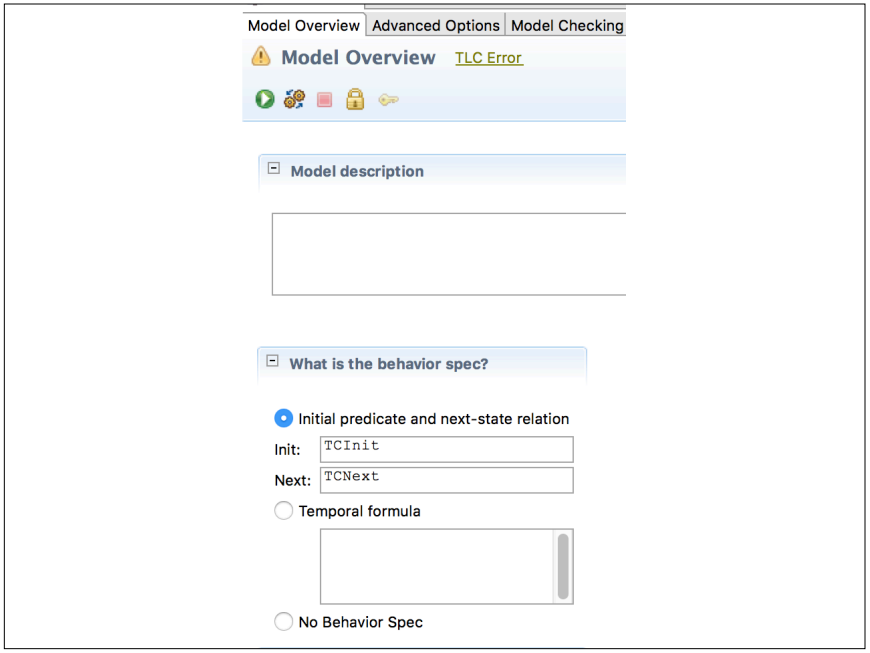
208



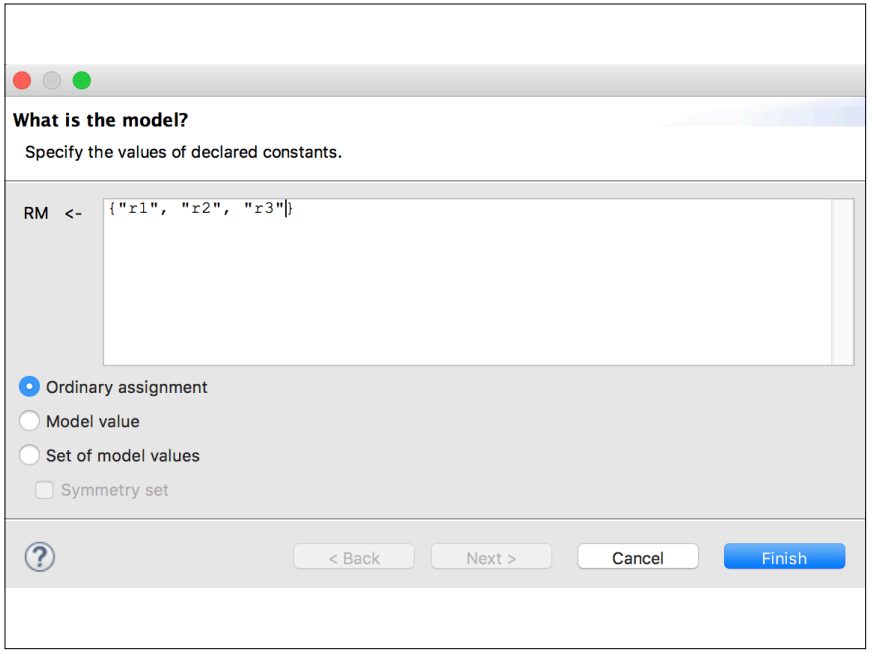
209



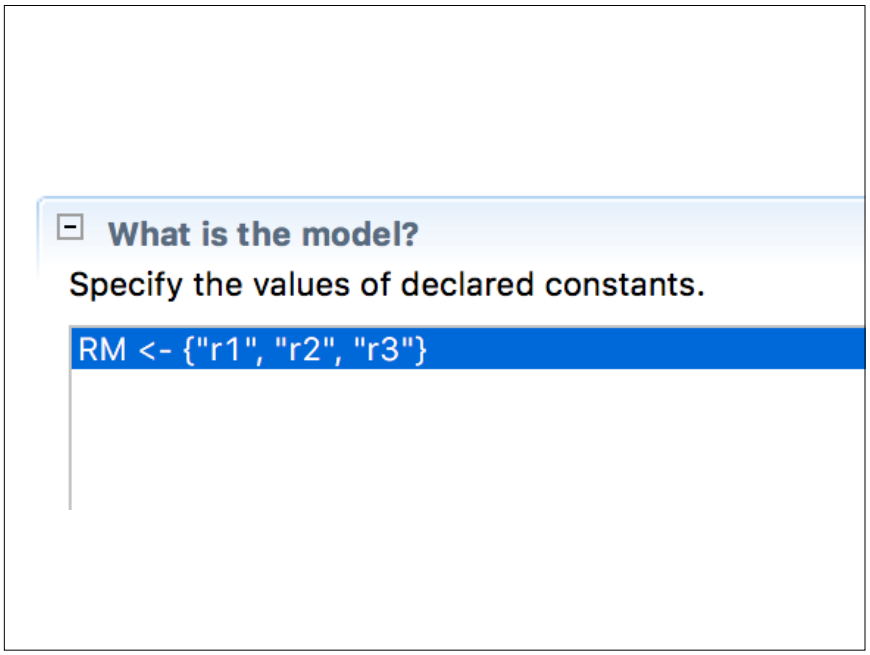
210



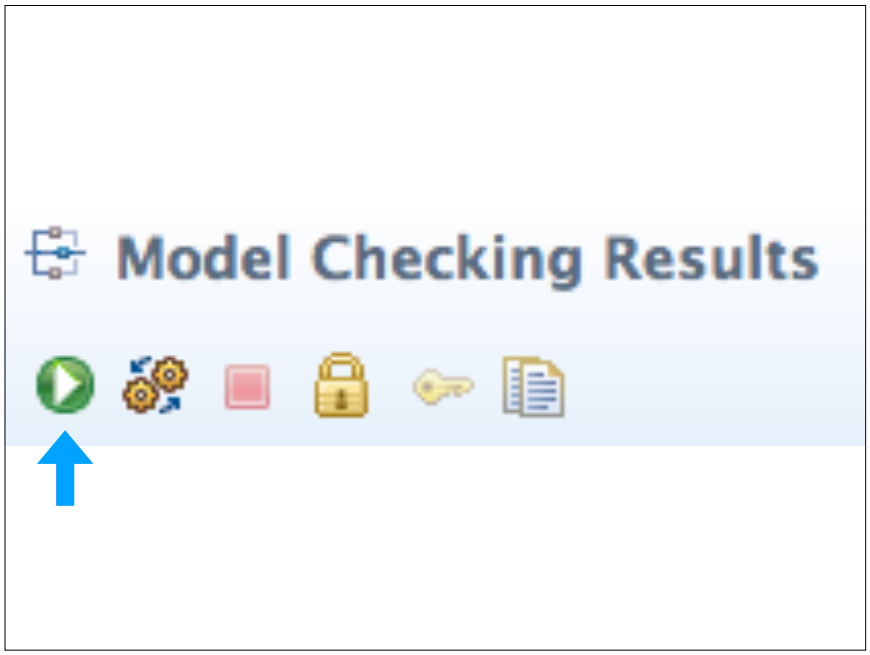
211



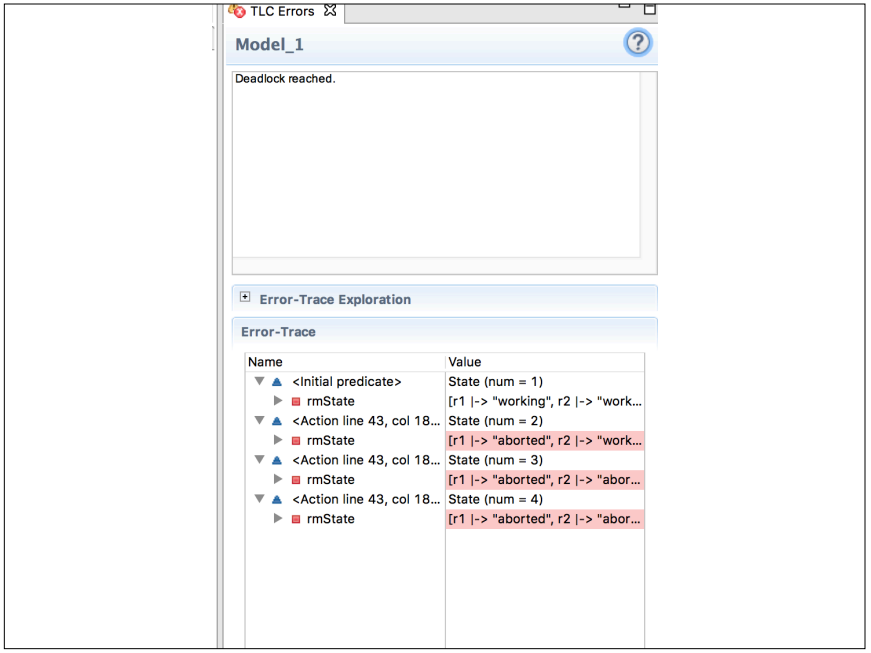
212



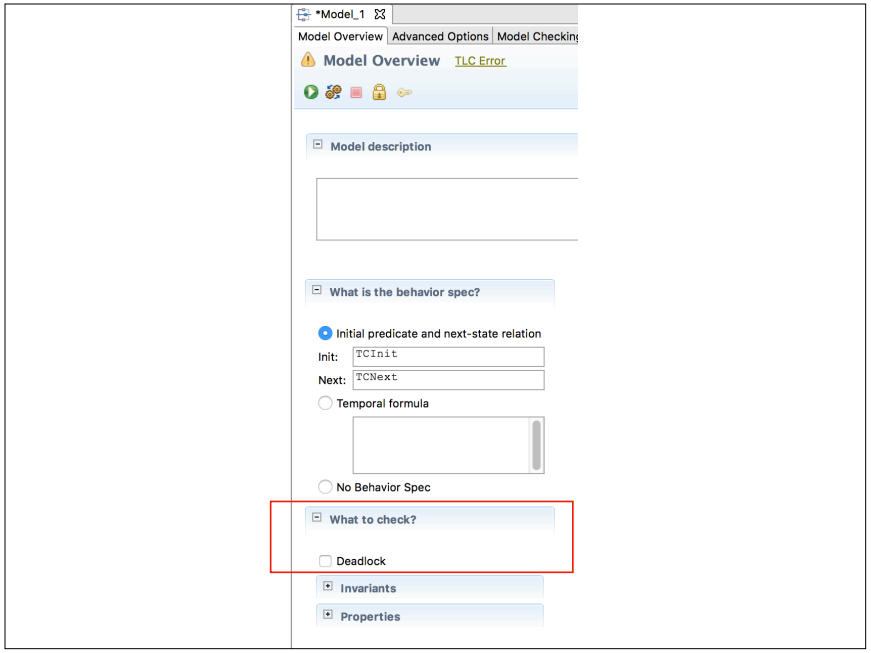
213



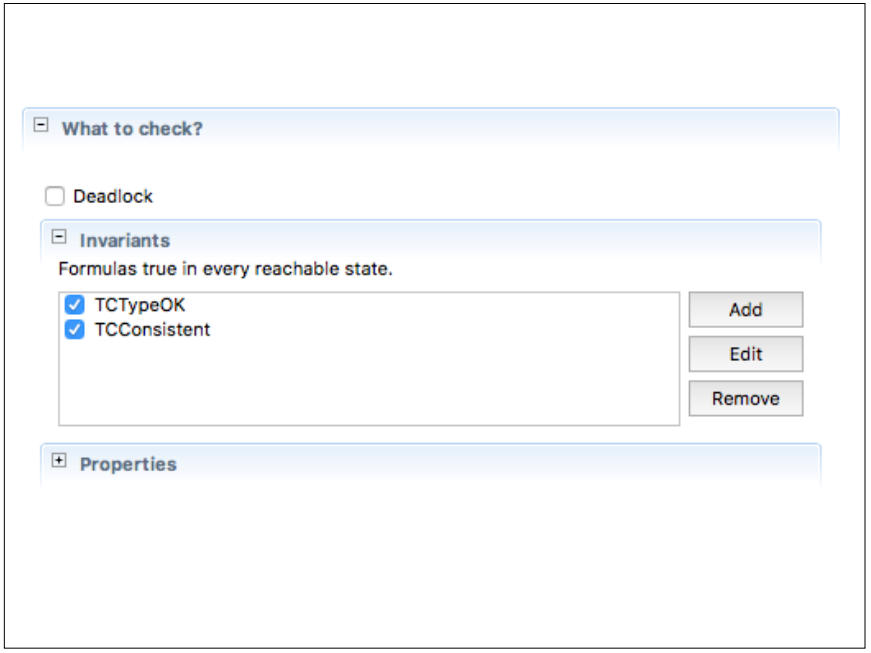
214



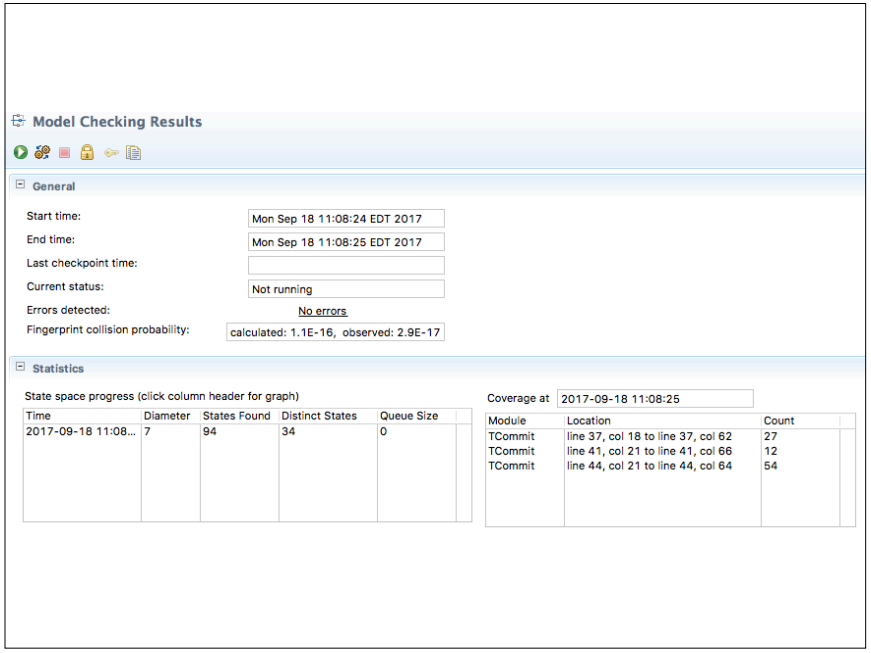
215



216



217



218

Let's try breaking something!


```
Decide(r)  == \/ /\ rmState[r] = "prepared"
                 \* /\ canCommit
                 /\ rmState' = [rmState EXCEPT ![r] = "committed"]
              \/ /\ rmState[r] \in {"working", "prepared"}
                 /\ notCommitted
                 /\ rmState' = [rmState EXCEPT ![r] = "aborted"]
```



```
Decide(r)  == \/ /\ rmState[r] = "prepared"
                 \* /\ canCommit
                 /\ rmState' = [rmState EXCEPT ![r] = "committed"]
              \/ /\ rmState[r] \in {"working", "prepared"}
                 /\ notCommitted
                 /\ rmState' = [rmState EXCEPT ![r] = "aborted"]
```

- The ***** is a TLA+ single-line comment, we're temporarily removing the **canCommit** requirement.

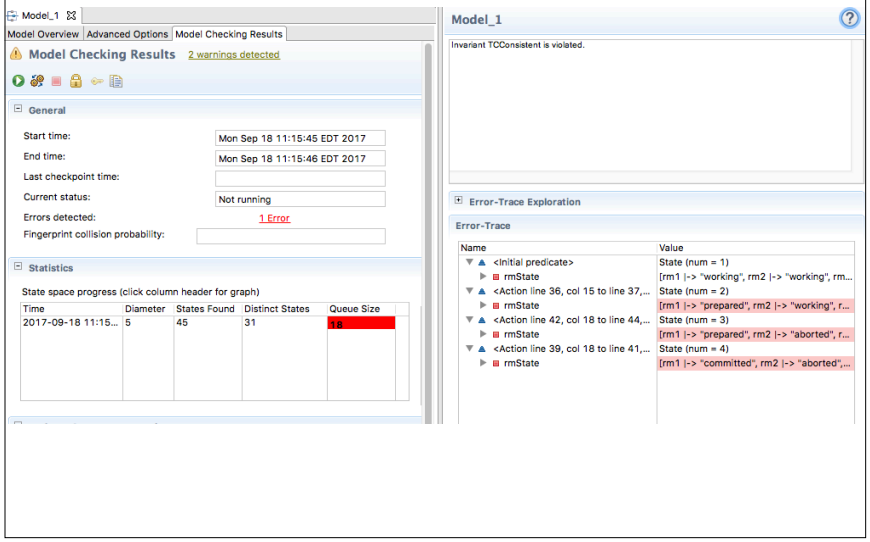

```
Decide(r)  == \/ /\ rmState[r] = "prepared"
                 \* /\ canCommit
                 /\ rmState' = [rmState EXCEPT ![r] = "committed"]
              \/ /\ rmState[r] \in {"working", "prepared"}
                 /\ notCommitted
                 /\ rmState' = [rmState EXCEPT ![r] = "aborted"]
```

- 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."

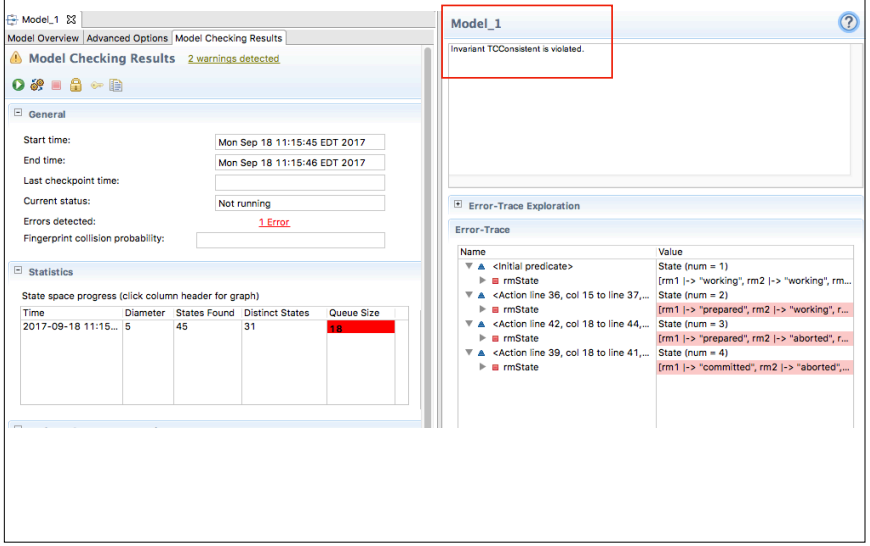
```
Decide(r)  == \/ /\ rmState[r] = "prepared"
                 \* /\ canCommit
                 /\ rmState' = [rmState EXCEPT ![r] = "committed"]
              \/ /\ rmState[r] \in {"working", "prepared"}
                 /\ notCommitted
                 /\ rmState' = [rmState EXCEPT ![r] = "aborted"]
```

- The ***** is a TLA+ single-line comment, we're temporarily removing the **canCommit** requirement.
- This means that **canCommit** no longer needs to be true to allow a state transition where some RM **r** goes from "prepared" to "committed."
- There could now be some other RM in an "aborted" state, and the specification would permit this transition.

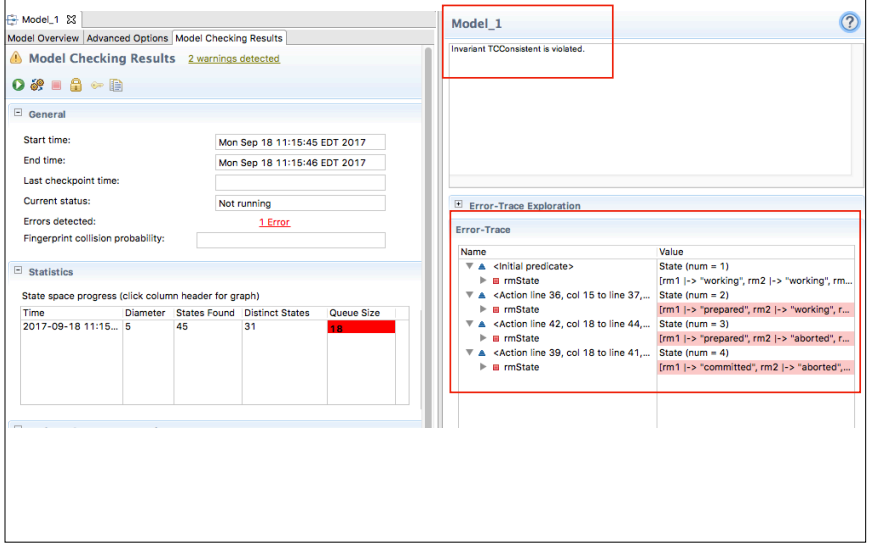
220-4



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

```
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) == /\ rmState[r] = "working"
37              /\ rmState' = [rmState EXCEPT ![r] = "prepared"]
38
39 Decide(r)  == \/ /\ rmState[r] = "prepared"
40                  \* /\ canCommit
41                  /\ rmState' = [rmState EXCEPT ![r] = "committed"]
42               \/ /\ rmState[r] \in {"working", "prepared"}
43                  /\ notCommitted
44                  /\ rmState' = [rmState EXCEPT ![r] = "aborted"]
45
```

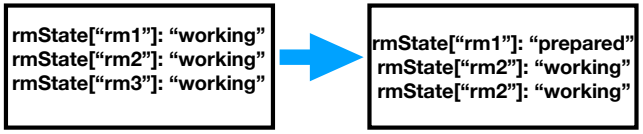
224

Resource Managers

rmState[“rm1”]: “working”
rmState[“rm2”]: “working”
rmState[“rm3”]: “working”

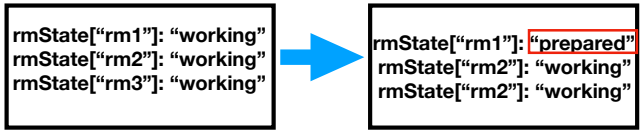
225-1

Resource Managers



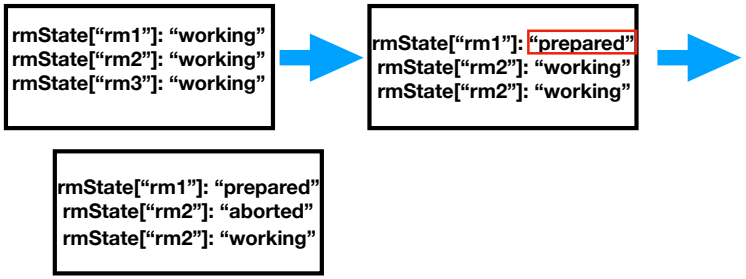
225-2

Resource Managers



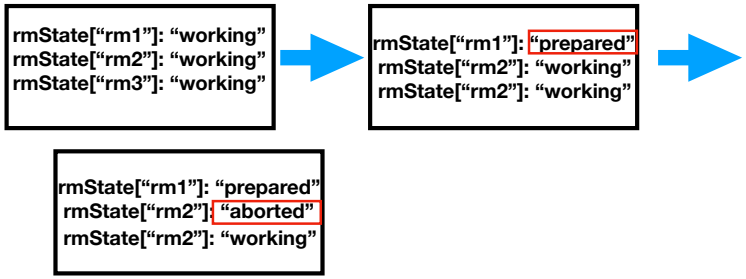
225-3

Resource Managers



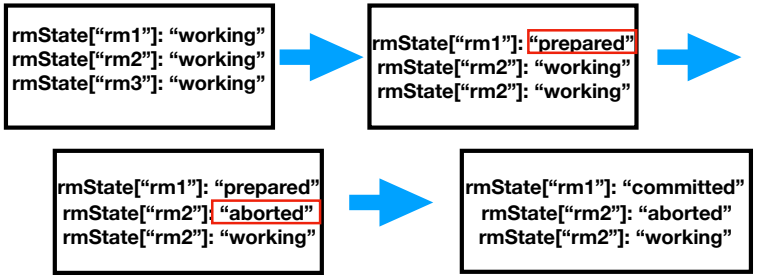
225-4

Resource Managers

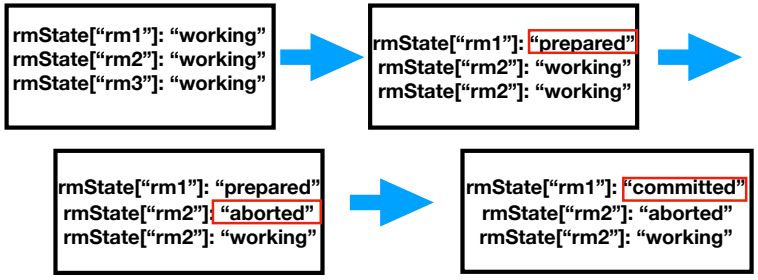


225-5

Resource Managers



Resource Managers



Syntax time

Functions

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"

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

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.

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

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

Functions

Functions

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

Functions

- TLA+ provides lots of ways to create functions. We'll only look at the commonly used ones
- [i ∈ S |-> 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**

Functions

- TLA+ provides lots of ways to create functions. We'll only look at the commonly used ones
- [i ∈ S |-> 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 ∈ {2,3,4} |-> i*2] = (2 :> 4 @@ 3 :> 6 @@ 4 :> 8)

Functions

- TLA+ provides lots of ways to create functions. We'll only look at the commonly used ones
- [i ∈ S |-> 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 ∈ {2,3,4} |-> i*2] = (2 :> 4 @@ 3 :> 6 @@ 4 :> 8)

- Read this as: "Take the set to the left of **|->** and use that as the keys. For each key, evaluate the expression to the right of **|->**, and use that as the value"

Functions

- TLA+ provides lots of ways to create functions. We'll only look at the commonly used ones
- [i ∈ S |-> 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 ∈ {2,3,4} |-> i*2] = (2 :> 4 @@ 3 :> 6 @@ 4 :> 8)

- Read this as: "Take the set to the left of **|->** and use that as the keys. For each key, evaluate the expression to the right of **|->**, 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}

Functions

- TLA+ provides lots of ways to create functions. We'll only look at the commonly used ones
- [i ∈ S |-> 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 ∈ {2,3,4} |-> i*2] = (2 :> 4 @@ 3 :> 6 @@ 4 :> 8)

- Read this as: "Take the set to the left of **|->** and use that as the keys. For each key, evaluate the expression to the right of **|->**, 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"

Functions

- TLA+ provides lots of ways to create functions. We'll only look at the commonly used ones
- [i ∈ S |-> 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 ∈ {2,3,4} |-> i*2] = (2 :> 4 @@ 3 :> 6 @@ 4 :> 8)
- Read this as: "Take the set to the left of **|->** and use that as the keys. For each key, evaluate the expression to the right of **|->**, 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]}
```


Functions

- The value for some key can be accessed using the same syntax as array/dictionary access in Python

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

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


Functions

Functions

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

Functions

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

```
DOMAIN [ i ∈ {2, 4, 6} |-> i*2 ] = {2,4,6}
```


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?

Functions

Functions

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

Functions

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

Answer:

Functions

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

Answer:

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


Sets of functions

Sets of functions

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

Sets of functions

- It is very common in TLA+ to need to construct sets of functions.
- The general syntax is `[S -> T]`, where **S** and **T** are sets.

Sets of functions

- It is very common in TLA+ to need to construct sets of functions.
- The general syntax is `[S -> T]`, where **S** and **T** are sets.

> NOTE: That's **->**, not the **|->** we just looked at. This will screw you up at some point.

Sets of functions

- It is very common in TLA+ to need to construct sets of functions.
- The general syntax is `[S -> T]`, where **S** and **T** are sets.

> NOTE: That's **->**, not the **|->** we just looked at. This will screw you up at some point.

- `[S -> T]` means

Sets of functions

- It is very common in TLA+ to need to construct sets of functions.
- The general syntax is `[S -> T]`, where **S** and **T** are sets.

> NOTE: That's **->**, not the **|->** we just looked at. This will screw you up at some point.

- `[S -> T]` means
  - Create a set of functions.

Sets of functions

- It is very common in TLA+ to need to construct sets of functions.
- The general syntax is `[S -> T]`, where **S** and **T** are sets.

  NOTE: That's **->**, not the **|->** we just looked at. This will screw you up at some point.
- `[S -> T]` means
  - Create a set of functions.
  - Each function in the set will have the domain **S**.

Sets of functions

- It is very common in TLA+ to need to construct sets of functions.
- The general syntax is `[S -> T]`, where **S** and **T** are sets.

  NOTE: That's **->**, not the **|->** we just looked at. This will screw you up at some point.
- `[S -> 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**.

Sets of functions

- It is very common in TLA+ to need to construct sets of functions.
- The general syntax is `[S -> T]`, where **S** and **T** are sets.

  NOTE: That's **->**, not the **|->** we just looked at. This will screw you up at some point.
- `[S -> 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.

Sets of functions

- It is very common in TLA+ to need to construct sets of functions.
- The general syntax is `[S -> T]`, where **S** and **T** are sets.

  NOTE: That's **->**, not the **|->** we just looked at. This will screw you up at some point.
- `[S -> 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?

Sets of functions

```
[{2,3} -> {"a", “b”, “c”}] =   { (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”} }
```


:> 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 as the separator between **keys** and **values**
@@ is used to concatenate together two functions (remember, a function is a set of key/value pairs)

Tuples

Tuples

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

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

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

- 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 `<< >>`

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

EXCEPT

EXCEPT

- A final important construct is `EXCEPT`

EXCEPT

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

EXCEPT

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

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

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]
```


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:

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**

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
```


Summary

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**.

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.

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.


```
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"}]
```



```
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"}]
```

```
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"}]
```



```
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"}]
```



```
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"}


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

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

`[S -> T]` creates a set of records, where each record in the set has all the elements of **S** as keys, with every possible assignment of values in **T** to those keys


```
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


```
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


```
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

```
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


```
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

If **RM** is defined to be `{“r1”, “r2”}` (i.e. we have two resource managers), then the resulting set of functions is:


```
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

If **RM** is defined to be `{“r1”, “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"] }
```



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

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

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


```
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.)


```
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:


```
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

```
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


```
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

Types

Types

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

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

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"

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

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`

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


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



```
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**."


```
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.

```
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”]
```



```
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


```
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

Initial State

Initial State

Next state formula

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


Next state formula

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

- Let’s skip a bunch and scroll down to `TCNext`, the “next state formula”

Next state formula

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

- Let’s skip a bunch and scroll down to `TCNext`, the “next state formula”
- This says:

Next state formula

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

- Let's skip a bunch and scroll down to `TCNext`, the "next state formula"
- This says:
  - "There exists an **`r`** in **`RM`** such that either **`Prepare(r)`** is TRUE or **`Decide(r)`** is TRUE"

Next state formula

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

- Let's skip a bunch and scroll down to `TCNext`, the "next state formula"
- This says:
  - "There exists an **`r`** in **`RM`** such that either **`Prepare(r)`** is TRUE or **`Decide(r)`** is TRUE"
- Remember the fundamental concept of TLA+

Next state formula

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

- Let's skip a bunch and scroll down to `TCNext`, the "next state formula"
- This says:
  - "There exists an **`r`** in **`RM`** such that either **`Prepare(r)`** is TRUE or **`Decide(r)`** is TRUE"
- Remember the fundamental concept of TLA+
  - Every single behaviour in the infinite universe of possibilities is generated, our formulas are used to specify a subset of behaviours

Next state formula

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

- Let's skip a bunch and scroll down to `TCNext`, the "next state formula"
- This says:
  - "There exists an **`r`** in **`RM`** such that either **`Prepare(r)`** is TRUE or **`Decide(r)`** is TRUE"
- Remember the fundamental concept of TLA+
  - Every single behaviour in the infinite universe of possibilities is generated, our formulas are used to specify a subset of behaviours
  - At every state, an infinite number of next states are theoretically possible. Our formula identifies which next states are **actually** possible, by applying the infinite set of next states to the formula and seeing which ones come out as TRUE

Next state formula

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


Next state formula

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

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

Next state formula

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

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

```
Prepare(r) == /\ rmState[r] = "working"
              /\ rmState' = [rmState EXCEPT ![r] = "prepared"]
```


Next state formula

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

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

```
Prepare(r) == /\ rmState[r] = "working"
              /\ rmState' = [rmState EXCEPT ![r] = "prepared"]
```

- This says:

Next state formula

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

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

```
Prepare(r) == /\ rmState[r] = "working"
              /\ rmState' = [rmState EXCEPT ![r] = "prepared"]
```

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

Next state formula

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

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

```
Prepare(r) == /\ rmState[r] = "working"
              /\ rmState' = [rmState EXCEPT ![r] = "prepared"]
```

- This says:
  - This formula is true for a pair of states if in the **current** state, resource manager **r** is "working"
  - And if in the **next** state, the state of resource manager **r** is "prepared", and all the other resource managers have the same state as they do right now

Next state formula

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

Prepare(r) == /\ rmState[r] = "working"
              /\ rmState' = [rmState EXCEPT ![r] = "prepared"]
```

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

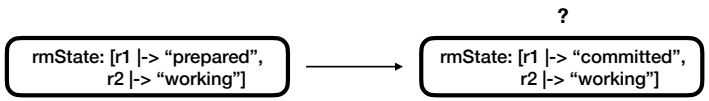
Next state formula

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

Prepare(r) == /\ rmState[r] = "working"
              /\ rmState' = [rmState EXCEPT ![r] = "prepared"]
```

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

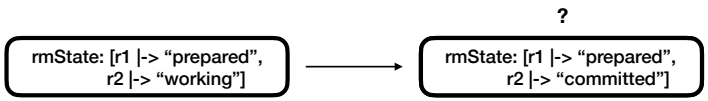
Next state formula

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

Prepare(r) == /\ rmState[r] = "working"
              /\ rmState' = [rmState EXCEPT ![r] = "prepared"]
```



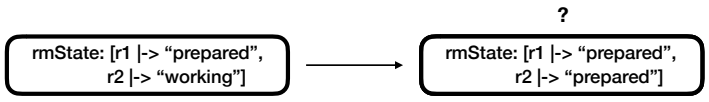
Next state formula

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

Prepare(r) == /\ rmState[r] = "working"
              /\ rmState' = [rmState EXCEPT ![r] = "prepared"]
```



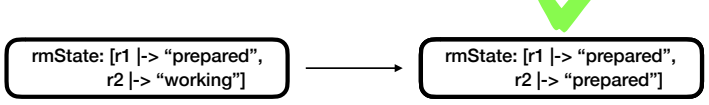
Next state formula

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

Prepare(r) == /\ rmState[r] = "working"
              /\ rmState' = [rmState EXCEPT ![r] = "prepared"]
```



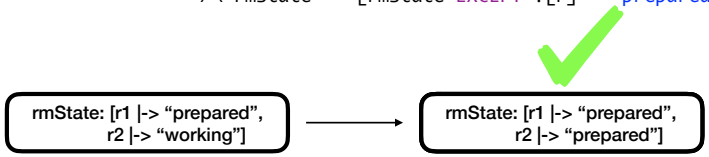
Next state formula

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

Prepare(r) == /\ rmState[r] = "working"
              /\ rmState' = [rmState EXCEPT ![r] = "prepared"]
```



Next state formula

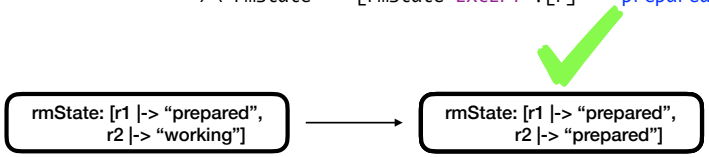
```
TCNext == \E r \in RM : Prepare(r) \/ Decide(r)

Prepare(r) == /\ rmState[r] = "working"
              /\ rmState' = [rmState EXCEPT ![r] = "prepared"]
```



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

Next state formula

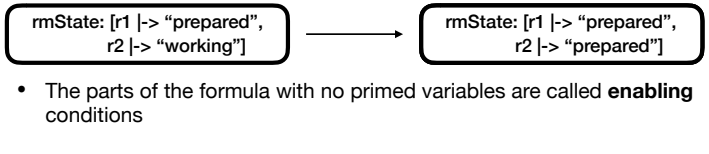
```
TCNext == \E r \in RM : Prepare(r) \/ Decide(r)

Prepare(r) == /\ rmState[r] = "working"
              /\ rmState' = [rmState EXCEPT ![r] = "prepared"]
```



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

Next state formula

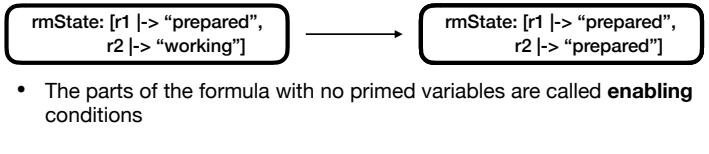
```
TCNext == \E r \in RM : Prepare(r) \/ Decide(r)

Prepare(r) == /\ rmState[r] = "working"
              /\ rmState' = [rmState 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

Next state formula

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

Prepare(r) == /\ rmState[r] = "working"
              /\ rmState' = [rmState EXCEPT ![r] = "prepared"]
```



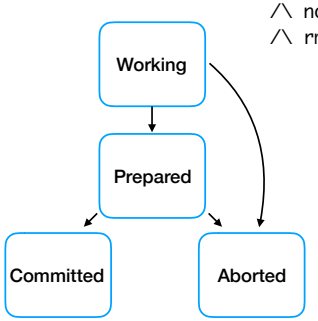
- The parts of the formula with no primed variables are called **enabling** conditions
- They describe what the **current state** must look like to be able to use this action
- Any formula with **primed variables** is called an **action**

Next state formula

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

Decide(r)  == \/ /\ rmState[r] = "prepared"
                 /\ canCommit
                 /\ rmState' = [rmState EXCEPT ![r] = "committed"]
              \/ /\ rmState[r] \in {"working", "prepared"}
                 /\ notCommitted
                 /\ rmState' = [rmState EXCEPT ![r] = "aborted"]
```

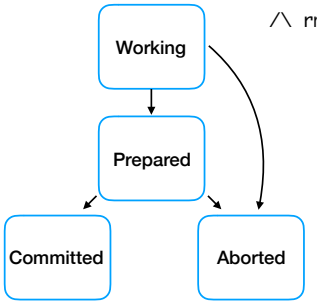

Next state formula

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

Decide(r)  == \/ /\ rmState[r] = "prepared"
                 /\ canCommit
                 /\ rmState' = [rmState EXCEPT ![r] = "committed"]
              \/ /\ rmState[r] \in {"working", "prepared"}
                 /\ notCommitted
                 /\ rmState' = [rmState EXCEPT ![r] = "aborted"]
```



Next state formula

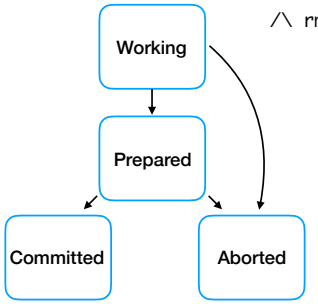
```
TCNext == \E r \in RM : Prepare(r) \/ Decide(r)

Decide(r)  == \/ /\ rmState[r] = "prepared"
                 /\ canCommit
                 /\ rmState' = [rmState EXCEPT ![r] = "committed"]
              \/ /\ rmState[r] \in {"working", "prepared"}
                 /\ notCommitted
                 /\ rmState' = [rmState EXCEPT ![r] = "aborted"]
```



- This formula is true when either

Next state formula

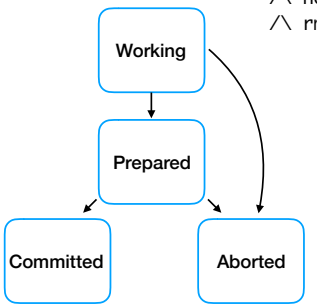
```
TCNext == \E r \in RM : Prepare(r) \/ Decide(r)

Decide(r)  == \/ /\ rmState[r] = "prepared"
                 /\ canCommit
                 /\ rmState' = [rmState EXCEPT ![r] = "committed"]
              \/ /\ rmState[r] \in {"working", "prepared"}
                 /\ notCommitted
                 /\ rmState' = [rmState EXCEPT ![r] = "aborted"]
```



- This formula is true when either

Next state formula

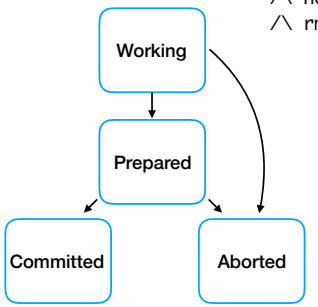
```
TCNext == \E r \in RM : Prepare(r) \/ Decide(r)

Decide(r)  == \/ /\ rmState[r] = "prepared"
              /\ canCommit
              /\ rmState' = [rmState EXCEPT ![r] = "committed"]
           \/ /\ rmState[r] \in {"working", "prepared"}
              /\ notCommitted
              /\ rmState' = [rmState EXCEPT ![r] = "aborted"]
```



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

Next state formula

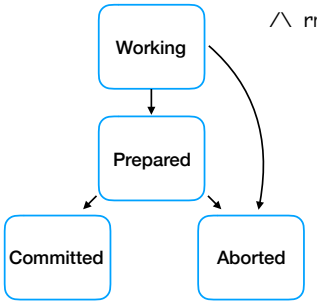
```
TCNext == \E r \in RM : Prepare(r) \/ Decide(r)

Decide(r)  == \/ /\ rmState[r] = "prepared"
              /\ canCommit
              /\ rmState' = [rmState EXCEPT ![r] = "committed"]
           \/ /\ rmState[r] \in {"working", "prepared"}
              /\ notCommitted
              /\ rmState' = [rmState EXCEPT ![r] = "aborted"]
```



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

Next state formula

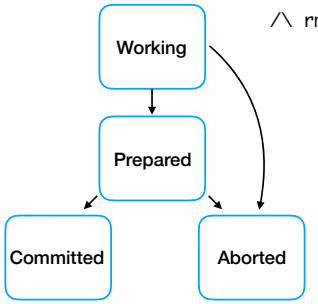
```
TCNext == \E r \in RM : Prepare(r) \/ Decide(r)

Decide(r)  == \/ /\ rmState[r] = "prepared"
              /\ canCommit
              /\ rmState' = [rmState EXCEPT ![r] = "committed"]
           \/ /\ rmState[r] \in {"working", "prepared"}
              /\ notCommitted
              /\ rmState' = [rmState EXCEPT ![r] = "aborted"]
```



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

Next state formula

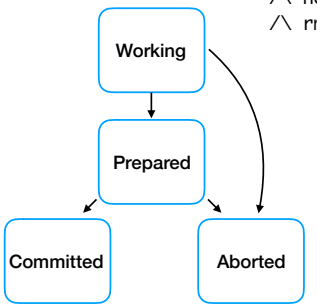
```
TCNext == \E r \in RM : Prepare(r) \/ Decide(r)

Decide(r)  == \/ /\ rmState[r] = "prepared"
              /\ canCommit
              /\ rmState' = [rmState EXCEPT ![r] = "committed"]
           \/ /\ rmState[r] \in {"working", "prepared"}
              /\ notCommitted
              /\ rmState' = [rmState EXCEPT ![r] = "aborted"]
```



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

Next state formula

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

Decide(r)  == \/ /\ rmState[r] = "prepared"
              /\ canCommit
              /\ rmState' = [rmState EXCEPT ![r] = "committed"]
           \/ /\ rmState[r] \in {"working", "prepared"}
              /\ notCommitted
              /\ rmState' = [rmState EXCEPT ![r] = "aborted"]
```

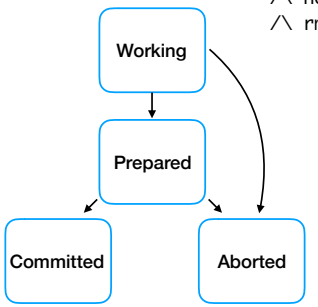


- This formula is true when either
  - RM **r** is in state “prepared”
  - And **canCommit** is true
  - And that RM **r** is “committed” in the next state

248-9

Next state formula

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

Decide(r)  == \/ /\ rmState[r] = "prepared"
              /\ canCommit
              /\ rmState' = [rmState EXCEPT ![r] = "committed"]
           \/ /\ rmState[r] \in {"working", "prepared"}
              /\ notCommitted
              /\ rmState' = [rmState EXCEPT ![r] = "aborted"]
```

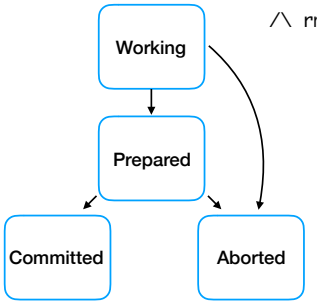


- This formula is true when either
  - RM **r** is in state “prepared”
  - And **canCommit** is true
  - And that RM **r** is “committed” in the next state

248-10

Next state formula

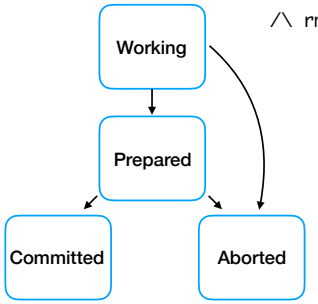
```
TCNext == \E r \in RM : Prepare(r) \/ Decide(r)

Decide(r)  == \/ /\ rmState[r] = "prepared"
              /\ canCommit
              /\ rmState' = [rmState EXCEPT ![r] = "committed"]
           \/ /\ rmState[r] \in {"working", "prepared"}
              /\ notCommitted
              /\ rmState' = [rmState EXCEPT ![r] = "aborted"]
```



249-1

Next state formula

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

Decide(r)  == \/ /\ rmState[r] = "prepared"
              /\ canCommit
              /\ rmState' = [rmState EXCEPT ![r] = "committed"]
           \/ /\ rmState[r] \in {"working", "prepared"}
              /\ notCommitted
              /\ rmState' = [rmState EXCEPT ![r] = "aborted"]
```

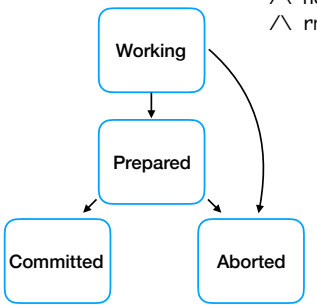


- or when

249-2

Next state formula

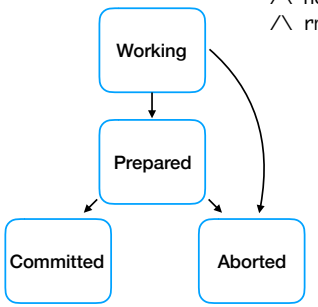
```
TCNext == \E r \in RM : Prepare(r) \/ Decide(r)

Decide(r)  == \/ /\ rmState[r] = "prepared"
              /\ canCommit
              /\ rmState' = [rmState EXCEPT ![r] = "committed"]
           \/ /\ rmState[r] \in {"working", "prepared"}
              /\ notCommitted
              /\ rmState' = [rmState EXCEPT ![r] = "aborted"]
```



- or when

Next state formula

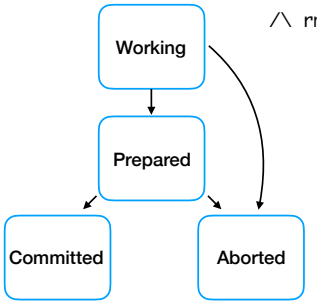
```
TCNext == \E r \in RM : Prepare(r) \/ Decide(r)

Decide(r)  == \/ /\ rmState[r] = "prepared"
              /\ canCommit
              /\ rmState' = [rmState EXCEPT ![r] = "committed"]
           \/ /\ rmState[r] \in {"working", "prepared"}
              /\ notCommitted
              /\ rmState' = [rmState EXCEPT ![r] = "aborted"]
```



- or when
  - RM **r** is currently “working” or “prepared”

Next state formula

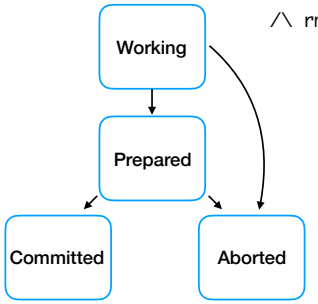
```
TCNext == \E r \in RM : Prepare(r) \/ Decide(r)

Decide(r)  == \/ /\ rmState[r] = "prepared"
              /\ canCommit
              /\ rmState' = [rmState EXCEPT ![r] = "committed"]
           \/ /\ rmState[r] \in {"working", "prepared"}
              /\ notCommitted
              /\ rmState' = [rmState EXCEPT ![r] = "aborted"]
```



- or when
  - RM **r** is currently “working” or “prepared”

Next state formula

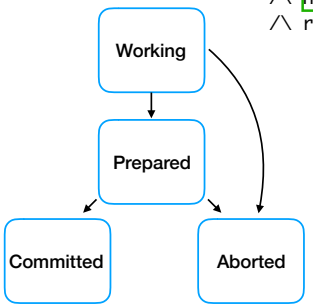
```
TCNext == \E r \in RM : Prepare(r) \/ Decide(r)

Decide(r)  == \/ /\ rmState[r] = "prepared"
              /\ canCommit
              /\ rmState' = [rmState EXCEPT ![r] = "committed"]
           \/ /\ rmState[r] \in {"working", "prepared"}
              /\ notCommitted
              /\ rmState' = [rmState EXCEPT ![r] = "aborted"]
```



- or when
  - RM **r** is currently “working” or “prepared”
  - And **notCommitted** is true

Next state formula

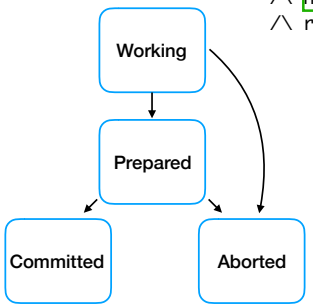
```
TCNext == \E r \in RM : Prepare(r) \/ Decide(r)

Decide(r)  == \/ /\ rmState[r] = "prepared"
              /\ canCommit
              /\ rmState' = [rmState EXCEPT ![r] = "committed"]
           \/ /\ rmState[r] \in {"working", "prepared"}
              /\ notCommitted
              /\ rmState' = [rmState EXCEPT ![r] = "aborted"]
```



- or when
  - RM **r** is currently “working” or “prepared”
  - And **notCommitted** is true

Next state formula

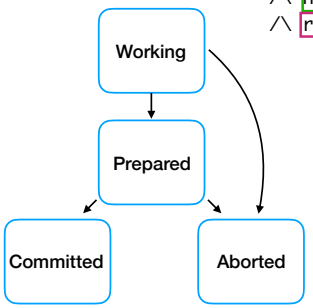
```
TCNext == \E r \in RM : Prepare(r) \/ Decide(r)

Decide(r)  == \/ /\ rmState[r] = "prepared"
              /\ canCommit
              /\ rmState' = [rmState EXCEPT ![r] = "committed"]
           \/ /\ rmState[r] \in {"working", "prepared"}
              /\ notCommitted
              /\ rmState' = [rmState EXCEPT ![r] = "aborted"]
```



- or when
  - RM **r** is currently “working” or “prepared”
  - And **notCommitted** is true
  - And in the next state RM **r** is “aborted”

Next state formula

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

Decide(r)  == \/ /\ rmState[r] = "prepared"
              /\ canCommit
              /\ rmState' = [rmState EXCEPT ![r] = "committed"]
           \/ /\ rmState[r] \in {"working", "prepared"}
              /\ notCommitted
              /\ rmState' = [rmState EXCEPT ![r] = "aborted"]
```



- or when
  - RM **r** is currently “working” or “prepared”
  - And **notCommitted** is true
  - And in the next state RM **r** is “aborted”

canCommit

```
canCommit == \A r \in RM : rmState[r] \in {"prepared", "committed"}
```

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))
```


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))
```


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))
```


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))
```

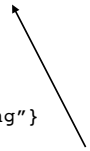
notCommitted

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

"Not equal." You can also use **/=**

notCommitted

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



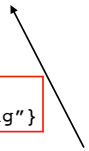
Python Equivalents

```
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))
```

"Not equal." You can also use **/=**

notCommitted

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



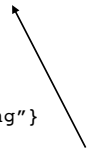
Python Equivalents

```
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))
```

"Not equal." You can also use **/=**

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))
```

"Not equal." You can also use **/=**

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))
```


Next state formula

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


Next state formula

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

Next state formula

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

TCConsistent

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


TCConsistent

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

- For every possible pair of resource managers r1 and r2

TCConsistent

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

- For every possible pair of resource managers r1 and r2
- It should not be the case that

TCConsistent

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

- For every possible pair of resource managers r1 and r2
- It should not be the case that
- rmState[r1] = “aborted”

TCConsistent

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

TCConsistent

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


TCConsistent

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

TCommit Recap

TCommit Recap

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

TCommit Recap

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

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

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

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

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)

Two Phase Commit

Two Phase Commit

- A simple solution of the Transaction Commit problem

Two Phase Commit

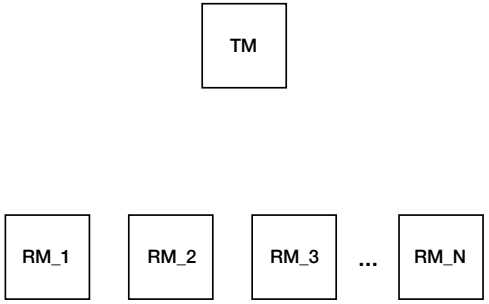
- A simple solution of the Transaction Commit problem
- Introduces a “Transaction Manager” (TM) to coordinate with the RMs

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

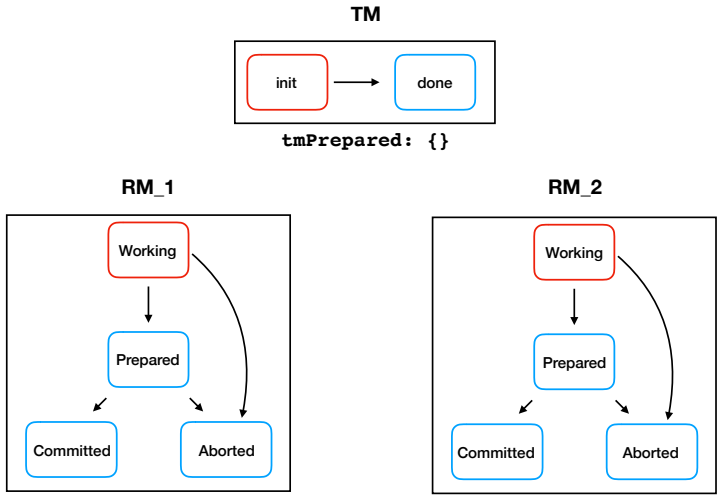
Transaction Manager

Transaction Manager



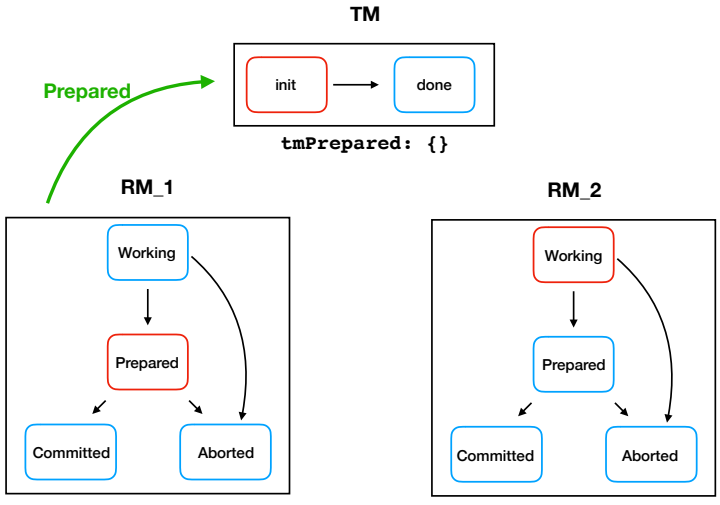
257-2

Transaction Manager



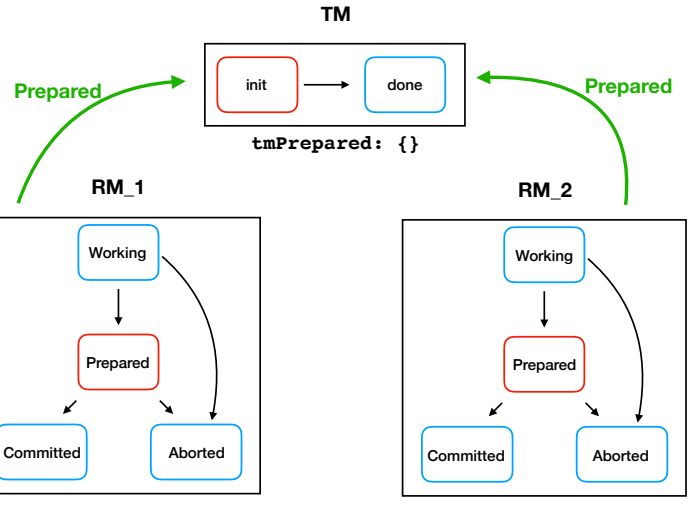
258

Transaction Manager



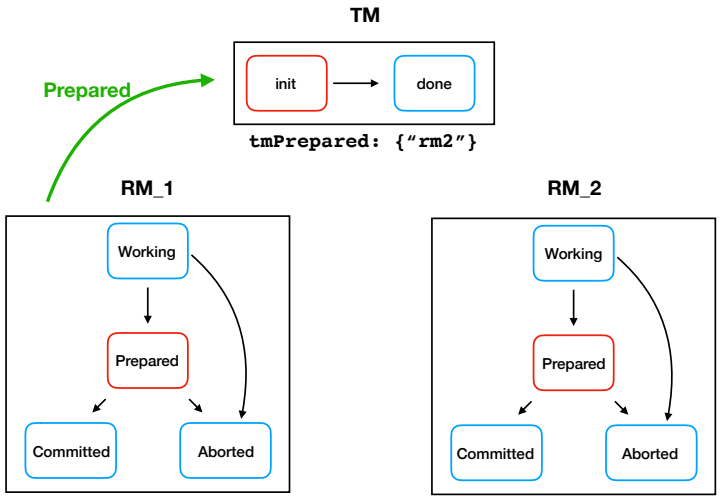
259

Transaction Manager



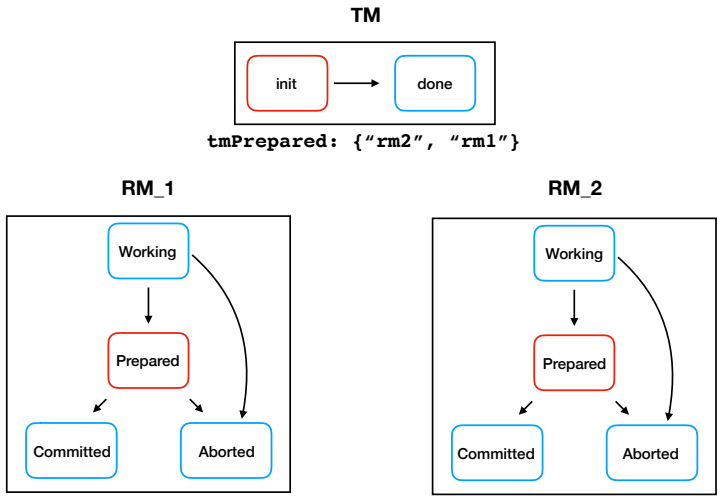
260

Transaction Manager



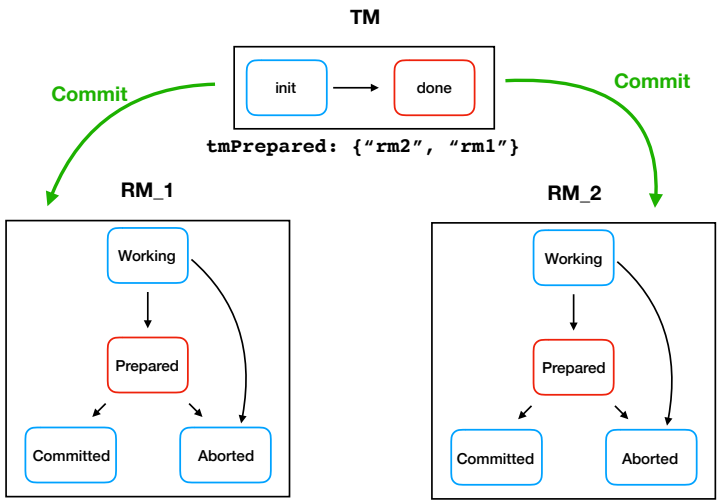
261

Transaction Manager



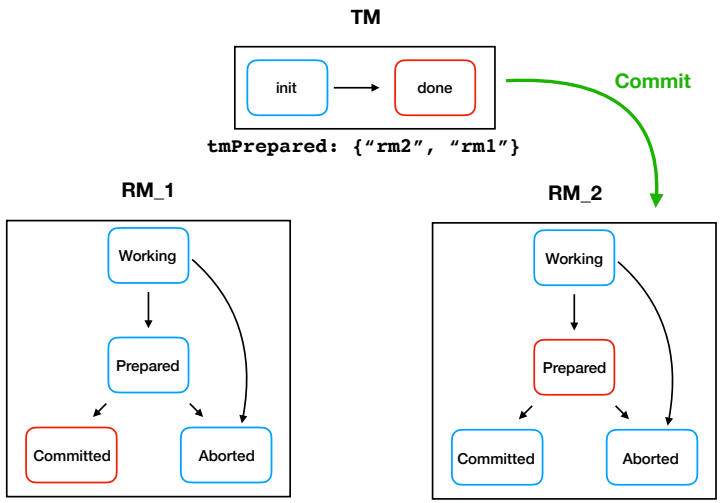
262

Transaction Manager



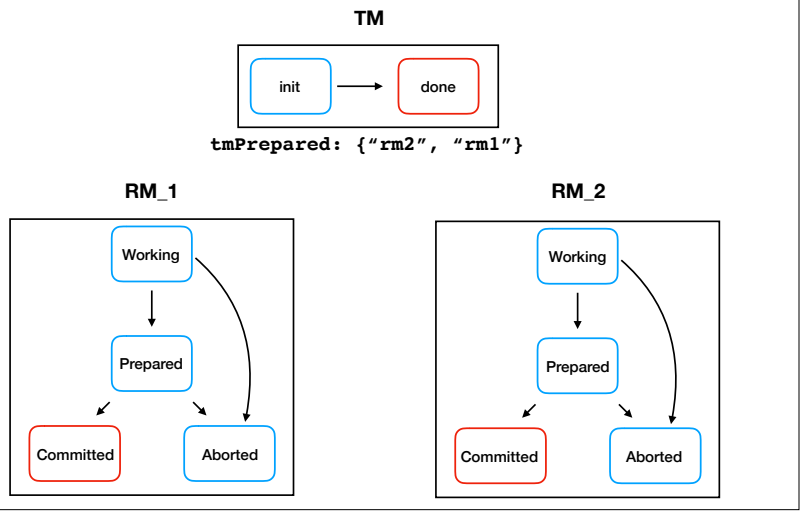
263

Transaction Manager

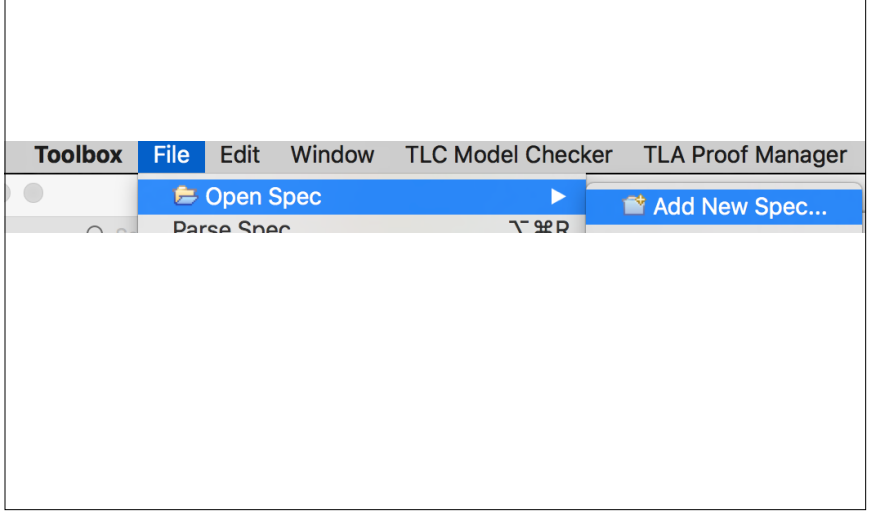


264

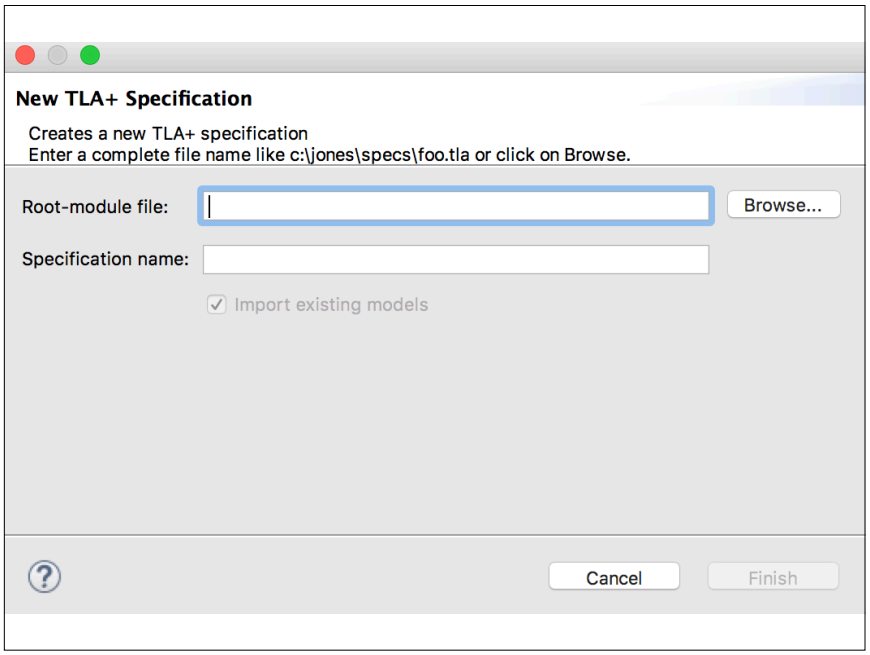
Transaction Manager



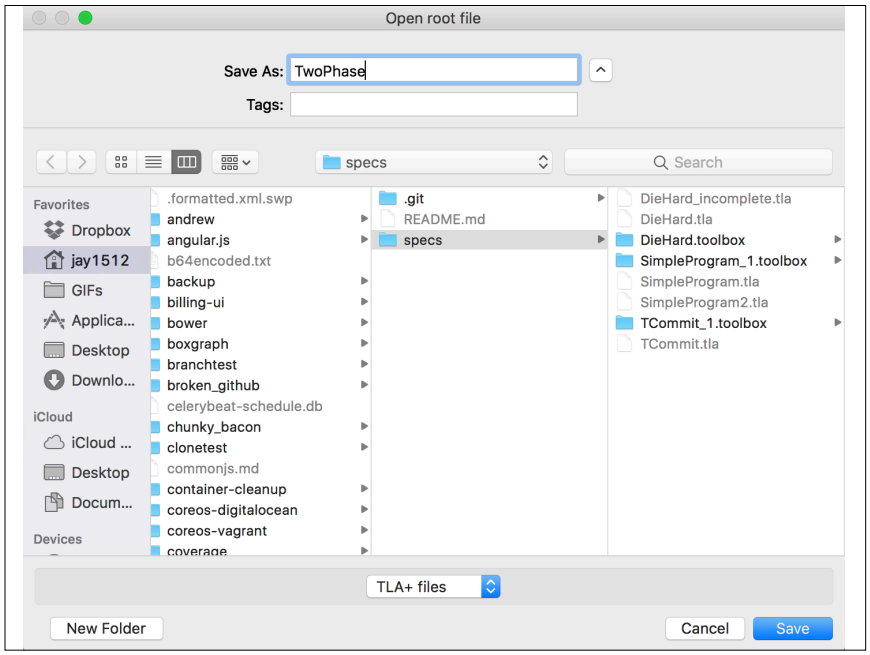
265



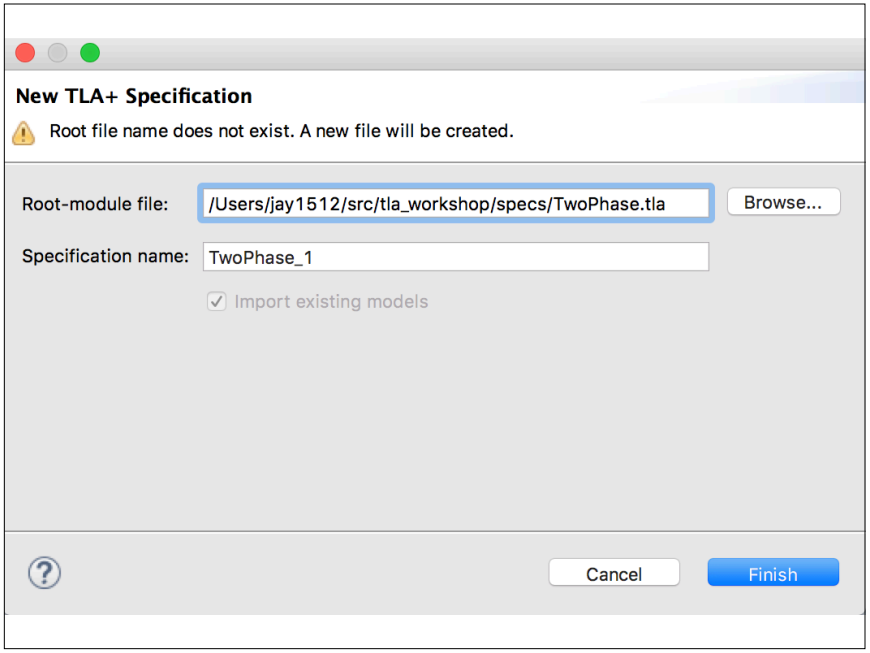
266



267



268



269

```
---------------------------- MODULE TwoPhase ----------------------------
(***************************************************************************)
(* This specification is discussed in "Two-Phase Commit", Lecture 6 of the *)
(* TLA+ Video Course.  It describes the Two-Phase Commit protocol, in      *)
(* 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      *)
  (* such 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}
```


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”}
```


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
```


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”}
```

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

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}
```


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}
```


Sets of records

Sets of records

- We also often need to easily create sets of records

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

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"}]
```


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"] }
```

Sets of records

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

- This is equivalent to the following Python:

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})
```

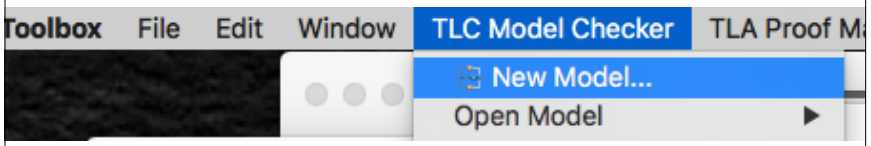

Sets of records

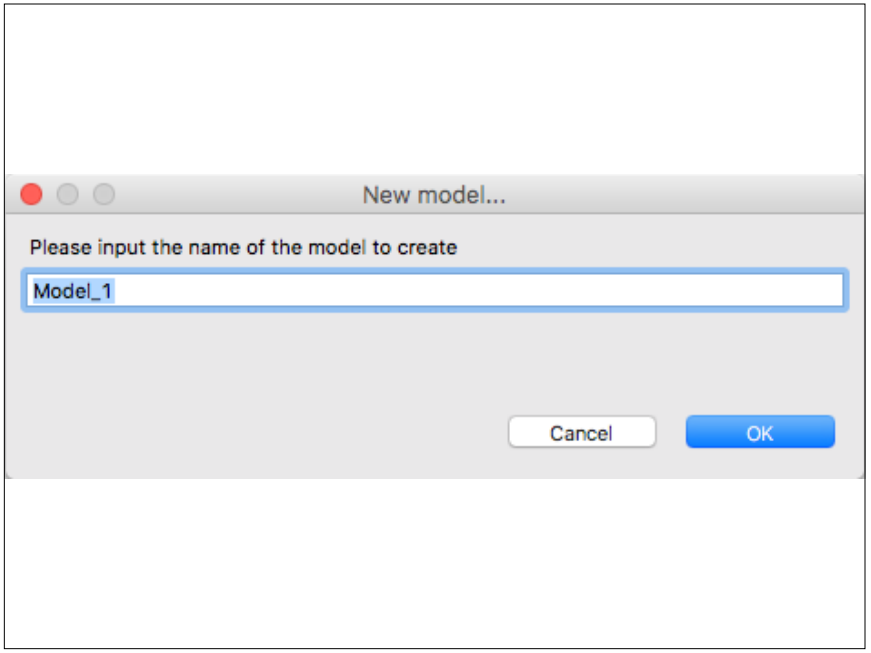
```
[nodes: {1,2}, edges: {"a","b","c"}]
```

- This is equivalent to the following Python:

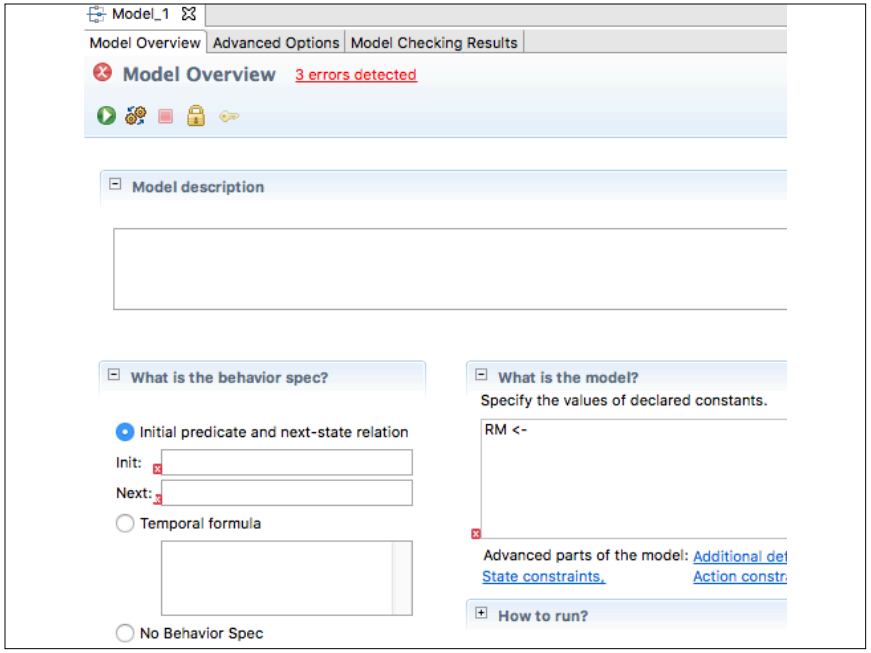
```
set_of_records = set()
for node in [1,2]:
    for edge in [“a”,”b”,”c”]:
        set_of_records.add({“nodes”: node, “edges”, edge})
```

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

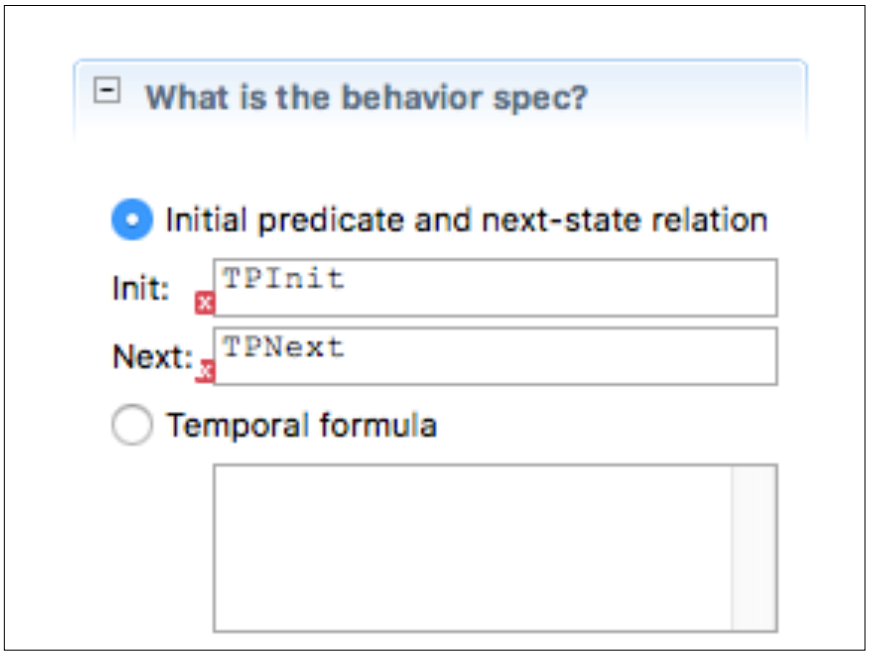





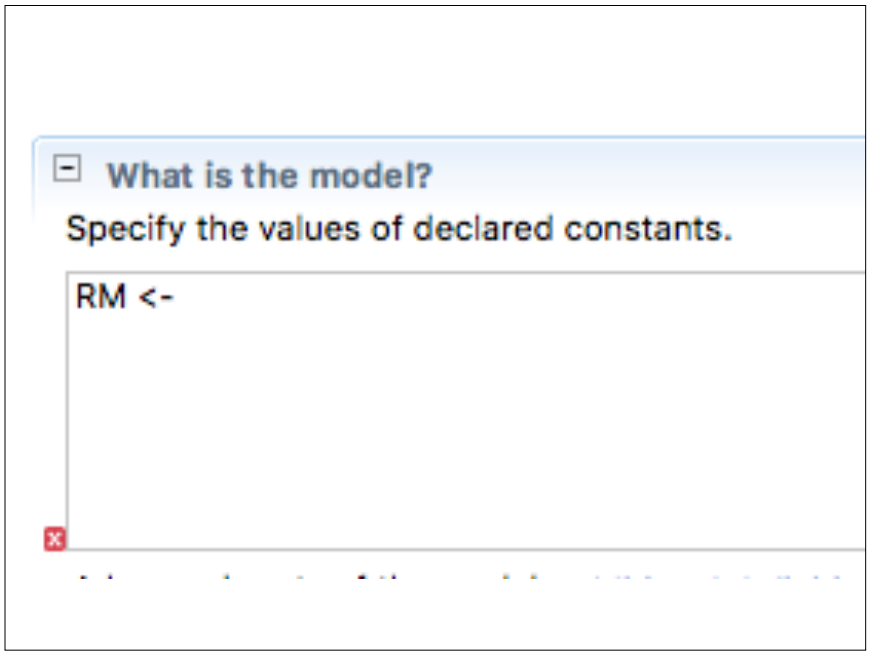
275



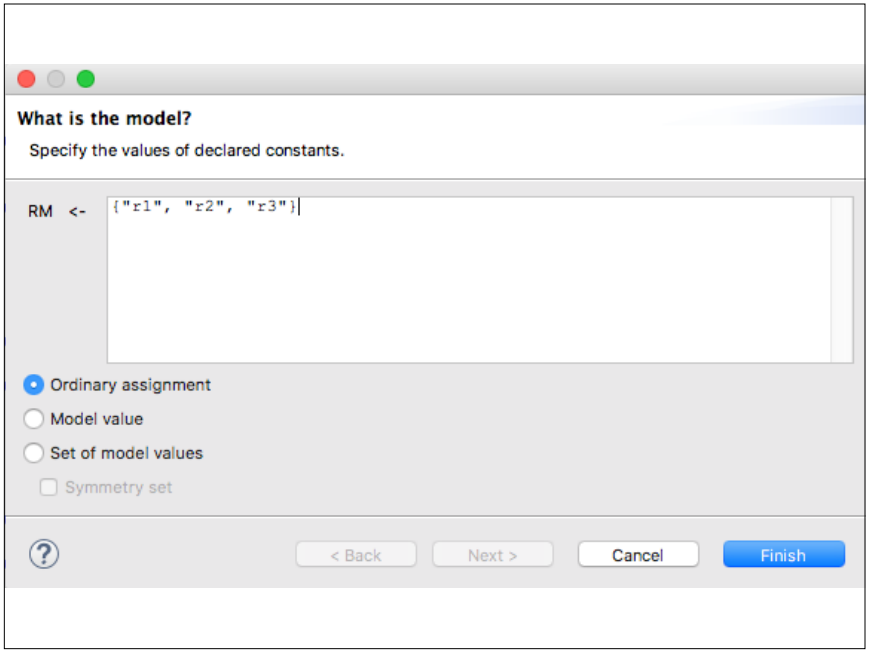
276



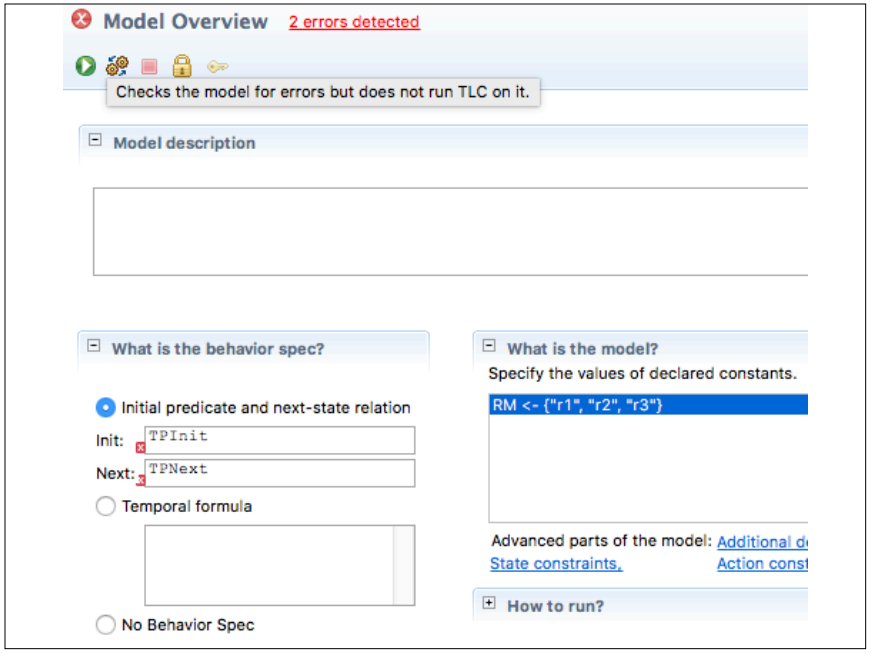
277



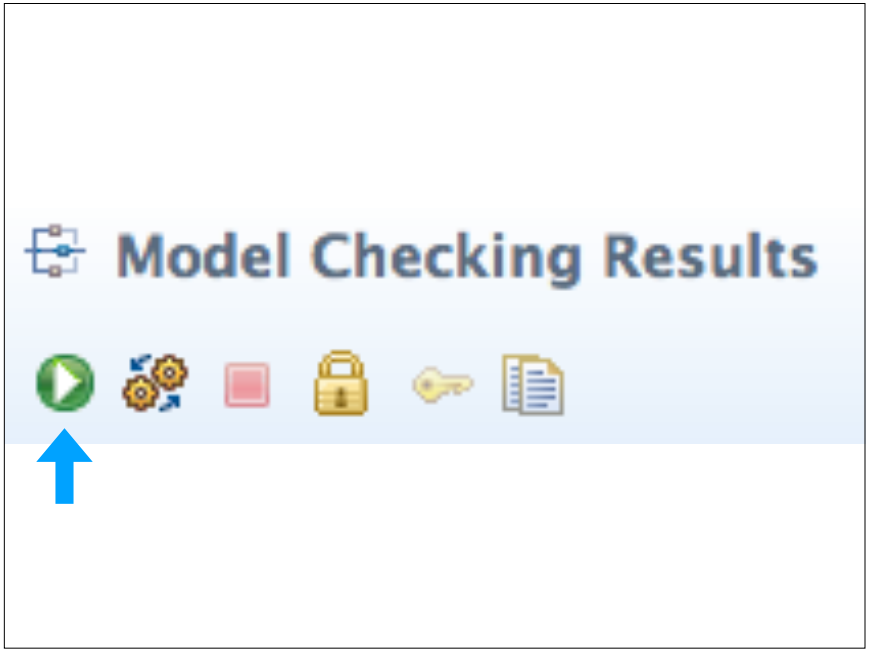
278



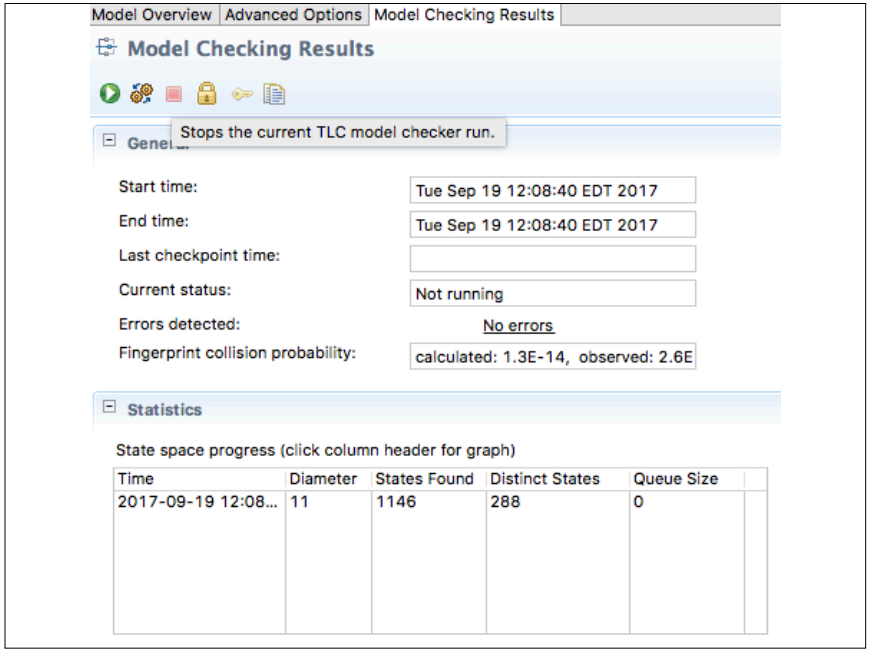
279



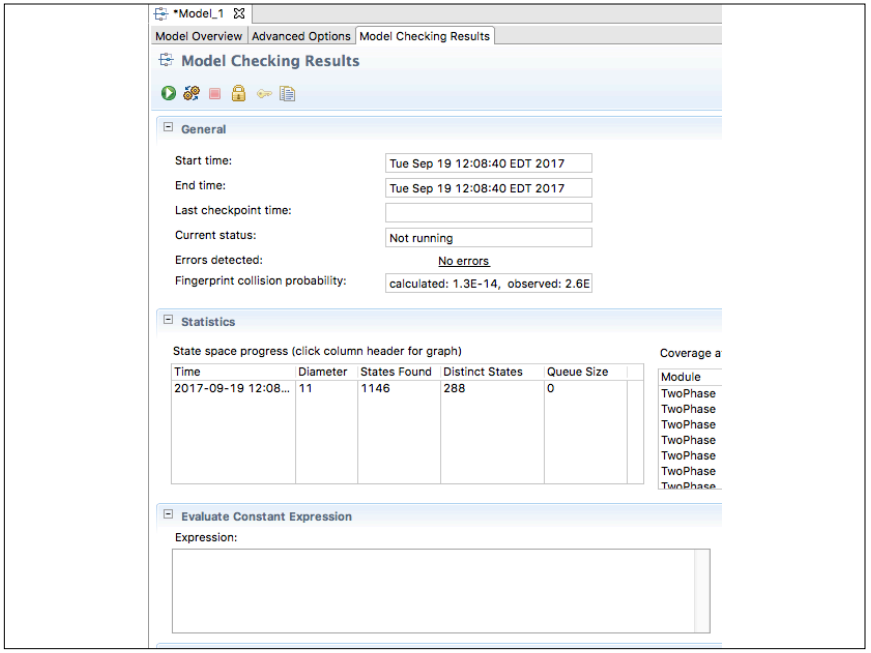
280



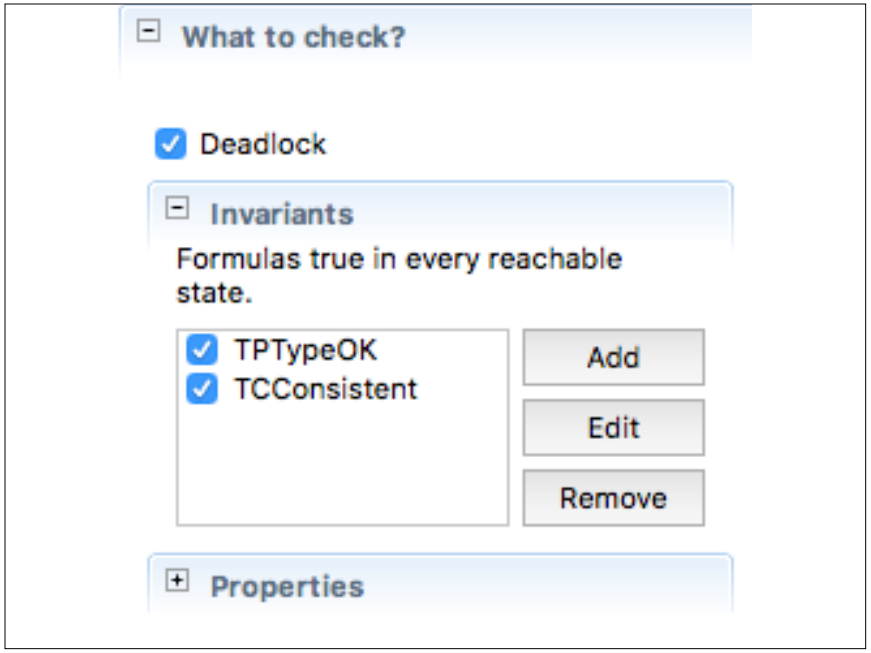
281



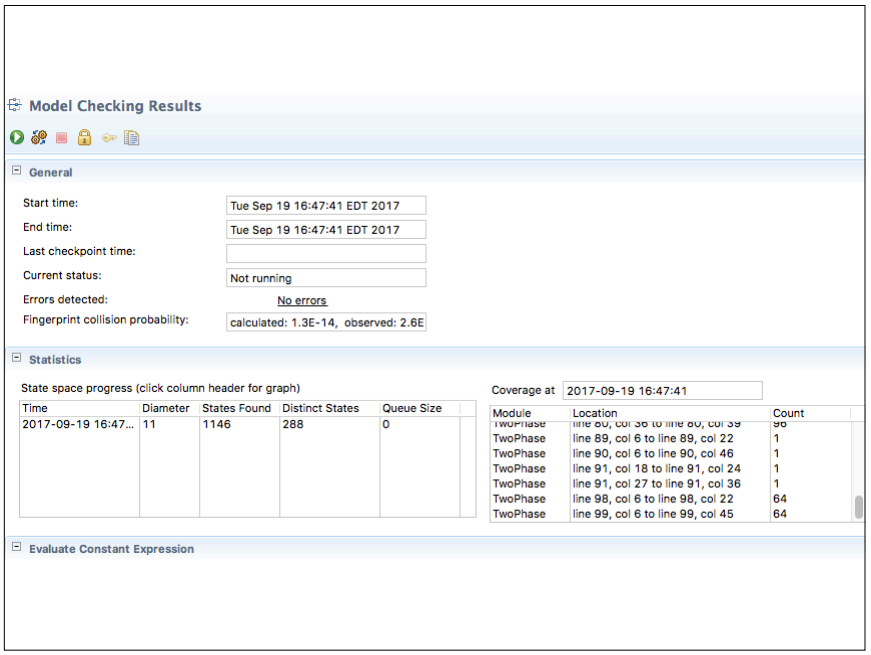
282



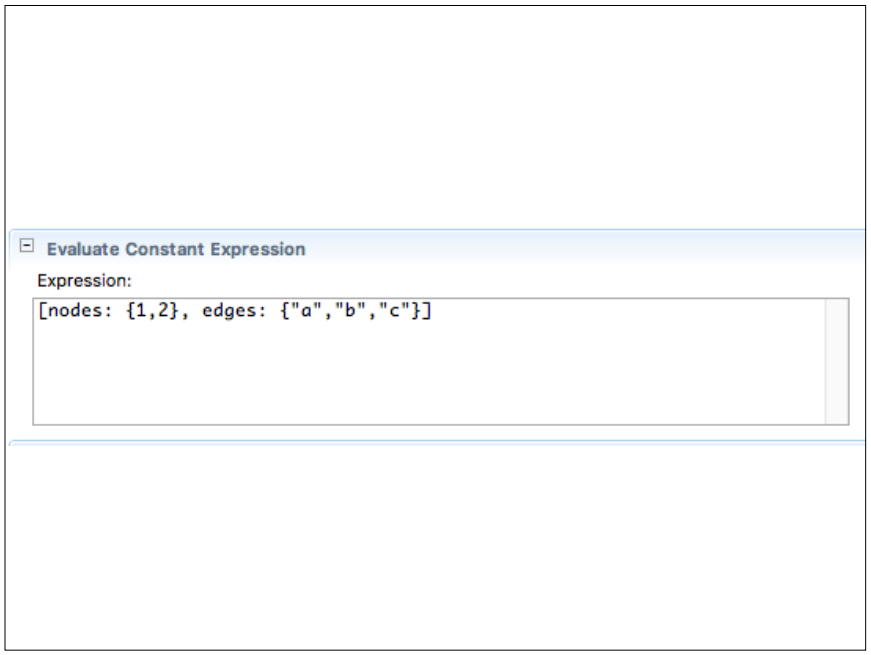
283



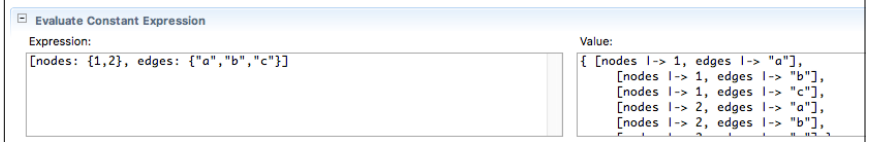
284



285



286



TwoPhase

```
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
```


TwoPhase

```
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
```

- **RM** and **rmState** are the same as before

TwoPhase

```
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
```

- **RM** and **rmState** are the same as before
- **tmState** is either “init” or “done”

TwoPhase

```
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
```

- **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”})

TwoPhase

```
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
```

- **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

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"}]
```


TwoPhase

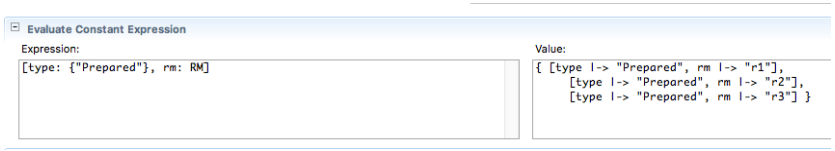
```
Messages ==
  (***************************************************************************)
  (* The set of all possible messages.  Messages of type "Prepared" are    *)
  (* sent from the RM indicated by the message's rm field to the TM.       *)
  (* Messages of type "Commit" and "Abort" are broadcast 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"}]
```

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"}]
```


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"}]
```



TwoPhase

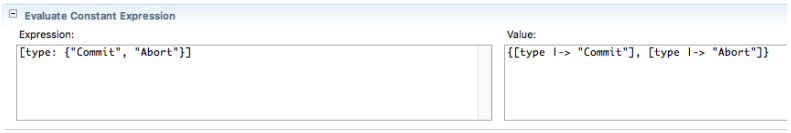
```
Messages ==
  (***************************************************************************)
  (* The set of all possible messages.  Messages of type "Prepared" are    *)
  (* sent from the RM indicated by the message's rm field to the TM.       *)
  (* Messages of type "Commit" and "Abort" are broadcast 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"}]
```


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"}]
```

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"}]
```



290-3

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

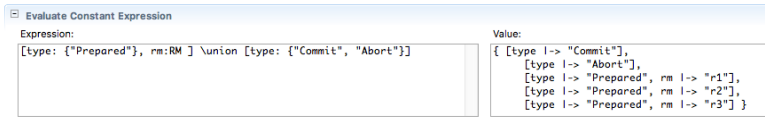
TwoPhase

```
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"}]
```

291-2

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"}]
```



291-3

TPTypeOK

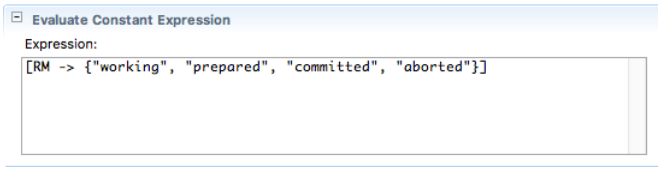
```
TPTypeOK ==
  (***************************************************************************)
  (* The type-correctness invariant                                          *)
  (***************************************************************************)
  /\ rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
  /\ tmState \in {"init", "done"}
  /\ tmPrepared \subseteq RM
  /\ msgs \subseteq Messages
```


TPTypeOK

```
TPTypeOK ==
  (***************************************************************************)
  (* The type-correctness invariant                                          *)
  (***************************************************************************)
  /\ rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
  /\ tmState \in {"init", "done"}
  /\ tmPrepared \subseteq RM
  /\ msgs \subseteq Messages
```


TPTypeOK

```
TPTypeOK ==
  (***************************************************************************)
  (* The type-correctness invariant                                          *)
  (***************************************************************************)
  /\ rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
  /\ tmState \in {"init", "done"}
  /\ tmPrepared \subseteq RM
  /\ msgs \subseteq Messages
```



TPTypeOK

```
TPTypeOK ==
  (***************************************************************************)
  (* The type-correctness invariant                                          *)
  (***************************************************************************)
  /\ rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
  /\ tmState \in {"init", "done"}
  /\ tmPrepared \subseteq RM
  /\ msgs \subseteq Messages
```

```
{ [r1 |-> "working", r2 |-> "working", r3 |-> "working"],
    [r1 |-> "working", r2 |-> "working", r3 |-> "prepared"],
    [r1 |-> "working", r2 |-> "working", r3 |-> "committed"],
    [r1 |-> "working", r2 |-> "working", r3 |-> "aborted"],
    [r1 |-> "working", r2 |-> "prepared", r3 |-> "working"],
    [r1 |-> "working", r2 |-> "prepared", r3 |-> "prepared"],
    [r1 |-> "working", r2 |-> "prepared", r3 |-> "committed"],
    [r1 |-> "working", r2 |-> "prepared", r3 |-> "aborted"],
    [r1 |-> "working", r2 |-> "committed", r3 |-> "working"],
    [r1 |-> "working", r2 |-> "committed", r3 |-> "prepared"],
    [r1 |-> "working", r2 |-> "committed", r3 |-> "committed"],
    [r1 |-> "working", r2 |-> "committed", r3 |-> "aborted"],
    [r1 |-> "aborted", r2 |-> "prepared", r3 |-> "aborted"],
    [r1 |-> "aborted", r2 |-> "committed", r3 |-> "working"],
```

TPTypeOK

```
TPTypeOK ==
  (***************************************************************************)
  (* The type-correctness invariant                                          *)
  (***************************************************************************)
  /\ rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
  /\ tmState \in {"init", "done"}
  /\ tmPrepared \subseteq RM
  /\ msgs \subseteq Messages
```


TPTypeOK

```
TPTypeOK ==
  (***************************************************************************)
  (* The type-correctness invariant                                          *)
  (***************************************************************************)
  /\ rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
  /\ tmState \in {"init", "done"}
  /\ tmPrepared \subseteq RM
  /\ msgs \subseteq Messages
```


TPTypeOK

```
TPTypeOK ==
  (***************************************************************************)
  (* The type-correctness invariant                                          *)
  (***************************************************************************)
  /\ rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
  /\ tmState \in {"init", "done"}
  /\ tmPrepared \subseteq RM
  /\ msgs \subseteq Messages
```

\subseteq means “subset or equal”

TPTypeOK

```
TPTypeOK ==
  (***************************************************************************)
  (* The type-correctness invariant                                          *)
  (***************************************************************************)
  /\ rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
  /\ tmState \in {"init", "done"}
  /\ tmPrepared \subseteq RM
  /\ msgs \subseteq Messages
```

\subseteq means “subset or equal”

msgs

msgs

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

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

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

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!

TPInit

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


TPInit

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


TPInit

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

Evaluate this expression. What do you get?

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"]
```


TPInit

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

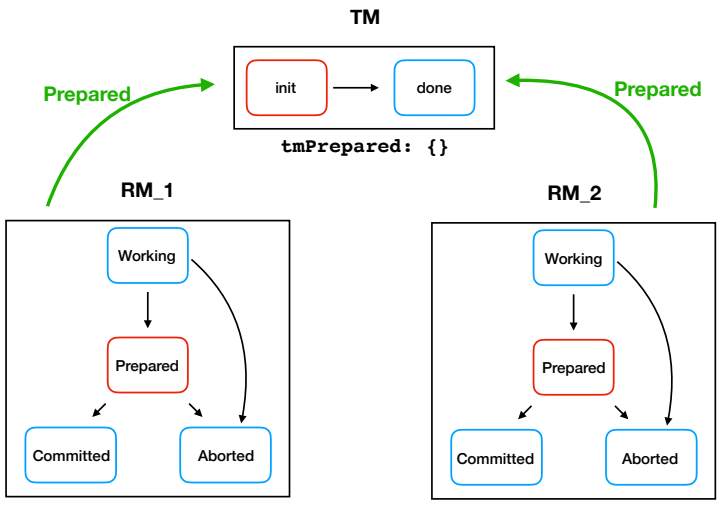
Evaluate this expression. What do you get?

Value:

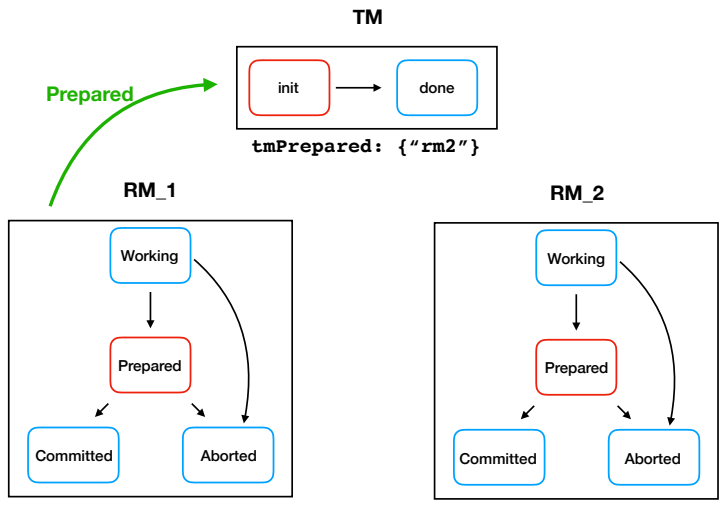
```
[r1 |-> "working", r2 |-> "working", r3 |-> "working"]
```

This is a single function, where the DOMAIN is made up of resource managers, and the value associated to each is “working”

TMRcvPrepared(r)



TMRcvPrepared(r)



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

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”

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?

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?

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?

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

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?

`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

TMRcvPrepared(r)

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


TMRcvPrepared(r)

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

TMRcvPrepared(r)

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

TMRcvPrepared(r)

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


TMRcvPrepared(r)

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


TMRcvPrepared(r)

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

This is new

TMRcvPrepared(r)

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

This is new

It says: “In the next state, **rmState** and **tmState** and **msgs** will be unchanged”

TMRcvPrepared(r)

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

```
/\ rmState’ = rmState
/\ tmState’ = tmState
/\ msgs’ = msgs
```


TMRcvPrepared(r)

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


TMRcvPrepared(r)

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

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

TMRcvPrepared(r)

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

TMRcvPrepared(r)

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

TMRcvPrepared(r)

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

TMRcvPrepared(r)

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

TMRcvPrepared(r)

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

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

TMRcvPrepared(r)

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

TMRcvPrepared(r)

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

RMPrepare(r)

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

RMPrepare(r)

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

- This action defines a RM **r** transitioning from “working” to “prepared”

RMPrepare(r)

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

RMPrepare(r)

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

RMPrepare(r)

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

RMPrepare(r)

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

TPNext

```
TPNext ==
  \/ TMCommit \/ TMAbort
  \/ \E r \in RM :
       TMRcvPrepared(r) \/ RMPrepare(r) \/ RMChooseToAbort(r)
         \/ RMRcvCommitMsg(r) \/ RMRcvAbortMsg(r)
```


TPNext

```
TPNext ==
  \/ TMCommit \/ TMAbort
  \/ \E r \in RM :
       TMRcvPrepared(r) \/ RMPrepare(r) \/ RMChooseToAbort(r)
         \/ RMRcvCommitMsg(r) \/ RMRcvAbortMsg(r)
```

- Our next state action. This determines all the possible next states

TPNext

```
TPNext ==
  \/ TMCommit \/ TMAbort
  \/ \E r \in RM :
       TMRcvPrepared(r) \/ RMPrepare(r) \/ RMChooseToAbort(r)
         \/ RMRcvCommitMsg(r) \/ RMRcvAbortMsg(r)
```

- Our next state action. This determines all the possible next states
- At any time, the TM can *potentially* commit or abort

TPNext

```
TPNext ==
  \/ TMCommit \/ TMAbort
  \/ \E r \in RM :
       TMRcvPrepared(r) \/ RMPrepare(r) \/ RMChooseToAbort(r)
         \/ RMRcvCommitMsg(r) \/ 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*

TPNext

```
TPNext ==
  \/ TMCommit \/ TMAbort
  \/ \E r \in RM :
       TMRcvPrepared(r) \/ RMPrepare(r) \/ RMChooseToAbort(r)
         \/ RMRcvCommitMsg(r) \/ 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**

TPNext

```
TPNext ==
  \/ TMCommit \/ TMAbort
  \/ \E r \in RM :
       TMRcvPrepared(r) \/ RMPrepare(r) \/ RMChooseToAbort(r)
         \/ RMRcvCommitMsg(r) \/ 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

TPNext

```
TPNext ==
  \/ TMCommit \/ TMAbort
  \/ \E r \in RM :
       TMRcvPrepared(r) \/ RMPrepare(r) \/ RMChooseToAbort(r)
         \/ RMRcvCommitMsg(r) \/ 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

TPNext

```
TPNext ==
  \/ TMCommit \/ TMAbort
  \/ \E r \in RM :
       TMRcvPrepared(r) \/ RMPrepare(r) \/ RMChooseToAbort(r)
         \/ RMRcvCommitMsg(r) \/ 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

TPNext

```
TPNext ==
  \/ TMCommit \/ TMAbort
  \/ \E r \in RM :
       TMRcvPrepared(r) \/ RMPrepare(r) \/ RMChooseToAbort(r)
         \/ RMRcvCommitMsg(r) \/ 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

INSTANCE TCommit

INSTANCE TCommit

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

INSTANCE TCommit

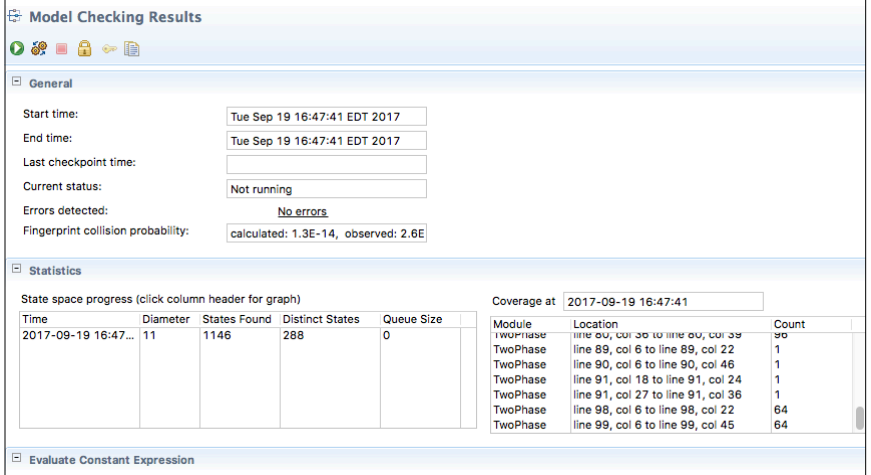
- This "imports" all variable declarations, constants, and definitions from the **TCommit** module
- This is how we got access to the **TCConsistent** invariant

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

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



The End?

- This brings us to the end of Video 6 in Lamport's series
  - TLA Video Series: 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!