CS6480: Model Checking and TLC

Robbert van Renesse

Cornell University

What is formal verification?

- Does software correctly implement a specification?
- Does software have desired properties (safety, liveness, other)?
- Is a particular optimization correct (equivalence, bi-simulation)?

Formal tools are used to check the above

Three parts to formal verification

- Soundness
  - If the formal verifier reports no bug, then the system does not fail
- Completeness
  - If the formal verifier reports a bug, then the system can fail
- Termination
  - The formal verifier terminates

Two types of formal verifiers

- Provers
  - Reason based on axioms and rules of inference
  - Automatic proof checking
    - but proof creation can be at least partly manual
  - *Difficult*
- Model checkers
  - Manually create a model
  - Automatically explore the state space of the model
  - *Relatively simple*

Recall TLA+

- A *state* is an assignment of values to all variables
- A *step* is a pair of states
- A *stuttering step* wrt some variable leaves the variable unchanged
- An *action* is a predicate over a pair of states
  - If x is a variable in the old state, then x' is the same variable in the new state
- A *behavior* is an infinite sequence of states (with an initial state)
- A *specification* characterizes the initial state and actions

Some more terms

- A *state function* is a first-order logic expression
- A *state predicate* is a Boolean state function
- A *temporal formula* is an assertion about behaviors
- *A theorem* of a specification is a temporal formula that holds over every behavior of the specification
- If S is a specification and I is a predicate and $S \Rightarrow \Box I$ is a theorem then we call I an *invariant* of S.

Temporal Formula

Based on Chapter 8 of Specifying Systems

- A *temporal formula* F assigns a Boolean value to a behavior σ
- $\sigma \vDash F$ means that F holds over σ
- If P is a state predicate, then $\sigma \vDash P$ means that P holds over the first state in σ
- If A is an action, then $\sigma \vDash A$ means that A holds over the first two states in σ
  - i.e., the first step in σ is an A step
  - note that a state predicate is simply an action without primed variables
- If A is an action, then $\sigma \vDash [A]_v$ means that the first step in σ is an A step or a stuttering step with respect to v

$\square$Always

- $\sigma \vDash \square F$ means that F holds over every suffix of σ
- More formally
  - Let σ^{+n} be σ with the first n states removed
  - Then $\sigma \vDash \square F \triangleq \forall n \in \mathbb{N}\colon \sigma^{+n} \vDash F$

Not every temporal formula is a TLA+ formula

- TLA+ formulas are temporal formulas that are *invariant under stuttering*
    - They hold even if you add or remove stuttering steps

Eventually an A step occurs…

$\Diamond\langle A\rangle_v \triangleq \neg\Box[\neg A]_v$

HourClock with *liveness*

clock that never stops

Module HourClock

- Variable hr
- HCini $\triangleq hr \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$
- HCnxt $\triangleq hr' = hr \bmod 12 + 1$
- HC $\triangleq$ HCini $\land \Box[\text{HCnxt}]_{hr}$
- LiveHC $\triangleq$ HC $\land \Box(\Diamond\langle \text{HCnxt} \rangle_{hr})$

Weak Fairness as a liveness condition

- $\text{ENABLED } \langle A \rangle_v$ means action A is possible in some state
- $WF_v(A) \triangleq \Box(\Box\text{ENABLED } \langle A \rangle_v \Rightarrow \Diamond\langle A \rangle_v)$

- HourClock: $WF_{hr}(HCnxt)$

Strong Fairness

- $SF_v(A) \triangleq \Diamond\Box(\neg\text{ENABLED}\ \langle A\rangle_v) \lor \Box\Diamond\langle A\rangle_v$

- A is eventually disabled forever or infinitely many A steps occur

$SF_v(A)$: an A step must occur if A is continually enabled

$WF_v(A)$: an A step must occur if A is continuously enabled

As always, better to make the weaker assumption if you can

How important is liveness?

- Liveness rules out behaviors that have only stuttering steps
  - Add non-triviality of a specification
- In practice, “eventual” is often not good enough
- Instead, need to specify performance requirements
  - Service Level Objectives (SLOs)
  - Usually done quite informally

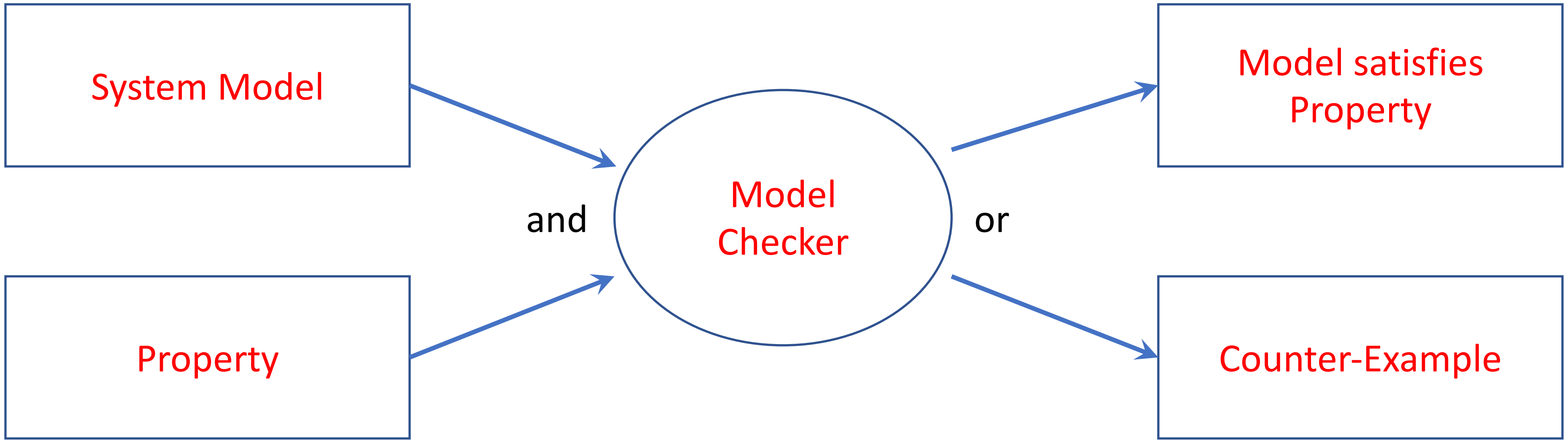
What is Model Checking?

- Check whether a finite state machine satisfies certain properties
- More generally: check whether the set of behaviors of one specification is a subset of the behaviors of another
  - Or even check whether two different specs are equivalent
- By exploring all possible executions of the FSM
- Suffers from combinatorial explosion
  - But still useful for “small” models
- Very successful for hardware designs

Turing Awards

- Amir Pnueli received the 1996 Turing award for "seminal work introducing temporal logic into computing science”
  - Led to checking models where the specification is given by a temporal logic formula
- Edmund Clarke (Cornell Ph.D. 1976), Allen Emerson, and Joseph Sifaki received to 2007 Turing award for their seminal work founding and developing the field of model checking
- Leslie Lamport received the 2013 Turing award for imposing clear, well-defined coherence on the seemingly chaotic behavior of distributed computing systems […]
  - And the development of TLA+ and TLC can be considered part of this

Basic Concept

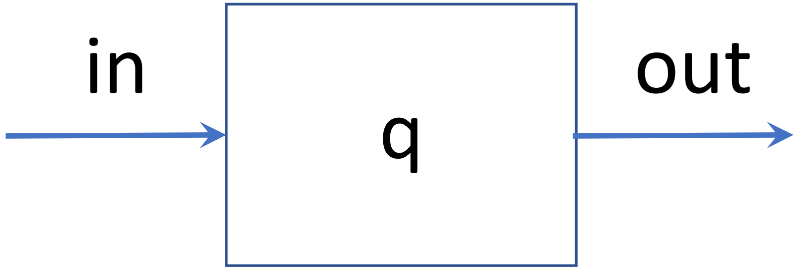


TLC Model Checker

- Model: $Spec \triangleq Init \land \Box[Next]_{vars} \land Temporal$
- TLC checks for
  - ”Sillliless errors”: 1/0, 1/”string”, $\langle 1, 2, 3\rangle[10]$, … (things that are undefined)
  - Deadlock: states where $Next$ is not enabled
  - User-specified properties
- Two modes:
  - Model check: explore all states
  - Simulate: explore randomly generated behaviors

Finite State Models

- Model Checkers can only check finite state models
- Many specs are not finite state
  - Recall “FIFO” spec: allows for arbitrarily long queues



- Need to add *constraints* on allowable states
  - Recall “BoundedFIFO” spec, where we bounded the size of the queue

MODULE *BoundedFIFO*

EXTENDS *Naturals*, *Sequences*

VARIABLES *in*, *out*

CONSTANT *Message*, *N*

ASSUME $(N \in Nat) \land (N > 0)$

$Inner(q) \triangleq$ INSTANCE *InnerFIFO*

$$BNext(q) \triangleq \begin{array}{l} \land\ Inner(q)!Next \\ \land\ Inner(q)!BufRcv \Rightarrow (Len(q) < N) \end{array}$$

If it is a *BufRcv* step,
then $len(q) < N$

$$Spec \triangleq \exists\, q : Inner(q)!Init \land \Box[BNext(q)]_{\langle in, out, q \rangle}$$

Other limitations

- CONSTANTS must all be specific
  - Although can support "model values", e.g.: $Data \leftarrow \{d1, d2, d3\}$
  - Model values are any identifiers
- Does not support unbounded quantification or CHOOSE
- Does not support $\exists$ (the temporal existential quantifier)
  - See previous page
  - Must model check InnerFIFO instead
- Variables can only contain "TLC values"
  - See next page

TLC values

- Primitive values: Boolean, Integers, Strings, …
- Model values: d1, d2, …
- Finite sets of TLC values
  - But have to be “comparable”: { “x”, 1 } is not allowed
- Functions whose domains and ranges are TLC values
  - Includes tuples

- Nat is not a TLC value
- Therefore $[x \in Nat \rightarrow x + 1]$ is not a TLC value
- However, it will turn out that $[x \in Nat \rightarrow x + 1][3]$ can be evaluated and renders the TLC value 4

Example: HourClock

VARIABLE hr

HCini $\triangleq hr \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$

HCnxt $\triangleq hr' = hr \ \% \ 12 + 1$

HC $\triangleq$ HCini $\land \square[\text{HCnxt}]_{hr} \land \square(\diamond\langle\text{HCnxt}\rangle_{hr})$

HCTypeInvariant $\triangleq \square$ HCini

- No constants
- Variable can only contains integers
- State space is bounded

TLA+ is a *macro preprocessor*

VARIABLE hr

HC $\triangleq hr \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\} \land$

$\square[hr' = hr \,\%\, 12 + 1]_{hr} \land \square(\Diamond\langle hr' = hr \,\%\, 12 + 1\rangle_{hr})$

HCTypeInvariant $\triangleq \square\ hr \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$

- When done, all substitutions have been performed:
  - There are no “calls” to operators in expressions
  - There are no references to constants
  - There are no LET expressions
  - There are no INSTANCE calls to other modules
- Semantics of each of these are described in book (and rather complicated), but not really needed

Evaluating (non-primed, non-temporal) expressions

- Mostly done "left to right"
  - $expr_1 + expr_2$
    - First evaluates $expr_1$ then $expr_2$ the adds the results
  - IF $expr_1$ THEN $expr_2$ ELSE $expr_3$
    - First evaluates $expr_1$, then evaluates *either* $expr_2$ or $expr_3$
- Why does it matter?
  - 1/0 is not a TLC value, and 1/0 would throw an error
  - IF $x \neq 0$ THEN 1/x ELSE -1 does not throw an error if $x = 0$
  - Similarly, $x \neq 0 \land 1/x < 3$ simply evaluates to FALSE if $x = 0$
  - But mathematically equivalent $1/x < 3 \land x \neq 0$ throws an error in TLC!

Evaluating primed expressions

- $v' = 3$ evaluates to TRUE iff v' does not have a value or if v' = 3 already
  - In the first case, v' receives the value 3
- In all other cases, v' throws an error iff v' does not have a value
- Note that mathematically equivalent $v' = 3$ and $3 = v'$ behave differently if v' does not have a value
- Note that $v' = v$ (aka UNCHANGED v) always evaluates to TRUE, but assigns v' its former value v if it did not yet have a value

Quiz

What is the value of evaluating $(FALSE \ \wedge v' = 3) \vee (TRUE \ \wedge v' = 4)$ and what is the effect on the value of v'?

v' before	$(FALSE \ \wedge v' = 3) \vee (TRUE \ \wedge v' = 4)$	v' after
3		
4		
5		
unassigned		

Recall: Asynchronous FIFO Channel Specification

$TypeInvariant \triangleq \land\ val \in Data$
$\qquad \land\ rdy \in \{0, 1\}$
$\qquad \land\ ack \in \{0, 1\}$

$Init \triangleq \land\ val \in Data$
$\qquad \land\ rdy \in \{0, 1\}$
$\qquad \land\ ack = rdy$

$Send \triangleq \land\ rdy = ack$
$\qquad \land\ val' \in Data$
$\qquad \land\ rdy' = 1 - rdy$
$\qquad \land\ ack' = ack$

$Rcv \triangleq \land\ rdy \neq ack$
$\qquad \land\ ack' = 1 - ack$
$\qquad \land\ val' = val$
$\qquad \land\ rdy' = rdy$

$Next \triangleq Send \lor Recv$

$Spec \triangleq \text{Init} \land \square[\text{Next}]_{\langle rdy, ack, val \rangle}$

Quiz

What is the value of evaluating $(v' = 2 \lor v' = 3) \land v' = 3$ and what is the effect on the value of v'?

v' before	$(v' = 2 \lor v' = 3) \land v' = 3$	v' after
2		
3		
4		
unassigned		

Computing States

- TLC evaluates disjunctions in primed formulas in a different way
  - $x \lor y$
  - $\exists x \in S: P(x)$
  - $x \Rightarrow y \quad (\equiv \neg x \lor y)$
  - x’ ∈ S $\quad (\equiv \exists y \in S: x' = y)$

- It evaluates all branches even if one branch evaluates to TRUE
- Each may lead to a different state
- Computing next states is SAT solving…

Example

$$
\begin{aligned}
\lor\ &\land\ x' \in 1 \mathrel{..} Len(y) \\
&\land\ y' = Append(Tail(y), x') \\
\lor\ &\land\ x' = x + 1 \\
&\land\ y' = Append(y, x')
\end{aligned}
$$

$$
\begin{aligned}
x &= 1 \\
y &= \langle 2, 3 \rangle \\
x' &= unassigned \\
y' &= unassigned
\end{aligned}
$$

Example

$$x' \in \{1, 2\}$$
$$y' = Append(\langle 3 \rangle, x')$$

$$\begin{aligned}
&\lor \land x' \in 1 \,..\, Len(y) \\
&\quad \land y' = Append(Tail(y), x') \\
&\lor \land x' = x + 1 \\
&\quad \land y' = Append(y, x')
\end{aligned}$$

$$x = 1$$
$$y = \langle 2, 3 \rangle$$
$$x' = unassigned$$
$$y' = unassigned$$

Example

$$x' = 1$$
$$y' = \langle 3, 1 \rangle$$

$$x' \in \{1, 2\}$$
$$y' = Append(\langle 3 \rangle, x')$$

$$\begin{aligned}
\lor\ & \land\ x' \in 1 \,..\, Len(y) \\
& \land\ y' = Append(Tail(y), x') \\
\lor\ & \land\ x' = x + 1 \\
& \land\ y' = Append(y, x')
\end{aligned}$$

$$x = 1$$
$$y = \langle 2, 3 \rangle$$
$$x' = unassigned$$
$$y' = unassigned$$

Example

$$\begin{aligned}
\lor\ & \land\ x' \in 1\,..\,Len(y) \\
& \land\ y' = Append(Tail(y), x') \\
\lor\ & \land\ x' = x + 1 \\
& \land\ y' = Append(y, x')
\end{aligned}$$

$$x = 1$$
$$y = \langle 2, 3 \rangle$$
$$x' = unassigned$$
$$y' = unassigned$$

$$x' \in \{1, 2\}$$
$$y' = Append(\langle 3 \rangle, x')$$

$$x' = 1$$
$$y' = \langle 3, 1 \rangle$$

$$x' = 2$$
$$y' = \langle 3, 2 \rangle$$

Example

$$\begin{aligned}
\lor\ &\land\ x' \in 1\ ..\ Len(y) \\
&\land\ y' = Append(Tail(y), x') \\
\lor\ &\land\ x' = x + 1 \\
&\land\ y' = Append(y, x')
\end{aligned}$$

$$x = 1$$
$$y = \langle 2, 3 \rangle$$
$$x' = unassigned$$
$$y' = unassigned$$

$$x' \in \{1, 2\}$$
$$y' = Append(\langle 3 \rangle, x')$$

$$x' = 1$$
$$y' = \langle 3, 1 \rangle$$

$$x' = 2$$
$$y' = \langle 3, 2 \rangle$$

$$x' = 2$$
$$y' = Append(\langle 2, 3 \rangle, 2)$$

Example

$$
\begin{aligned}
\lor\ &\land\ x' \in 1 \,..\, Len(y) \\
&\land\ y' = Append(Tail(y), x') \\
\lor\ &\land\ x' = x + 1 \\
&\land\ y' = Append(y, x')
\end{aligned}
$$

$$
\begin{gathered}
x = 1 \\
y = \langle 2, 3 \rangle \\
x' = unassigned \\
y' = unassigned
\end{gathered}
$$

$$
\begin{gathered}
x' \in \{1, 2\} \\
y' = Append(\langle 3 \rangle, x')
\end{gathered}
$$

$$
\begin{gathered}
x' = 1 \\
y' = \langle 3, 1 \rangle
\end{gathered}
$$

$$
\begin{gathered}
x' = 2 \\
y' = \langle 3, 2 \rangle
\end{gathered}
$$

$$
\begin{gathered}
x' = 2 \\
y' = Append(\langle 2, 3 \rangle, 2)
\end{gathered}
$$

$$
\begin{gathered}
x' = 2 \\
y' = \langle 2, 3, 2 \rangle
\end{gathered}
$$

Computing Next States

- Start with a completely unassigned next state
- Then recursively
- For each expression
  - $=, \wedge, \text{ and } \vee$ expressions are special
- And for each next state under consideration
- Evaluate possibly multiple resulting next states
- And for each such next state the value of the expression

TLC algorithm to compute all behaviors (including infinite ones)

- State of TLC model checker:
  - $G = (V, E)$: directed graph of states. Edge from s -> s' if s' is reachable from s through the *Next* relation
  - $U \subseteq V$: set of states whose next states have not yet been computed
- Initialization: compute set of initial states and add them to V and U
  - Much like computing *Next* states
  - Indeed, simply compute *Init'* essentially
- while $U \neq \emptyset$:
  - Select s in U
  - Compute T: set of next states from s
  - If $T = \emptyset$ report deadlock
  - Add $T \setminus V$ to U
  - Remove s from U
  - Add T to V and add edges from s to the states in T to E
- Add self-edges to each state in V

TLC algorithm to compute all behaviors (including infinite ones)

- State of TLC model checker:
  - $G = (V, E)$: directed graph of states. Edge from s -> s’ if s’ is reachable from s through the *Next* relation
  - $U \subseteq V$: set of states whose next states have not yet been computed
- Initialization: compute set of initial states and add them to V and U
  - Much like computing *Next* states
  - Indeed, simply compute *Init’* essentially
- while $U \neq \emptyset$:
  - Select s in U
  - Compute T: set of next states from s
  - If $T = \emptyset$ report deadlock
  - Add $T \setminus V$ to U
  - Remove s from U
  - Add T to V and add edges from s to the states in T to E
- Add self-edges to each state in V

Resulting G is a “Kripke Structure”

Checking properties

- Check safety (invariant) properties in each state that is computed
  - Property of the form $\square P$, where P is a state predicate
  - If property is violated, report shortest path from an initial state to the state that violates the safety property
- Check liveness (fairness) properties, for example:
  - For $\diamond P$, check that a state satisfying P is reachable from any initial state
  - For $\diamond \square P$, check that a state satisfying P is reachable from any initial state, and that any state reachable from there satisfies P as well
  - For $\square \diamond P$, check that a state satisfying P is reachable from any state

Leveraging Symmetry

- Recall Peterson: the two processes have identical specs
- Hence swapping the processes doesn’t change anything
- In general, it is often the case in concurrent algorithms that permuting a set of processes doesn’t change anything
- You can tell TLC this: SYMMETRY Permutations(Procs)
- If there are n processes, reduces the state space by n! (n factorial)
- There are often other symmetries, such as the set of memory addresses
    - Other model checkers also leverage symbolic execution for improved efficiency

“Be suspicious of success”

- Try out properties that should not hold and see if TLC finds the bug
- A finite model may have properties not held by the actual implementation, which might have an infinite number of states
  - In theory, TLC can find any safety violation; you must just pick a model large enough to find it
  - Not so for liveness violations

PlusPy

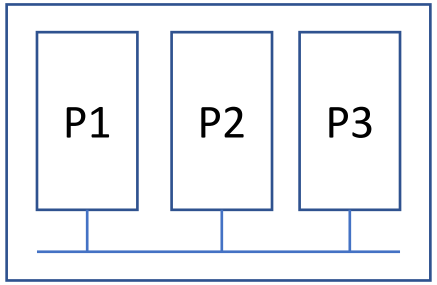
- RVR's TLA+ interpreter in Python
- Why an interpreter?
  - Can *test* models that TLC can't (fewer restrictions)
  - Can be used for safety-critical code (no hand translation)
- Why Python?
  - In some way like TLA+
    - Big integers
    - No types
    - No expectation that it'll be fast ☺

Distributed PlusPy

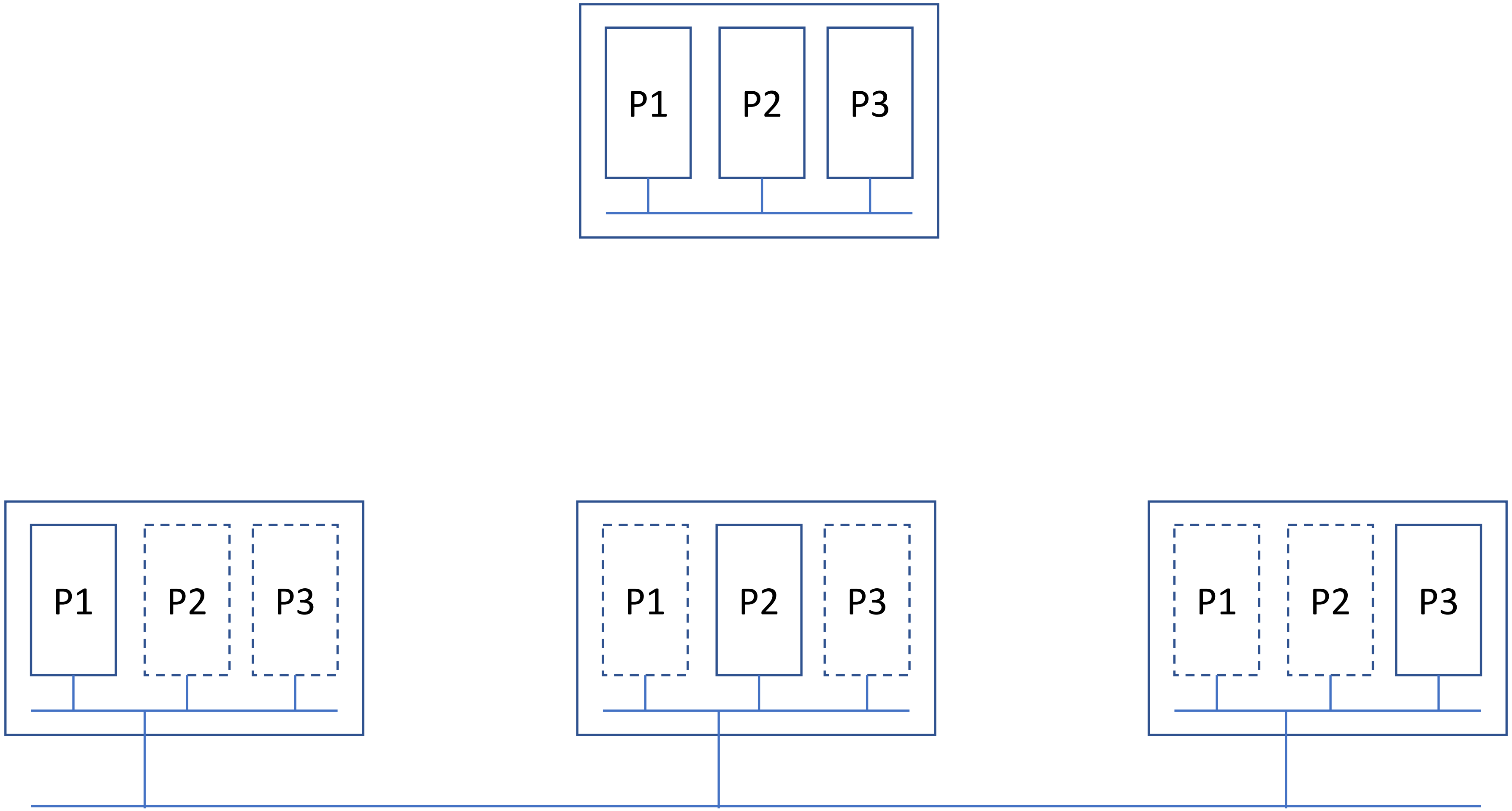
- Distributed (and concurrent) specs usually written like this:
  - Init == …
  - Proc(p) == …
  - Next == \E p \in Processes: Proc(p)
  - Spec == Init /\ [][Next]

- Processes communicate through “interface variable”
  - Like the queue in FIFO
- PlusPy has option to only evaluate Proc(p) for one specific p
- PlusPy supports “distributed interface variables”

Distributed PlusPy Illustrated



Distributed PlusPy Illustrated



Distributed PlusPy Illustrated