Introduction

EECS4315 Z:
Mission-Critical Systems
Winter 2025

CHEN-WEI WANG

Learning Outcomes

This module is designed to help you understand:

- ***Mission***-Critical Systems vs. ***Safety***-Critical Systems
- **Code of Ethics** for Professional Engineers
- What a ***Formal Method*** Is
- ***Verification*** vs. ***Validation***
- Catching **Defects**: When?
- ***Model***-Based Development: EECS3342 vs. EECS4315

What is a Safety-Critical System (SCS)?

- A *safety-critical system (**SCS**)* is a system whose ***failure*** or ***malfunction*** has one (or more) of the following consequences:
  - death or serious injury to **people**
  - loss or severe damage to **equipment/property**
  - harm to the **environment**
- Based on the above definition, do you know of any systems that are ***safety-critical***?

Professional Engineers: Code of Ethics

- *Code of Ethics* is a basic guide for ***professional conduct*** and imposes duties on practitioners, with respect to **society**, **employers**, **clients**, **colleagues** (including employees and subordinates), the **engineering profession** and him or herself.
- It is the duty of a practitioner to act at all times with,
  1. ***fairness*** and ***loyalty*** to the practitioner's associates, employers, clients, subordinates and employees;
  2. ***fidelity*** (i.e., dedication, faithfulness) to public needs;
  3. devotion to ***high ideals*** of personal honour and professional integrity;
  4. ***knowledge*** of developments in the area of professional engineering relevant to any services that are undertaken; and
  5. ***competence*** in the performance of any professional engineering services that are undertaken.
- Consequence of misconduct?
  - ***suspension*** or ***termination*** of professional licenses
  - civil ***law suits***

Source: PEO's Code of Ethics

Developing Safety-Critical Systems

Industrial standards in various domains list *acceptance criteria* for **mission**- or **safety**-critical systems that practitioners need to comply with: e.g.,

- ***Aviation*** Domain: ***RTCA DO-178C*** "*Software Considerations in Airborne Systems and Equipment Certification*"
- ***Nuclear*** Domain: ***IEEE 7-4.3.2*** "*Criteria for Digital Computers in Safety Systems of Nuclear Power Generating Stations*"

Two important criteria are:

1. System ***requirements*** are precise and complete
2. System ***implementation*** conforms to the requirements

But how do we accomplish these criteria?

Safety-Critical vs. Mission-Critical?

- ***Critical***:

  A task whose successful completion ensures the success of a larger, more complex operation.

  e.g., Success of a pacemaker $\Rightarrow$ Regulated heartbeats of a patient
- ***Safety***:

  Being free from danger/injury to or loss of human lives.
- ***Mission***:

  An operation or task assigned by a higher authority.

  Q. Formally relate being ***safety***-critical and ***mission***-critical.

  A.
  - ***safety***-critical $\Rightarrow$ ***mission***-critical
  - ***mission***-critical $\not\Rightarrow$ ***safety***-critical
- Relevant industrial standard: ***RTCA DO-178C*** (replacing RTCA DO-178B in 2012) "*Software Considerations in Airborne Systems and Equipment Certification*"

Source: Article from OpenSystems

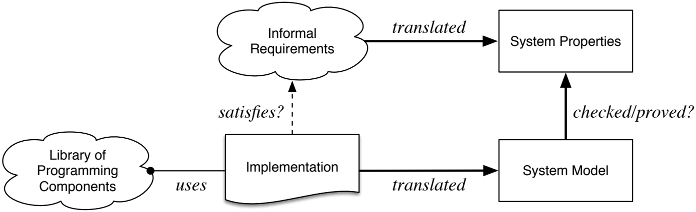
Using Formal Methods for Certification

- A ***formal method (FM)*** is a ***mathematically rigorous*** technique for the specification, development, and verification of software and hardware systems.
- ***DO-333*** "*Formal methods supplement to DO-178C and DO-278A*" advocates the use of formal methods:

  *The use of **formal methods** is motivated by the expectation that, as in other engineering disciplines, performing appropriate **mathematical analyses** can contribute to establishing the **correctness** and **robustness** of a design.*

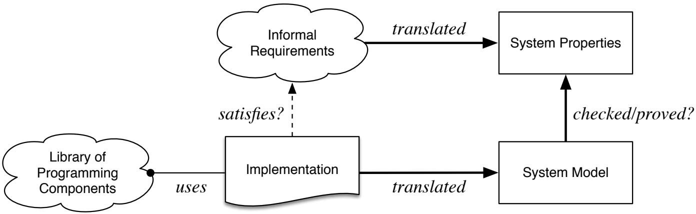
- FMs, because of their mathematical basis, are capable of:
  - *Unambiguously* describing software system requirements.
  - Enabling *precise* communication between engineers.
  - Providing ***verification (towards certification) evidence*** of:
    - A ***formal*** representation of the system being *healthy*.
    - A ***formal*** representation of the system *satisfying* *safety properties*.

Verification: Building the Product Right?



- ***Implementation*** built via ***reusable programming components***.
- *Goal* : *Implementation Satisfies Intended Requirements*
- To verify this, we ***formalize*** them as a ***system model*** and a set of (e.g., safety) ***properties***, using the specification language of a theorem prover (EECS3342) or a model checker (EECS4315).
- Two Verification Issues:
  1. Library components may ***not behave as intended***.
  2. Successful checks/proofs ensure that we *built the product right*, with respect to the informal requirements. **But**...

Validation: Building the Right Product?



- Successful checks/proofs $\not\Rightarrow$ We ***built the right product***.
- The target of our checks/proofs may not be valid:

  The requirements may be ***ambiguous***, ***incomplete***, or ***contradictory***.
- Solution: ***Precise Documentation*** [EECS4312]

Catching Defects – When?

- To minimize *development costs*, minimize *software defects*.
- Software Development Cycle:

  Requirements → *Design* → *Implementation* → Release

  Q. Design or Implementation Phase?

  Catch defects *as early as possible*.

Design and architecture	Implementation	Integration testing	Customer beta test	Postproduct release
1X*	5X	10X	15X	30X

  ∵ The cost of fixing defects *increases exponentially* as software progresses through the development lifecycle.

- Discovering *defects* after **release** costs up to 30 times more than catching them in the **design** phase.
- Choice of a *design language*, amendable to ***formal verification***, is therefore critical for your project.

Source: IBM Report

Model-Based Development in EECS3342

- ***Modelling*** and ***formal reasoning*** should be performed **before** implementing/coding a system.
  - A system's ***model*** is its ***abstraction***, filtering irrelevant details.

    A system ***model*** means as much to a software engineer as a ***blueprint*** means to an architect.
  - A system may have a list of ***models***, "sorted" by **accuracy**:

    $$\langle m_0, m_1, \ldots, \boxed{m_i}, \boxed{m_j}, \ldots, m_n \rangle$$

    - The list starts by the most ***abstract*** model with least details.
    - A more ***abstract*** model $\boxed{m_i}$ is said to be ***refined by*** its subsequent, more ***concrete*** model $\boxed{m_j}$.
    - The list ends with the most ***concrete/refined*** model with most details.
  - It is far easier to reason about:
    - a system's ***abstract*** models (rather than its full ***implementation***)
    - ***refinement steps*** between subsequent models
- The final product is ***correct by construction***.

Model-Based Development in EECS4315

- ***Modelling*** and ***formal reasoning*** should be performed **before** implementing/coding a system.
  - A system's ***model*** is its ***abstraction***, filtering irrelevant details.

    A system ***model*** means as much to a software engineer as a ***blueprint*** means to an architect.
- A design ***model*** m specified at the "right" level of ***abstraction***:

  State space **not** causing a ***state explosion***.
  - m is checked against ***invariant*** and ***temporal*** properties.
  - m may be added with more details (e.g., variables) to result in a more "refined" model m'.
  - m' is consistent with (or "refines") m as long as:
    - **No** ***combinatorial explosion*** from variable ranges
    - **All** properties that m passes also pass in m'.

TLA+: An Industrial Strength Toolbox

From https://lamport.azurewebsites.net/tla/tla.html:

> *TLA+ (**Temporal Logic of Actions**) is a **high-level language** for modeling programs and systems–especially concurrent and distributed ones. It's based on the idea that the best way to describe things precisely is with **simple mathematics**. TLA+ and its tools are useful for eliminating fundamental **design errors**, which are hard to find and expensive to correct in code.*

> TLA+ is a language for modeling ***software*** above the code level and ***hardware*** above the circuit level. It has an ***IDE*** (Integrated Development Environment) for writing models and running tools to check them. The tool most commonly used by engineers is the ***TLC model checker***, but there is also a proof checker. TLA+ is based on mathematics and does not resemble any programming language. Most engineers will find *PlusCal*, described below, to be the easiest way to start using TLA+.

Beyond this lecture ...

- The ***TLA+ toolbox*** has been report about its use in industry: `https://lamport.azurewebsites.net/tla/industrial-use.html`
- Two papers have been made available on eClass:
  - Newcombe, C. ***Why Amazon Chose TLA+***. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pp 25 – 39. Springer (2014).
  - Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M. ***How Amazon Web Services Uses Formal Methods***. In *Communications of the ACM*, 58(4), pp 66 – 73. ACM (2015).
- You're encouraged to read them first: we will guide you through some highlights later in the course (after you've gained experience on the TLA+ toolbox).

Index (1)

Index (2)

Review of Math

EECS4315 Z:
Mission-Critical Systems
Winter 2025

Chen-Wei Wang

Learning Outcomes of this Lecture

This module is designed to help you **review**:

- Propositional Logic
- Predicate Logic

Propositional Logic (1)

- A *proposition* is a statement of claim that must be of either *true* or *false*, but not both.
- Basic logical operands are of type Boolean: *true* and *false*.
- We use logical operators to construct compound statements.
  - Unary logical operator: negation ($\neg$)

p	$\neg p$
true	*false*
false	*true*

  - Binary logical operators: conjunction ($\wedge$), disjunction ($\vee$), implication ($\Rightarrow$), equivalence ($\equiv$), and if-and-only-if ($\iff$).

p	q	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \iff q$	$p \equiv q$
true	*true*	*true*	*true*	*true*	*true*	*true*
true	*false*	*false*	*true*	*false*	*false*	*false*
false	*true*	*false*	*true*	*true*	*false*	*false*
false	*false*	*false*	*false*	*true*	*true*	*true*

Propositional Logic: Implication (1)

- Written as $p \Rightarrow q$ [pronounced as "p implies q"]
  - We call p the antecedent, assumption, or premise.
  - We call q the consequence or conclusion.
- Compare the *truth* of $p \Rightarrow q$ to whether a contract is *honoured*:
  - antecedent/assumption/premise $p \approx$ promised terms [e.g., salary]
  - consequence/conclusion $q \approx$ obligations [e.g., duties]
- When the promised terms are met, then the contract is:
  - *honoured* if the obligations fulfilled. [$(true \Rightarrow true) \iff true$]
  - *breached* if the obligations violated. [$(true \Rightarrow false) \iff false$]
- When the promised terms are not met, then:
  - Fulfilling the obligation (q) or not ($\neg q$) does *not breach* the contract.

p	q	$p \Rightarrow q$
false	*true*	*true*
false	*false*	*true*

Propositional Logic: Implication (2)

There are alternative, equivalent ways to expressing $p \Rightarrow q$:

- q **if** p

  q is *true* if p is *true*

- p **only if** q

  If p is *true*, then for $p \Rightarrow q$ to be *true*, it can only be that q is also *true*.
  Otherwise, if p is *true* but q is *false*, then $(true \Rightarrow false) \equiv false$.

  Note. To prove $p \equiv q$, prove $p \iff q$ (pronounced: "p if and only if q"):

  - p **if** q [$p \Leftarrow q \equiv q \Rightarrow p$]
  - p **only if** q [$p \Rightarrow q$]

- p is **sufficient** for q [similar to q **if** p]

  For q to be *true*, it is sufficient to have p being *true*.

- q is **necessary** for p [similar to p **only if** q]

  If p is *true*, then it is necessarily the case that q is also *true*.
  Otherwise, if p is *true* but q is *false*, then $(true \Rightarrow false) \equiv false$.

- q **unless** $\neg p$ [When is $p \Rightarrow q$ *true*?]

  If q is *true*, then $p \Rightarrow q$ *true* regardless of p.
  If q is *false*, then $p \Rightarrow q$ cannot be *true* unless p is *false*.

Propositional Logic: Implication (3)

Given an implication $p \Rightarrow q$, we may construct its:

- **Inverse**: $\neg p \Rightarrow \neg q$ [negate antecedent and consequence]
- **Converse**: $q \Rightarrow p$ [swap antecedent and consequence]
- **Contrapositive**: $\neg q \Rightarrow \neg p$ [inverse of converse]

Propositional Logic (2)

- **Axiom**: Definition of $\Rightarrow$

$$p \Rightarrow q \equiv \neg p \vee q$$

- **Theorem**: Identity of $\Rightarrow$

$$true \Rightarrow p \equiv p$$

- **Theorem**: Zero of $\Rightarrow$

$$false \Rightarrow p \equiv true$$

- **Axiom**: De Morgan

$$\begin{array}{rcl} \neg(p \wedge q) & \equiv & \neg p \vee \neg q \\ \neg(p \vee q) & \equiv & \neg p \wedge \neg q \end{array}$$

- **Axiom**: Double Negation

$$p \equiv \neg\,(\neg\, p)$$

- **Theorem**: Contrapositive

$$p \Rightarrow q \equiv \neg q \Rightarrow \neg p$$

Predicate Logic (1)

- A *predicate* is a ***universal*** or ***existential*** statement about objects in some universe of disclosure.
- Unlike propositions, predicates are typically specified using ***variables***, each of which declared with some ***range*** of values.
- We use the following symbols for common numerical ranges:
  - $\mathbb{Z}$: the set of integers $[\,-\infty, \dots, -1, 0, 1, \dots, +\infty\,]$
  - $\mathbb{N}$: the set of natural numbers $[\,0, 1, \dots, +\infty\,]$
- Variable(s) in a predicate may be ***quantified***:
  - *Universal quantification* :
    All values that a variable may take satisfy certain property.
    e.g., Given that *i* is a natural number, *i* is *always* non-negative.
  - *Existential quantification* :
    Some value that a variable may take satisfies certain property.
    e.g., Given that *i* is an integer, *i* *can be* negative.

Predicate Logic (2.1): Universal Q. ($\forall$)

- A *universal quantification* has the form $(\forall X \bullet R \Rightarrow P)$
  - X is a comma-separated list of variable names
  - R is a *constraint on types/ranges* of the listed variables
  - P is a *property* to be satisfied
- ***For all*** (combinations of) values of variables listed in X that satisfies R, it is the case that P is satisfied.
  - $\forall i \bullet i \in \mathbb{N} \Rightarrow i \geq 0$ [*true*]
  - $\forall i \bullet i \in \mathbb{Z} \Rightarrow i \geq 0$ [*false*]
  - $\forall i, j \bullet i \in \mathbb{Z} \land j \in \mathbb{Z} \Rightarrow i < j \lor i > j$ [*false*]
- *Proof Strategies*
  1. How to prove $(\forall X \bullet R \Rightarrow P)$ ***true***?
     - **Hint.** When is $R \Rightarrow P$ ***true***? [*true* $\Rightarrow$ *true*, *false* $\Rightarrow$ _]
     - Show that for all instances of $x \in X$ s.t. $R(x)$, $P(x)$ holds.
     - Show that for all instances of $x \in X$ it is the case $\neg R(x)$.
  2. How to prove $(\forall X \bullet R \Rightarrow P)$ ***false***?
     - **Hint.** When is $R \Rightarrow P$ ***false***? [*true* $\Rightarrow$ *false*]
     - Give a ***witness/counterexample*** of $x \in X$ s.t. $R(x)$, $\neg P(x)$ holds.

Predicate Logic (2.2): Existential Q. ($\exists$)

- An *existential quantification* has the form $(\exists X \bullet R \wedge P)$
  - X is a comma-separated list of variable names
  - R is a *constraint on types/ranges* of the listed variables
  - P is a *property* to be satisfied
- ***There exist*** (a combination of) values of variables listed in X that satisfy both R and P.
  - $\exists i \bullet i \in \mathbb{N} \wedge i \geq 0$ [*true*]
  - $\exists i \bullet i \in \mathbb{Z} \wedge i \geq 0$ [*true*]
  - $\exists i, j \bullet i \in \mathbb{Z} \wedge j \in \mathbb{Z} \wedge (i < j \vee i > j)$ [*true*]
- *Proof Strategies*
  1. How to prove $(\exists X \bullet R \wedge P)$ ***true***?
     - **Hint.** When is $R \wedge P$ ***true***? [$true \wedge true$]
     - Give a ***witness*** of $x \in X$ s.t. $R(x)$, $P(x)$ holds.
  2. How to prove $(\exists X \bullet R \wedge P)$ ***false***?
     - **Hint.** When is $R \wedge P$ ***false***? [$true \wedge false$, $false \wedge \_$]
     - Show that for all instances of $x \in X$ s.t. $R(x)$, $\neg P(x)$ holds.
     - Show that for all instances of $x \in X$ it is the case $\neg R(x)$.

Predicate Logic (3): Exercises

- Prove or disprove: $\forall x \bullet (x \in \mathbb{Z} \land 1 \leq x \leq 10) \Rightarrow x > 0$.

  All 10 integers between 1 and 10 are greater than 0.
- Prove or disprove: $\forall x \bullet (x \in \mathbb{Z} \land 1 \leq x \leq 10) \Rightarrow x > 1$.

  Integer 1 (a witness/counterexample) in the range between 1 and 10 is *not* greater than 1.
- Prove or disprove: $\exists x \bullet (x \in \mathbb{Z} \land 1 \leq x \leq 10) \land x > 1$.

  Integer 2 (a witness) in the range between 1 and 10 is greater than 1.
- Prove or disprove that $\exists x \bullet (x \in \mathbb{Z} \land 1 \leq x \leq 10) \land x > 10$?

  All integers in the range between 1 and 10 are *not* greater than 10.

Predicate Logic (4): Switching Quantifications

Conversions between ∀ and ∃:

$$(\forall X \bullet R \Rightarrow P) \iff \neg(\exists X \bullet R \wedge \neg P)$$
$$(\exists X \bullet R \wedge P) \iff \neg(\forall X \bullet R \Rightarrow \neg P)$$

Index (1)

Verification by Model Checking

EECS4315 Z:
Mission-Critical Systems
Winter 2025

CHEN-WEI WANG

Motivation for Formal Verification

- ***Safety-Critical Systems***
  e.g., shutdown system of a nuclear power plant
- ***Mission-Critical Systems***
  e.g., mass-produced computer chips
- ***Formal verification*** of the *correctness* of critical systems can prevent loss of fortune or even lives.
- Formal verification consists of:
  1. ***Systems***:
     Need a **specification** language for modelling abstractions.
  2. ***Properties***: Need a **specification** language for expressing (e.g., safety, temporal) concerns.
  3. ***Verification***: Need a **systematic method** for establishing that a system satisfies the desired properties.
- The **earlier** errors are caught in the course of system development, the **cheaper** it is to rectify.
  - e.g., Much cheaper to catch an error in the design phase than recalling defected products after release.

Example of Formal Verification

Pentium FDIV bug: https://en.wikipedia.org/wiki/Pentium_FDIV_bug

*The Pentium FDIV bug is a hardware bug affecting the **floating-point unit (FPU)** of the early Intel Pentium processors. Because of the bug, the processor would return incorrect binary floating point results when dividing certain pairs of high-precision numbers.*

*In December 1994, Intel **recalled** the defective processors ... In its 1994 annual report, Intel said it incurred "**a $475 million pre-tax charge** ... to recover replacement and write-off of these microprocessors."*

*In the aftermath of the **bug** and subsequent **recall**, there was a marked increase in the use of formal verification of hardware floating point operations across the **semiconductor industry**. Prompted by the discovery of the bug, a technique ... called "word-level **model checking**" was developed in 1996. Intel went on to use **formal verification** extensively in the development of later CPU architectures. In the development of the Pentium 4, symbolic trajectory evaluation and **theorem proving** were used to **find a number of bugs that could have led to a similar recall incident** had they gone undetected.*

Classification of Verification Methods

- **Degree of Automation**: Automatic, Interactive, or Manual
- **~~Model~~Check-based vs. Proof-based**
  - ***Proof***-based:
    - The ***system*** (abstractly) described as a set of formulas Γ
    - ***Properties*** specified as a set of formulas ϕ
    - ***Prove*** (automatically or interactively) that $\Gamma \vdash \phi$ [*undecidable*]

      i.e., Γ can be derived to ϕ (via ***inference rules***).
  - ***Check***-based:
    - The ***system*** (abstractly) described as a *finite* model $\mathbb{M}$
    - ***Properties*** specified as a set of formulas ϕ
    - ***Decide*** (automatically) that $\mathbb{M} \models \phi$ [*decidable*, *algorithmic*]

      i.e., Traversing $\mathbb{M}$'s ***state/reachability graph*** decides if ϕ is satisfied.
- **Domain of Application**
  - Hardware vs. Software
  - Sequential vs. Concurrent
  - Reactive (e.g., bridge controller) vs. Terminating (e.g., sorting alg.)
- **Pre-development vs. Post-development**

Verification via Model Checking

- Automatic, Check-based
- Intended for ***reactive***, ***concurrent*** systems
  - ***Reactivity***:
    Continuous reaction to stimuli from the environment
    e.g., communication protocols, operating systems, embedded systems, etc.
  - ***Concurrency***:
    Simultaneous execution of (independent or inter-dependent) system units, each of which evolving its own states
- ***Testing*** of concurrent, reactive systems is hard:
  - Many scenarios are **non-reproducible**.
  - Hard to **systematically** cover all important interactions
  - E. W. Dijkstra: ***Program testing can be used to show the presence of bugs, but never to show their absence!***
- Originated as a ***post***-development method
- But should be used as ***pre***-development method to save cost

Model Checking: Temporal Logic

- **System**
  - A system model $\mathbb{M}$ is a ***labeled transition system (LTS)*** with a (large) number of states and transitions between states.
  - A ***model*** of an actual physical system ***abstracts away*** details that are irrelevant to the ***properties*** to be checked.
- **Properties**
  - *Temporal logic (TL)* incorporates the notion of ***timing***.
  - A TL formula ϕ is **not** statically true or false.
  - Instead, the truth of a TL formula ϕ depends on where the SUV ***dynamically*** evolves into (by following transitions).
- **Verification**
  - A computer program, called a *model checker*, takes as inputs $\mathbb{M}$ and ϕ, and **decides** if $\mathbb{M} \models \phi$
    - **Yes** $\Rightarrow$ All ***reachable*** states of $\mathbb{M}$ satisfy ϕ.
    - **No** $\Rightarrow$ An ***error trace***, leading to a state satisfying $\neg\phi$, is generated. This facilitates debugging through reproducing a problematic scenario.
    - **Unknown** $\Rightarrow$ The checker runs out of memory due to ***state explosion***.

Linear-Time Temporal Logic (LTL)

- *LTL (Linear-time Temoral Logic)* has connectives/operators which allow us to refer to the ***future***.
- Two features of *LTL*:
  - ***(Computation) Path***:
    Time is modelled as an ***infinite*** sequence of states.
  - ***Undetermined Future***:
    Alternative paths exist, one of which being the "actual" path.

LTL: Syntax in CFG (1)

ϕ	::=	$\top$	[*true*]
	$\mid$	$\bot$	[*false*]
	$\mid$	p	[`propositional atom`]
	$\mid$	$(\neg\phi)$	[`logical negation`]
	$\mid$	$(\phi \wedge \phi)$	[`logical conjunction`]
	$\mid$	$(\phi \vee \phi)$	[`logical disjunction`]
	$\mid$	$(\phi \Rightarrow \phi)$	[`logical implication`]
	$\mid$	$(\mathbf{X}\,\phi)$	[`neXt state`]
	$\mid$	$(\mathbf{F}\,\phi)$	[`some Future state`]
	$\mid$	$(\mathbf{G}\,\phi)$	[`all future states (Globally)`]
	$\mid$	$(\phi\,\mathbf{U}\,\phi)$	[`Until`]
	$\mid$	$(\phi\,\mathbf{W}\,\phi)$	[`Weak-until`]
	$\mid$	$(\phi\,\mathbf{R}\,\phi)$	[`Release`]

p denotes ***atomic***, propositional statements

e.g., Printer `ltr2` is available.

e.g., Reading of sensor `s3` exceeds some threshold.

e.g., The sudoku board is filled out with a correct solution.

LTL: Syntax in CFG (2)

ϕ	::=	$\top$	[*true*]
	$\mid$	$\bot$	[*false*]
	$\mid$	p	[propositional atom]
	$\mid$	$(\neg\phi)$	[logical negation]
	$\mid$	$(\phi \wedge \phi)$	[logical conjunction]
	$\mid$	$(\phi \vee \phi)$	[logical disjunction]
	$\mid$	$(\phi \Rightarrow \phi)$	[logical implication]
	$\mid$	$(\mathbf{X}\,\phi)$	[ne**X**t state]
	$\mid$	$(\mathbf{F}\,\phi)$	[some **F**uture state]
	$\mid$	$(\mathbf{G}\,\phi)$	[all future states (**G**lobally)]
	$\mid$	$(\phi\,\mathbf{U}\,\phi)$	[**U**ntil]
	$\mid$	$(\phi\,\mathbf{W}\,\phi)$	[**W**eak-until]
	$\mid$	$(\phi\,\mathbf{R}\,\phi)$	[**R**elease]

$\forall$ and $\exists$ are embedded in defining the ***temporal*** connectives.

Universe of disclosure: Set of alternative (computation) ***paths***

LTL: Syntax in CFG (3)

ϕ	::=	$\top$	[*true*]
	$\mid$	$\bot$	[*false*]
	$\mid$	p	[propositional atom]
	$\mid$	$(\neg \phi)$	[logical negation]
	$\mid$	$(\phi \land \phi)$	[logical conjunction]
	$\mid$	$(\phi \lor \phi)$	[logical disjunction]
	$\mid$	$(\phi \Rightarrow \phi)$	[logical implication]
	$\mid$	$(\mathbf{X}\,\phi)$	[ne**X**t state]
	$\mid$	$(\mathbf{F}\,\phi)$	[some **F**uture state]
	$\mid$	$(\mathbf{G}\,\phi)$	[all future states (**G**lobally)]
	$\mid$	$(\phi\,\mathbf{U}\,\phi)$	[**U**ntil]
	$\mid$	$(\phi\,\mathbf{W}\,\phi)$	[**W**eak-until]
	$\mid$	$(\phi\,\mathbf{R}\,\phi)$	[**R**elease]

- ***Temporal*** connectives bind **tighter** than ***logical*** ones.
- Unary ***temporal*** connectives bind **tighter** than binary ones.
  - Use parentheses to force the intended order of evaluation.
  - Use a ***parse tree***, a ***LMD***, or a ***RMD*** to verify the order of evaluation.

LTL: Symbols of Unary Temporal Operators

Temporal Connective	Letter	Symbol
Next	**X**	$\bigcirc$
Future/Eventually	**F**	$\Diamond$
Global/Henceforth	**G**	$\Box$

Practical Knowledge about Parsing

- A ***context-free grammar (CFG)*** g
  - defines, **recursively**, **all** (typically an infinite number of) possible strings that can be ***derived*** from it.
  - contains both ***terminals***/***tokens*** (base cases) and ***non-terminals***/***variables*** (recursive cases)
- Given an input string s, to show that $s \in L(g)$, we can either:
  - **Draw** a ***parse tree (PT)*** of s, based on g, where:
    - All ***internal nodes*** (i.e., roots of subtrees) are ϕ (non-terminals).
    - All ***external nodes*** (a.k.a. leaves) are characters of s.
  - **Perform** a ***left-most derivation (LMD)***, by starting with ϕ (the ***start variable***) and continuing to substitute the leftmost non-terminal, until **no** non-terminals remain.
  - **Perform** a ***right-most derivation (RMD)***, by starting with ϕ (the ***start variable***) and continuing to substitute the rightmost non-terminal, until **no** non-terminals remain.
- PTs, LMDs, and RMDs are legitimate, and equivalent, ways for showing ***interpretations*** of a valid LTL formula string.

LTL: Exercises on Parsing Formulas

- Draw and compare the ***parse trees*** of:

$$\mathbf{F}\ p \land \mathbf{G}\ q \Rightarrow p\,\mathbf{U}\,r$$
$$\text{vs. } \mathbf{F}\ (p \land \mathbf{G}\ q \Rightarrow p\,\mathbf{U}\,r)$$
$$\text{vs. } \mathbf{F}\ p \land (\mathbf{G}\ q \Rightarrow p\,\mathbf{U}\,r)$$
$$\text{vs. } \mathbf{F}\ p \land ((\mathbf{G}\ q \Rightarrow p)\,\mathbf{U}\,r)$$

- The above formulas are all ***derivable*** from the grammar of LTL.
  - Show using the ***LMD*** (Left-Most Derivations)
  - Show using the ***RMD*** (Right-Most Derivations)

LTL Formulas: More Exercises

Draw the ***parser trees*** for:

$$(\ \mathbf{F}(p \Rightarrow \mathbf{G}\, r) \vee ((\neg q)\, \mathbf{U}\, p)\ )$$

vs. $\mathbf{F}\, p \Rightarrow \mathbf{G}\, r \vee \neg q\, \mathbf{U}\, p$

vs. $\mathbf{F}(\ (p \Rightarrow \mathbf{G}\, r) \vee (\neg q\, \mathbf{U}\, p)\ )$

LTL Formulas: Subformulas

Given an LTL formula ϕ, its ***subformulas*** are all those whose ***parse trees (rooted at ϕ)*** are subtrees of ϕ's parse tree.

e.g., Enumerate all subformula of ($\mathbf{F}(p \Rightarrow \mathbf{G}\, r) \vee ((\neg q)\, \mathbf{U}\, p)$).

1. p [appearing twice in the parse tree]
2. r
3. $\mathbf{G}\, r$
4. $p \Rightarrow (\mathbf{G}\, r)$
5. $\mathbf{F}(p \Rightarrow (\mathbf{G}\, r))$
6. q
7. $\neg q$
8. p
9. $(\neg q)\, \mathbf{U}\, p$
10. ($\mathbf{F}(p \Rightarrow \mathbf{G}\, r) \vee ((\neg q)\, \mathbf{U}\, p)$)

LTL Semantics: Labelled Transition Systems (LTS)

- **Definition.** Given that P is a set of atoms/propositions of concern, a ***transition system*** $\mathbb{M}$ is a ***formal model*** represented as a triple $\mathbb{M} = (S, \longrightarrow, L)$:
  - S

    A **finite** set of ***states***
  - $\longrightarrow: S \leftrightarrow S$

    A ***transition relation*** on S
  - $L : S \rightarrow \mathbb{P}(P)$

    A ***labelling function*** mapping each state to its satisfying atoms

  Assumption. No state of the system can ***deadlock***:

  From any state, it's always possible to make progress (by taking a transition).

$$\forall s \bullet s \in S \Rightarrow (\ \exists s' \bullet s' \in S \wedge (s, s') \in \longrightarrow\ )$$

Background for Self-Study

- Topics of ***sets*** and ***relations*** were covered in EECS3342.
- Slide 18 to Slide 28 contain what you should recall.

Set of Tuples

Given n sets $S_1, S_2, \ldots, S_n$, a ***cross/Cartesian product*** of theses sets is a set of n-tuples.

Each ***n-tuple*** $(e_1, e_2, \ldots, e_n)$ contains n elements, each of which a member of the corresponding set.

$$S_1 \times S_2 \times \cdots \times S_n = \{(e_1, e_2, \ldots, e_n) \mid e_i \in S_i \land 1 \leq i \leq n\}$$

e.g., $\{a, b\} \times \{2, 4\} \times \{\$, \&\}$ is a set of triples:

$$\begin{aligned}
& \{a, b\} \times \{2, 4\} \times \{\$, \&\} \\
= \; & \{\, (e_1, e_2, e_3) \mid e_1 \in \{a, b\} \land e_2 \in \{2, 4\} \land e_3 \in \{\$, \&\} \,\} \\
= \; & \left\{ \begin{array}{l} (a, 2, \$), (a, 2, \&), (a, 4, \$), (a, 4, \&), \\ (b, 2, \$), (b, 2, \&), (b, 4, \$), (b, 4, \&) \end{array} \right\}
\end{aligned}$$

Relations (1): Constructing a Relation

A *relation* is a set of mappings, each being an ***ordered pair*** that maps a member of set S to a member of set T.

e.g., Say $S = \{1, 2, 3\}$ and $T = \{a, b\}$

- $\varnothing$ is the ***minimum*** relation (i.e., an empty relation).
- $\boxed{S \times T}$ is the ***maximum*** relation (say r_1) between S and T, mapping from each member of S to each member in T:

$$\{(1, a), (1, b), (2, a), (2, b), (3, a), (3, b)\}$$

- $\{(x, y) \mid (x, y) \in S \times T \land x \neq 1\}$ is a relation (say r_2) that maps only some members in S to every member in T:

$$\{(2, a), (2, b), (3, a), (3, b)\}$$

Relations (2.1): Set of Possible Relations

- We use the ***power set*** operator to express the set of ***all possible relations*** on S and T:

$$\mathbb{P}(S \times T)$$

  Each member in $\mathbb{P}(S \times T)$ is a relation.

- To declare a relation variable r, we use the colon (`:`) symbol to mean ***set membership***:

$$r : \mathbb{P}(S \times T)$$

- Or alternatively, we write:

$$r : S \leftrightarrow T$$

  where the set $S \leftrightarrow T$ is synonymous to the set $\mathbb{P}(S \times T)$

Relations (2.2): Exercise

Enumerate $\{a, b\} \leftrightarrow \{1, 2, 3\}$.

- **Hints**:
  - You may enumerate all relations in $\mathbb{P}(\{a, b\} \times \{1, 2, 3\})$ via their *cardinalities*: 0, 1, ..., $|\{a, b\} \times \{1, 2, 3\}|$.
  - What's the *maximum* relation in $\mathbb{P}(\{a, b\} \times \{1, 2, 3\})$?

    $$\{ (a, 1), (a, 2), (a, 3), (b, 1), (b, 2), (b, 3) \}$$
- The answer is a set containing ***all*** of the following relations:
  - Relation with cardinality 0: $\varnothing$
  - How many relations with cardinality 1? $[\ \binom{|\{a,b\} \times \{1,2,3\}|}{1} = 6\ ]$
  - How many relations with cardinality 2? $[\ \binom{|\{a,b\} \times \{1,2,3\}|}{2} = \frac{6 \times 5}{2!} = 15\ ]$

    ...
  - Relation with cardinality $|\{a, b\} \times \{1, 2, 3\}|$:

    $$\{ (a, 1), (a, 2), (a, 3), (b, 1), (b, 2), (b, 3) \}$$

Relations (3.1): Domain, Range, Inverse

Given a relation

$$r = \{(a, 1), (b, 2), (c, 3), (a, 4), (b, 5), (c, 6), (d, 1), (e, 2), (f, 3)\}$$

- ***domain*** **of** r: set of first-elements from r
  - Definition: $\text{dom}(r) = \{\ d \mid (d, r') \in r\ \}$
  - e.g., $\text{dom}(r) = \{a, b, c, d, e, f\}$
- ***range*** **of** r: set of second-elements from r
  - Definition: $\text{ran}(r) = \{\ r' \mid (d, r') \in r\ \}$
  - e.g., $\text{ran}(r) = \{1, 2, 3, 4, 5, 6\}$
- ***inverse*** **of** r: a relation like r with elements swapped
  - Definition: $r^{-1} = \{\ (r', d) \mid (d, r') \in r\ \}$
  - e.g., $r^{-1} = \{(1, a), (2, b), (3, c), (4, a), (5, b), (6, c), (1, d), (2, e), (3, f)\}$

Relations (3.2): Image

Given a relation

$$r = \{(a, 1), (b, 2), (c, 3), (a, 4), (b, 5), (c, 6), (d, 1), (e, 2), (f, 3)\}$$

relational image of r over set s : sub-range of r mapped by s.

- Definition: $r[s] = \{ r' \mid (d, r') \in r \wedge d \in s \}$
- e.g., $r[\{a, b\}] = \{1, 2, 4, 5\}$

Relations (3.3): Restrictions

Given a relation

$$r = \{(a, 1), (b, 2), (c, 3), (a, 4), (b, 5), (c, 6), (d, 1), (e, 2), (f, 3)\}$$

- ***domain restriction*** of r over set ds: sub-relation of r with domain ds.
  - Definition: $ds \lhd r = \{\ (d, r') \mid (d, r') \in r \land d \in ds\ \}$
  - e.g., $\{a, b\} \lhd r = \{(\mathbf{a}, 1), (\mathbf{b}, 2), (\mathbf{a}, 4), (\mathbf{b}, 5)\}$
- ***range restriction*** of r over set rs: sub-relation of r with range rs.
  - Definition: $r \rhd rs = \{\ (d, r') \mid (d, r') \in r \land r' \in rs\ \}$
  - e.g., $r \rhd \{1, 2\} = \{(a, \mathbf{1}), (b, \mathbf{2}), (d, \mathbf{1}), (e, \mathbf{2})\}$

Relations (3.4): Subtractions

Given a relation

r = {(a, 1), (b, 2), (c, 3), (a, 4), (b, 5), (c, 6), (d, 1), (e, 2), (f, 3)}

- ***domain subtraction*** of r over set ds: sub-relation of r with domain not ds.
  - Definition: $ds \lhd r = \{ (d, r') \mid (d, r') \in r \land d \notin ds \}$
  - e.g., $\{a, b\} \lhd r = \{(\mathbf{c}, 3), (\mathbf{c}, 6), (\mathbf{d}, 1), (\mathbf{e}, 2), (\mathbf{f}, 3)\}$
- ***range subtraction*** of r over set rs: sub-relation of r with range not rs.
  - Definition: $r \rhd rs = \{ (d, r') \mid (d, r') \in r \land r' \notin rs \}$
  - e.g., $r \rhd \{1, 2\} = \{(c, \mathbf{3}), (a, \mathbf{4}), (b, \mathbf{5}), (c, \mathbf{6}), (f, \mathbf{3})\}$

Functions (1): Functional Property

- A ***relation*** r on sets S and T (i.e., $r \in S \leftrightarrow T$) is also a *function* if it satisfies the ***functional property***:

$$isFunctional(r)$$
$$\iff$$
$$\forall s, t_1, t_2 \bullet (s \in S \land t_1 \in T \land t_2 \in T) \Rightarrow ((s, t_1) \in r \land (s, t_2) \in r \Rightarrow t_1 = t_2)$$

  - That is, in a ***function***, it is forbidden for a member of S to map to more than one members of T.
  - Equivalently, in a ***function***, two distinct members of T cannot be mapped by the same member of S.
- e.g., Say $S = \{1, 2, 3\}$ and $T = \{a, b\}$, which of the following ***relations*** satisfy the above ***functional property***?
  - $S \times T$ [No]
    Witness 1: $(1, a), (1, b)$; ***Witness*** 2: $(2, a), (2, b)$; ***Witness*** 3: $(3, a), (3, b)$.
  - $(S \times T) \setminus \{(x, y) \mid (x, y) \in S \times T \land x = 1\}$ [No]
    Witness 1: $(2, a), (2, b)$; ***Witness*** 2: $(3, a), (3, b)$
  - $\{(1, a), (2, b), (3, a)\}$ [Yes]
  - $\{(1, a), (2, b)\}$ [Yes]

Functions (2.1): Total vs. Partial

Given a **relation** $r \in S \leftrightarrow T$

- r is a ***partial function*** if it satisfies the ***functional property***:

$$\boxed{r \in S \nrightarrow T} \iff (\texttt{isFunctional(r)} \wedge \text{dom}(r) \subseteq S)$$

Remark. $r \in S \nrightarrow T$ means there **may (or may not) be** $s \in S$ s.t. $r(s)$ is ***undefined*** (i.e., $r[\{s\}] = \varnothing$).

  - e.g., $\{\ \{(\mathbf{2}, a), (\mathbf{1}, b)\}, \{(\mathbf{2}, a), (\mathbf{3}, a), (\mathbf{1}, b)\}\ \} \subseteq \{1, 2, 3\} \nrightarrow \{a, b\}$

- r is a ***total function*** if there is a mapping for each $s \in S$:

$$\boxed{r \in S \rightarrow T} \iff (\texttt{isFunctional(r)} \wedge \text{dom}(r) = S)$$

Remark. $r \in S \rightarrow T$ implies $r \in S \nrightarrow T$, but not vice versa. Why?

  - e.g., $\{(\mathbf{2}, a), (\mathbf{3}, a), (\mathbf{1}, b)\} \in \{1, 2, 3\} \rightarrow \{a, b\}$
  - e.g., $\{(\mathbf{2}, a), (\mathbf{1}, b)\} \notin \{1, 2, 3\} \rightarrow \{a, b\}$

Functions (2.2): Relation Image vs. Function Application

- Recall: A ***function*** is a ***relation***, but a ***relation*** is not necessarily a ***function***.
- Say we have a ***partial function*** $f \in \{1,2,3\} \nrightarrow \{a,b\}$:

$$f = \{(\mathbf{3}, a), (\mathbf{1}, b)\}$$

  - With f wearing the ***relation*** hat, we can invoke *relational images* :

$$\begin{array}{lcl} f[\{3\}] & = & \{a\} \\ f[\{1\}] & = & \{b\} \\ f[\{2\}] & = & \varnothing \end{array}$$

  Remark. $\Rightarrow |f[\{v\}]| \leq 1 \because$
    - each member in $\mathrm{dom}(f)$ is mapped to at most one member in $\mathrm{ran}(f)$
    - each input set $\{v\}$ is a **singleton** set
  - With f wearing the ***function*** hat, we can invoke *functional applications* :

$$\begin{array}{lcl} f(3) & = & a \\ f(1) & = & b \\ f(2) & \texttt{is} & \textit{undefined} \end{array}$$

LTL Semantics: Example of LTS

- We may visual a transition system $\mathbb{M}$ using a ***directed graph***:
  - Nodes/Vertices denote ***states***.
  - Edges/Arcs denote ***transitions***.
- **Exercises** Consider the system with a counter c with the following assumption:

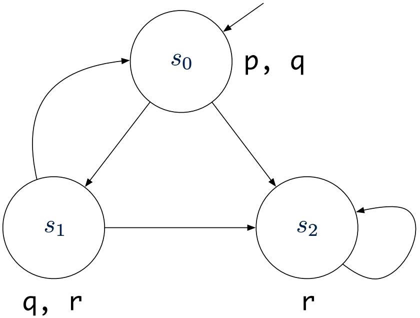
$$0 \leq c \leq 3$$

  Say c is initialized 0 and may be incremented (via a transition *inc*, enabled when $c < 3$) or decremented (via a transition *dec*, enabled when $c > 0$).

  - **Draw** a ***state graph*** of this system.
  - **Formulate** the state graph as an ***LTS*** (via a triple $(S, \longrightarrow, L)$).

    Assume: Set P of atoms is: $\{ c \geq 1, c \leq 1 \}$

LTL Semantics: More Example of LTS



$\mathbb{M} = (S, \longrightarrow, L)$:

- $S = \{s_0, s_1, s_2\}$
- $\longrightarrow = \{(s_0, s_1), (s_0, s_2), (s_1, s_0), (s_1, s_2), (s_2, s_2)\}$
- $L = \{(s_0, \{p, q\}), (s_1, \{q, r\}), (s_2, \{r\})\}$

LTL Semantics: Paths

Definition. A ***path*** in a model $\mathbb{M} = (S, \longrightarrow, L)$ is an ***infinite sequence of states*** $s_i \in S$, where $i \geq 1$, such that $s_i \longrightarrow s_{i+1}$.

- We write the path, starting at the ***initial state*** s_1, as

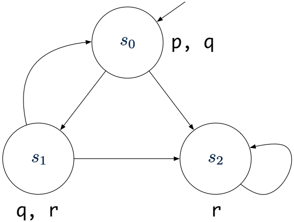
$$s_1 \longrightarrow s_2 \longrightarrow \dots$$

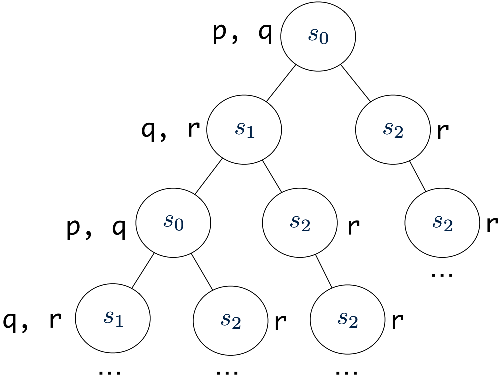
- **Note.** s_1 in the above path pattern denotes the first, initial state of the path, but in general, the actual name of the initial state may cause confusion, e.g., s_0.
- A ***path*** $\pi = s_1 \longrightarrow s_2 \longrightarrow \dots$ represents a ***possible future*** of $\mathbb{M}$.
- We write $\pi^{\mathbf{i}}$ for the ***suffix*** of path π: a path starting from state $s_{\mathbf{i}}$.
  e.g., $\pi^3 = s_3 \longrightarrow s_4 \longrightarrow \dots$
  e.g., $\pi^1 = \pi$

LTL Semantics: All Possible Paths

Given a state *s*, we represent **all** possible ***(computation) paths*** as a ***computation tree*** by *unwinding* the transitions.

e.g.





LTL Semantics: Path Satisfaction (1)

Definition. Given a ***model*** $\mathbb{M} = (S, \longrightarrow, L)$ and a ***path*** $\pi = s_1 \longrightarrow \ldots$ in $\mathbb{M}$, whether or not path π satisfies an ***LTL formula*** is defined by the *satisfaction relation* $\models$ as follows:

$$
\begin{array}{lllll}
\pi & \models & p & \iff & p \in L(s_1) \\
\pi & \models & \top & & \\
\pi & \not\models & \bot & & \\
\pi & \models & \neg\phi & \iff & \neg(\pi \models \phi) \\
\pi & \models & \phi_1 \wedge \phi_2 & \iff & \pi \models \phi_1 \wedge \pi \models \phi_2 \\
\pi & \models & \phi_1 \vee \phi_2 & \iff & \pi \models \phi_1 \vee \pi \models \phi_2 \\
\pi & \models & \phi_1 \Rightarrow \phi_2 & \iff & \pi \models \phi_1 \Rightarrow \pi \models \phi_2
\end{array}
$$

Tips. To evaluate $\pi \models \phi_1 \wedge \phi_2$ (and similarly for $\neg$, $\vee$, $\Rightarrow$):

- If ϕ_1 and ϕ_2 are sophisticated, decompose it to $\pi \models \phi_1$ and $\pi \models \phi_2$.
- Otherwise, directly evaluate $\phi_1 \wedge \phi_2$ on s_1.

LTL Semantics: Path Satisfaction (2.1)

Definition. Given a ***model*** $\mathbb{M} = (S, \longrightarrow, L)$ and a ***path*** $\pi = s_1 \longrightarrow \ldots$ in $\mathbb{M}$, whether or not path π satisfies an ***LTL formula*** is defined by the *satisfaction relation* $\vDash$ as follows:

$$
\begin{array}{lllll}
\pi & \vDash & \mathbf{X}\,\phi & \iff & \pi^2 \vDash \phi \\
\pi & \vDash & \mathbf{G}\,\phi & \iff & (\ \forall i \bullet i \geq 1 \Rightarrow \pi^i \vDash \phi\ ) \\
\pi & \vDash & \mathbf{F}\,\phi & \iff & (\ \exists i \bullet i \geq 1 \wedge \pi^i \vDash \phi\ )
\end{array}
$$

LTL Semantics: Model Satisfaction (1)

- **Definition**. Given:
  - a model $\mathbb{M} = (S, \longrightarrow, L)$
  - a state $s \in S$
  - an LTL formula ϕ

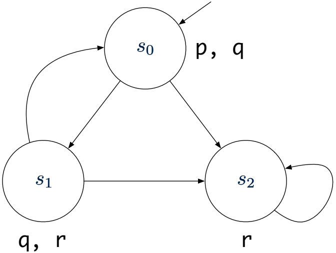
  $\boxed{\mathbb{M}, s \models \phi}$ if and only if for **every** path π of $\mathbb{M}$ starting at s, $\pi \models \phi$.

$$\mathbb{M}, s \models \phi \iff (\ \forall \pi \bullet (\ \pi = s \longrightarrow \dots\ ) \Rightarrow \pi \models \phi\ )$$

- When the model $\mathbb{M}$ is clear from the context, we write: $\boxed{s \models \phi}$.

LTL Semantics: Model Satisfaction (2.1)

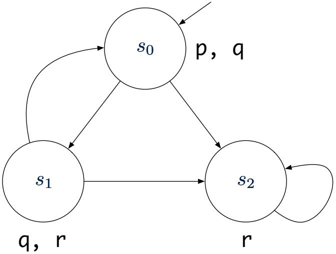
Consider the following system model:



- $s_0 \models \top$ [*true*]
- $s_0 \not\models \bot$ [*true*]
- $s_0 \models p \land q$ [*true*]
- $s_0 \models r$ [*false*]

LTL Semantics: Model Satisfaction (2.2)

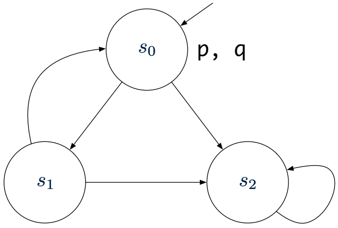
Consider the following system model:



- $s_0 \models \mathbf{X}\, q$ [*false*]
  Witness Path: $s_0 \longrightarrow \boxed{s_2} \longrightarrow s_2 \cdots \not\models \mathbf{X}\, q$
- $s_0 \models \mathbf{X}\, r$ [*true*]
- $s_0 \models \mathbf{X}(q \wedge r)$ [*false*]
  Witness Path: $s_0 \longrightarrow \boxed{s_2} \longrightarrow s_2 \cdots \not\models \mathbf{X}(q \wedge r)$
- $s_0 \models \mathbf{X}(q \Rightarrow r)$ [*true*]

LTL Semantics: Model Satisfaction (2.3)

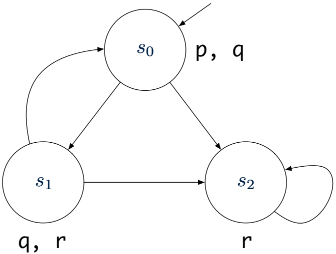
Consider the following system model:



- $s_0 \models \mathbf{G}\,\neg(p \land r)$ [*true*]
  $s \models \mathbf{G}\,\phi \iff \phi$ holds on all ***reachable*** states from s.
- $s_0 \models \mathbf{G}\,r$ [*false*]
  Witness Path: $\boxed{s_0} \longrightarrow s_2 \longrightarrow s_2 \cdots \not\models \mathbf{G}\,r$
- $s_2 \models \mathbf{G}\,r$ [*true*]

LTL Semantics: Model Satisfaction (2.4)

Consider the following system model:



- $s_0 \models \mathbf{F} \neg(p \wedge r)$ [*true*]
- $s_0 \models \mathbf{F}\, r$ [*true*]
- $s_0 \models \mathbf{F}(q \wedge r)$ [*false*]
  - Is is the case that $q \wedge r$ is eventually satisfied on every path?
  - No. Witness Path: $s_0 \longrightarrow s_2 \longrightarrow s_2 \longrightarrow \dots$
- $s_2 \models \mathbf{F}\, r$ [*true*]

LTL Semantics: Nested G and F (1)

Given a model $\mathbb{M} = (S, \longrightarrow, L)$ and a state $s \in S$:

$s \models \boldsymbol{FG}\phi$ means that:

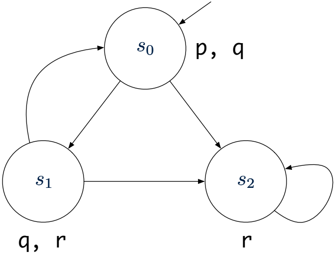
- **Each** path starting with s is such that **eventually**, ϕ holds **continuously**.
- For **all** paths π starting with s (i.e., $\pi = s \longrightarrow l \dots$):

$$\exists i \bullet i \geq 1 \land \left(\forall j \bullet j \geq i \Rightarrow \pi^i \models \phi\right)$$

- **Q.** How to ***prove*** and ***disprove*** the above formula pattern?
- **Hint.** Structure of pattern: $\forall \pi \bullet \dots \Rightarrow (\exists i \bullet \dots \land (\forall j \bullet \dots \Rightarrow \phi))$

LTL Semantics: Model Satisfaction (2.5.1)

Consider the following system model:



- $s_0 \models \mathbf{F}\,\mathbf{G}\, r$ [*false*]
  Witness: $s_0 \longrightarrow s_1 \longrightarrow s_0 \longrightarrow s_1 \longrightarrow \dots$
- $s_0 \models \mathbf{F}\,\mathbf{G}(p \vee q)$ [*false*]
  Witness: $s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow s_2 \longrightarrow \dots$
- $s_0 \models \mathbf{F}\,\mathbf{G}(p \vee r)$ [*true*]
  Justification: All possible paths from s_0 involve s_0, s_1, and s_2, all of which satisfying $p \vee r$.

LTL Semantics: Nested G and F (2)

Given a model $\mathbb{M} = (S, \longrightarrow, L)$ and a state $s \in S$:

$s \models \boldsymbol{F}\phi_1 \Rightarrow \boldsymbol{F}\boldsymbol{G}\phi_2$ means that:

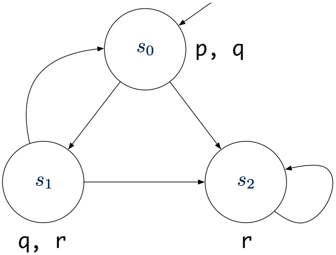
- **Each** path π starting with s is such that if ϕ_1 **eventually** holds on π, then ϕ_2 **eventually** holds **continuously** on the same π.

$$\forall \pi \bullet \pi = s \longrightarrow \ldots \Rightarrow \left(\begin{array}{l} (\exists i \bullet i \geq 1 \wedge \pi^i \models \phi_1) \\ \Rightarrow \\ (\exists i \bullet i \geq 1 \wedge (\forall j \bullet j \geq i \Rightarrow \pi^i \models \phi_2)) \end{array} \right)$$

- **Q.** How to ***disprove*** the above formula pattern?
- **A.** Find a witness path π which makes the "inner" implication ***false***.

LTL Semantics: Model Satisfaction (2.5.2)

Consider the following system model:



- $s_0 \models \mathbf{F}(\neg q \land r) \Rightarrow \mathbf{F}\,\mathbf{G}\,r$ [*true*]

  Justification:
  - $s_0 \longrightarrow s_1 \longrightarrow s_0 \longrightarrow \dots$ never satisfies $\neg q \land r$.
  - $s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow s_2 \longrightarrow \dots$ eventually satisfies $\neg q \land r$ continuously.
  - $s_0 \longrightarrow s_2 \longrightarrow s_2 \longrightarrow \dots$ eventually satisfies $\neg q \land r$ continuously.
- $s_0 \models \mathbf{F}(\neg q \lor r) \Rightarrow \mathbf{F}\,\mathbf{G}\,r$ [*false*]

  Witness: $s_0 \longrightarrow s_1 \longrightarrow s_0 \longrightarrow \dots$ eventually satisfies $\neg q \lor r$, but there is no point in this path where r holds continuously.

LTL Semantics: Nested G and F (3)

Given a model $\mathbb{M} = (S, \longrightarrow, L)$ and a state $s \in S$:

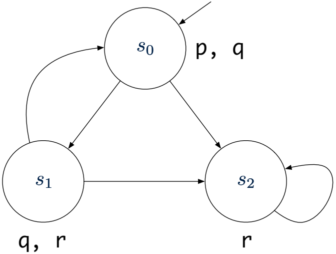
- $s \models \boldsymbol{GF}\phi$ means that:
  - **Each** path starting with s is such that **continuously**, ϕ holds **eventually**.
    $\Rightarrow$ ϕ holds ***infinitely often***!
  - For **all** paths π starting with s (i.e., $\pi = s \longrightarrow I \dots$):

$$\forall i \bullet i \geq 1 \Rightarrow \left(\exists j \bullet j \geq i \wedge \pi^i \models \phi\right)$$

  - **Q.** How to ***prove*** and ***disprove*** the above formula pattern?
  - **Hint.** Structure of pattern: $\forall \pi \bullet \dots \Rightarrow (\forall i \bullet \dots \Rightarrow (\exists j \bullet \cdots \wedge \phi))$

LTL Semantics: Model Satisfaction (2.6)

Consider the following system model:



- $s_0 \models \mathbf{G}\,\mathbf{F}\,p$ [*false*]
  Witness: In $s_0 \longrightarrow s_2 \longrightarrow \dots$, p is not satisfied ***infinitely often***.
- $s_0 \models \mathbf{G}\,\mathbf{F}(\ p \lor r\ )$ [*true*]
- $s_0 \models \mathbf{G}\,\mathbf{F}\,p \Rightarrow \mathbf{G}\,\mathbf{F}\,r$ [*true*]
  Hint: Consider paths making the antecedent $\mathbf{G}\,\mathbf{F}\,p$ ***true***.
- $s_0 \models \mathbf{G}\,\mathbf{F}\,r \Rightarrow \mathbf{G}\,\mathbf{F}\,p$ [*false*]
  Witness: $s_0 \longrightarrow s_2 \longrightarrow \dots$ [Why?]

LTL Semantics: Path Satisfaction (2.2)

Definition. Given a ***model*** $\mathbb{M} = (S, \longrightarrow, L)$ and a ***path*** $\pi = s_1 \longrightarrow \ldots$ in $\mathbb{M}$, whether or not path π satisfies an ***LTL formula*** is defined by the *satisfaction relation* $\models$ as follows:

$$\pi \models \phi_1 \, \mathbf{U} \, \phi_2 \iff \left(\exists i \bullet i \geq 1 \land \left(\begin{array}{l} \pi^i \models \phi_2 \\ \land \\ (\forall j \bullet 1 \leq j \leq i-1 \Rightarrow \pi^j \models \phi_1) \end{array} \right) \right)$$

$$\pi \models \phi_1 \, \mathbf{W} \, \phi_2 \iff \left(\begin{array}{ll} & \phi_1 \, \mathbf{U} \, \phi_2 \\ \lor & (\forall k \bullet k \geq 1 \Rightarrow \pi^k \models \phi_1) \end{array} \right)$$

$$\pi \models \phi_1 \, \mathbf{R} \, \phi_2 \iff \left(\begin{array}{ll} & \left(\exists i \bullet i \geq 1 \land \left(\begin{array}{l} \pi^i \models \phi_1 \\ \land \\ (\forall j \bullet 1 \leq j \leq i \Rightarrow \pi^j \models \phi_2) \end{array} \right) \right) \\ \lor & (\forall k \bullet k \geq 1 \Rightarrow \pi^k \models \phi_2) \end{array} \right)$$

LTL Semantics: Recall Model Satisfaction

- **Definition**. Given:
  - a model $\mathbb{M} = (S, \longrightarrow, L)$
  - a state $s \in S$
  - an LTL formula ϕ

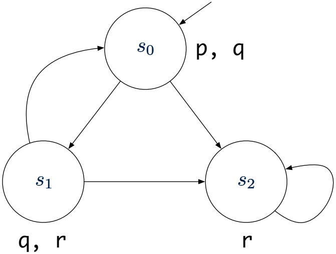
  $\boxed{\mathbb{M}, s \models \phi}$ if and only if for **every** path π of $\mathbb{M}$ starting at s, $\pi \models \phi$.

$$\mathbb{M}, s \models \phi \iff (\ \forall \pi \bullet (\ \pi = s \longrightarrow \dots\ ) \Rightarrow \pi \models \phi\ )$$

- When the model $\mathbb{M}$ is clear from the context, we write: $\boxed{s \models \phi}$.

LTL Semantics: Model Satisfaction (3.1)

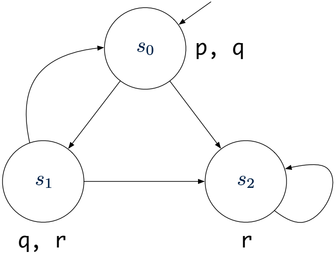
Consider the following system model:



- $s_0 \models p \, \mathbf{U} \, r$ [*true*]
  s_0 (satisfying p) branches out to s_1 or s_2 (both both satisfying r).
- $s_0 \models p \, \mathbf{W} \, r$ [*true*]
  $\phi_1 \, \mathbf{U} \, \phi_2 \Rightarrow \phi_1 \, \mathbf{W} \, \phi_2$
- $s_0 \models r \, \mathbf{R} \, p$ [*false*]
  Witness: Say $\pi = s_0 \longrightarrow s_1 \longrightarrow s_0 \longrightarrow s_1 \ldots$: $\pi \not\models p \land r$ and $\pi \not\models \mathbf{G} \, p$.

LTL Semantics: Model Satisfaction (3.2)

Consider the following system model:



- $s_0 \models (p \vee r)\,\mathbf{U}(p \wedge r)$ [*false*]
  Witness: In $s_0 \longrightarrow s_1 \longrightarrow s_0 \longrightarrow s_1 \ldots$, $p \wedge r$ never holds.
- $s_0 \models (p \vee r)\,\mathbf{W}(p \wedge r)$ [*true*]
  It is the case that: $s_0 \models \mathbf{G}(p \vee r)$.
- $s_0 \models (p \wedge r)\,\mathbf{R}(p \vee r)$ [*true*]
  It is the case that: $s_0 \models \mathbf{G}(p \vee r)$.

Clarification on the “Until” Connective

- $\phi_1 \, \mathbf{U} \, \phi_2$ requires that:
  - ϕ_2 must eventually become ***true***.
  - Before ϕ_2 becomes ***true***, ϕ_1 must hold.
- **Exercise**. Say:
  - Atom t: I was 22.
  - Atom s: I smoke.

  Formulate “I had smoked until I was 22” using LTL.
  - $s \, \mathbf{U} \, t$ [***inaccurate***]
  - $\phi_1 \, \mathbf{U} \, \phi_2$ does not insist $\boxed{\neg \phi_1}$ after $\boxed{\phi_2}$ eventually becomes ***true***.
  - “I smoked both before and after I was 22” satisfies $s \, \mathbf{U} \, t$.
  - Solution? [$s \; \boldsymbol{U} \; (\; t \wedge (\boldsymbol{G} \neg s) \;)$]

Formulating English as LTL Formulas (1)

- Assume the following atomic propositions:
  busy, *requested*, *acknowledged*, *enabled*, *floor*2, *floor*5, *directionUp*, *buttonPresssed*5.
- It is impossible to reach a state where the system is started but not ready.
  - $\mathbf{G}\,\neg(started \wedge \neg ready)$ [$\neg(\ \mathbf{F}(started \wedge \neg ready)\ )$]
- Whenever a request is made, it will be eventually be acknowledged.
  - $\mathbf{G}(\ requested \Rightarrow \mathbf{F}\, acknowledged\ )$
- A certain process will always be enabled.
  - $\mathbf{G}\, enabled$
- An upwards travelling lift at the second floor does not change its direction when it has passengers wishing to go to the fifth floor.
  -
$$\mathbf{G}\left(\begin{array}{c} floor2 \wedge directionUp \wedge buttonPresssed5 \\ \Rightarrow (\ directionUp\, \mathbf{U}\, floor5\ ) \end{array}\right)$$
  - Is it ok to change from **U** to **W**?

Formulating English as LTL Formulas (2)

Assume the following atomic propositions:

$$requested, waiting, granted, noOneInCS$$

Whenever a process makes a request, it starts waiting. As soon as no other process is in the critical section, the process is granted access to the critical section.

$$\mathbf{G}\ (requested \Rightarrow (noOneInCS\ \mathbf{R}\ waiting))$$

Q. Does the above formulation guarantee ***no starvation***?

Hint. Check the formal definition of **R**.

Formulating English as LTL Formulas (3)

Assume the following atomic propositions:

$$degReqFullfilled, allowedForGraduation$$

Until a student fullfils all their degree requirements, their academic staus remains “not allowed for graduation”. The change of status, when qualified, may not be instantaneous to account for human/manual processing.

$$\neg allowedForGraduation \ \mathbf{W} \ (degReqFulfilled \land \mathbf{G} \ allowedForGraduation)$$

Q. Does the above formulation account for situations where a student never fulfills their degree requirements?

Hint. Check the formal definition of **W**.

Index (1)

Index (2)

Index (3)

Index (4)

Index (5)

Program Verification

Readings: Chapter 4 of LICS2

EECS4315 Z:
Mission-Critical Systems
Winter 2025

CHEN-WEI WANG

Learning Objectives

1. Motivating Examples: ***Program Correctness***
2. ***Hoare Triple***
3. ***Weakest Precondition*** (*wp*)
4. Rules of *wp* ***Calculus***
5. Contract of Loops (***invariant*** vs. ***variant***)
6. ***Correctness Proofs*** of Loops

Assertions: Weak vs. Strong

- Describe each assertion as ***a set of satisfying value***.

  $x > 3$ has satisfying values $\{ x \mid x > 3 \} = \{ 4, 5, 6, 7, \ldots \}$
  $x > 4$ has satisfying values $\{ x \mid x > 4 \} = \{ 5, 6, 7, \ldots \}$
- An assertion p is ***stronger*** than an assertion q if p's set of satisfying values is a subset of q's set of satisfying values.
  - Logically speaking, p being stronger than q (or, q being weaker than p) means $p \Rightarrow q$.
  - e.g., $x > 4 \Rightarrow x > 3$
- What's the weakest assertion? [**TRUE**]
- What's the strongest assertion? [**FALSE**]
- In ***System Specification***:
  - A weaker ***invariant*** has more acceptable object states
    e.g., $balance > 0$ vs. $balance > 100$ as an invariant for `ACCOUNT`
  - A weaker ***precondition*** has more acceptable input values
  - A weaker ***postcondition*** has more acceptable output values

Assertions: Preconditions

Given *preconditions* P_1 and P_2, we say that

P_2 ***requires less*** than P_1 if

P_2 is ***less strict*** on (thus ***allowing more***) inputs than P_1 does.

$$\{ x \mid P_1(x) \} \subseteq \{ x \mid P_2(x) \}$$

More concisely:

$$P_1 \Rightarrow P_2$$

e.g., For command `withdraw(amount: INTEGER)`,

$P_2 : amount \geq 0$ ***requires less*** than $P_1 : amount > 0$

What is the *precondition* that ***requires the least***? [***true***]

Assertions: Postconditions

Given ***postconditions*** or ***invariants*** Q_1 and Q_2, we say that

Q_2 ***ensures more*** than Q_1 if

Q_2 is ***stricter*** on (thus ***allowing less***) outputs than Q_1 does.

$$\{\ x \mid Q_2(x)\ \} \subseteq \{\ x \mid Q_1(x)\ \}$$

More concisely:

$$Q_2 \Rightarrow Q_1$$

e.g., For query `q(i: INTEGER): BOOLEAN`,

$Q_2 : \textbf{Result} = (i > 0) \wedge (i \textbf{ mod } 2 = 0)$ ***ensures more*** than

$Q_1 : \textbf{Result} = (i > 0) \vee (i \textbf{ mod } 2 = 0)$

What is the ***postcondition*** that ***ensures the most***? [***false***]

Motivating Examples (1)

Is this algorithm correct?

```
--algorithm increment_by_9 {
  variable i;
  {
    (* precondition *)
    assert  i > 3

    (* implementation *)
    i := i + 9;

    (* postcondition *)
    assert  i > 13
  }
}
```

Q: Is $i > 3$ is too weak or too strong?

A: Too weak

$\because$ assertion $i > 3$ allows value 4 which would fail postcondition.

Motivating Examples (2)

Is this algorithm correct?

```
--algorithm increment_by_9 {
  variable i;
  {
    (* precondition *)
    assert  i > 5

    (* implementation *)
    i := i + 9;

    (* postcondition *)
    assert  i > 13
  }
}
```

Q: Is $i > 5$ too weak or too strong?

A: Maybe too strong

$\because$ assertion $i > 5$ disallows 5 which would not fail postcondition.

Whether 5 should be allowed depends on the requirements.

Software Correctness

- Correctness is a *relative* notion:

  consistency of *implementation* with respect to *specification*.

  $\Rightarrow$ This assumes there is a specification!
- We introduce a formal and systematic way for formalizing a program **S** and its *specification* (pre-condition ***Q*** and post-condition ***R***) as a *Boolean predicate*: $\{Q\}\ \mathbf{S}\ \{R\}$
  - e.g., $\{i > 3\}$ `i := i + 9` $\{i > 13\}$
  - e.g., $\{i > 5\}$ `i := i + 9` $\{i > 13\}$
  - If $\{Q\}\ \mathbf{S}\ \{R\}$ **can** be proved **TRUE**, then the **S** is correct.

    e.g., $\{i > 5\}$ `i := i + 9` $\{i > 13\}$ can be proved TRUE.
  - If $\{Q\}\ \mathbf{S}\ \{R\}$ **cannot** be proved **TRUE**, then the **S** is incorrect.

    e.g., $\{i > 3\}$ `i := i + 9` $\{i > 13\}$ cannot be proved TRUE.

Hoare Logic

- Consider a program **S** with precondition ***Q*** and postcondition ***R***.
  - {***Q***} S {***R***} is a *correctness predicate* for program **S**
  - {***Q***} S {***R***} is TRUE if program **S** starts executing in a state satisfying the precondition ***Q***, and then:
    (a) The program **S** terminates.
    (b) Given that program **S** terminates, then it terminates in a state satisfying the postcondition ***R***.
- Separation of concerns

  (a) requires a proof of *termination*.

  (b) requires a proof of ***partial** correctness*.

  Proofs of (a) + (b) imply ***total** correctness*.

Hoare Logic and Software Correctness

Consider the ***contract/specification*** view of an algorithm f (whose body of implementation is **S**) as a Hoare Triple:

$$\{Q\}\ \texttt{S}\ \{R\}$$

Q is the *precondition* of f.
$\texttt{S}$ is the implementation of f.
R is the *postcondition* of f.

- $\{\textit{true}\}\ \texttt{S}\ \{R\}$
  All input values are valid [Most-user friendly]
- $\{\textit{false}\}\ \texttt{S}\ \{R\}$
  All input values are invalid [Most useless for clients]
- $\{Q\}\ \texttt{S}\ \{\textit{true}\}$
  All output values are valid [Most risky for clients; Easiest for suppliers]
- $\{Q\}\ \texttt{S}\ \{\textit{false}\}$
  All output values are invalid [Most challenging coding task]
- $\{\textit{true}\}\ \texttt{S}\ \{\textit{true}\}$
  All inputs/outputs are valid (No specification) [Least informative]

Proof of Hoare Triple using *wp*

$$\{Q\}\ \mathtt{S}\ \{R\} \equiv Q \Rightarrow wp(S, R)$$

- $wp(S, R)$ is the *weakest precondition for S to establish R*.
  - If $Q \Rightarrow wp(S, R)$, then **any** execution started in a state satisfying Q will terminate in a state **satisfying** R.
  - If $Q \not\Rightarrow wp(S, R)$, then **some** execution started in a state satisfying Q will terminate in a state **violating** R.
- S can be:
  - Assignments [`x := y`]
  - Alternations [`if ... then ... else ... end`]
  - Sequential compositions [S_1 ; S_2]
  - Loops [`while(...) { ... }`]
- We will learn how to calculate the *wp* for the above programming constructs.

Denoting Pre- and Post-State Values

- In the *postcondition*, for a program variable x:
  - We write $\boxed{x_0}$ to denote its ***pre-state (old)*** value.
  - We write $\boxed{x}$ to denote its ***post-state (new)*** value.

  Implicitly, in the *precondition*, all program variables have their ***pre-state*** values.

  e.g., $\{b_0 > a\}$ `b := b - a` $\{b = b_0 - a\}$
- Notice that:
  - We may choose to write "b" rather than "b_0" in preconditions
    ∵ All variables are pre-state values in preconditions
  - We don't write "b_0" in program
    ∵ there might be ***multiple intermediate values*** of a variable due to ***sequential composition***

wp Rule: Assignments (1)

$$wp(\texttt{x := e}, R) = R[x := e]$$

$R[x := e]$ means to substitute all ***free occurrences*** of variable x in postcondition R by expression e.

wp Rule: Assignments (2)

Recall:

$$\{Q\}\ \mathtt{S}\ \{R\}\ \equiv\ Q \Rightarrow wp(S, R)$$

How do we prove $\{Q\}\ \mathtt{x\ :=\ e}\ \{R\}$?

$$\{Q\}\ \mathtt{x\ :=\ e}\ \{R\} \iff Q \Rightarrow \underbrace{R[x := e]}_{wp(\mathtt{x\ :=\ e},\ R)}$$

wp Rule: Assignments (3) Exercise

What is the weakest precondition for a program `x := x + 1` to establish the postcondition $x > x_0$?

$$\{??\}\ \texttt{x := x + 1}\ \{x > x_0\}$$

For the above Hoare triple to be ***TRUE***, it must be that $?? \Rightarrow wp(\texttt{x := x + 1}, x > x_0)$.

$$
\begin{array}{cl}
 & wp(\texttt{x := x + 1}, x > x_0) \\
= & \{\ \textit{Rule of } wp\textit{: Assignments}\ \} \\
 & x > x_0[x := x_0 + 1] \\
= & \{\ \textit{Replacing } x \textit{ by } x_0 + 1\ \} \\
 & x_0 + 1 > x_0 \\
= & \{\ 1 > 0 \textit{ always true}\ \} \\
 & \textit{True}
\end{array}
$$

Any precondition is OK. ***False*** is valid but not useful.

wp Rule: Assignments (4) Exercise

What is the weakest precondition for a program `x := x + 1` to establish the postcondition $x = 23$?

$$\{??\}\ \texttt{x := x + 1}\ \{x = 23\}$$

For the above Hoare triple to be ***TRUE***, it must be that $?? \Rightarrow wp(\texttt{x := x + 1}, x = 23)$.

$$
\begin{array}{ll}
 & wp(\texttt{x := x + 1}, x = 23) \\
= & \{\ \textit{Rule of } \textbf{\textit{wp}}\textbf{:}\ \textit{Assignments}\ \} \\
 & x = 23[x := x_0 + 1] \\
= & \{\ \textit{Replacing } x \textit{ by } x_0 + 1\ \} \\
 & x_0 + 1 = 23 \\
= & \{\ \textit{arithmetic}\ \} \\
 & x_0 = 22
\end{array}
$$

Any precondition weaker than $x = 22$ is not OK.

wp Rule: Assignments (4) Revisit

Given $\{??\} n := n + 9 \{n > 13\}$:

- $\boxed{n > 4}$ is the *weakest precondition (wp)* for the given implementation (n := n + 9) to start and establish the postcondition ($n > 13$).
- Any precondition that is ***equal to or stronger than*** the *wp* ($n > 4$) will result in a correct program.

  e.g., $\{n > 5\} n := n + 9 \{n > 13\}$ can be proved **TRUE**.
- Any precondition that is ***weaker than*** the *wp* ($n > 4$) will result in an incorrect program.

  e.g., $\{n > 3\} n := n + 9 \{n > 13\}$ cannot be proved **TRUE**.

  Counterexample: $n = 4$ satisfies precondition $n > 3$ but the output $n = 13$ fails postcondition $n > 13$.

wp Rule: Alternations (1)

$$wp(\texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ end}, \boldsymbol{R}) = \begin{pmatrix} B \Rightarrow wp(S_1, \boldsymbol{R}) \\ \wedge \\ \neg B \Rightarrow wp(S_2, \boldsymbol{R}) \end{pmatrix}$$

The *wp* of an alternation is such that ***all branches*** are able to establish the postcondition ***R***.

wp Rule: Alternations (2)

Recall: $\{Q\}\ S\ \{R\} \equiv Q \Rightarrow wp(S, R)$

How do we prove that $\{Q\}$ **if** B **then** S_1 **else** S_2 **end** $\{R\}$?

$\{Q\}$
if B **then**
  $\{Q \wedge B\}\ S_1\ \{R\}$
else
  $\{Q \wedge \neg B\}\ S_2\ \{R\}$
end
$\{R\}$

$$\{Q\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end}\ \{R\}$$

$$\iff \begin{pmatrix} \{\ Q \wedge B\ \}\ S_1\ \{\ R\ \} \\ \wedge \\ \{\ Q \wedge \neg B\ \}\ S_2\ \{\ R\ \} \end{pmatrix} \iff \begin{pmatrix} (Q \wedge B) \Rightarrow wp(S_1, R) \\ \wedge \\ (Q \wedge \neg B) \Rightarrow wp(S_2, R) \end{pmatrix}$$

wp Rule: Alternations (3) Exercise

Is this program correct?

$\{x > 0 \wedge y > 0\}$
```
if x > y then
  bigger := x ; smaller := y
else
  bigger := y ; smaller := x
end
```
$\{bigger \geq smaller\}$

$$\left(\begin{array}{l} \{(x > 0 \wedge y > 0) \wedge (x > y)\} \\ \qquad \texttt{bigger := x ; smaller := y} \\ \{bigger \geq smaller\} \end{array}\right)$$
$$\wedge$$
$$\left(\begin{array}{l} \{(x > 0 \wedge y > 0) \wedge \neg(x > y)\} \\ \qquad \texttt{bigger := y ; smaller := x} \\ \{bigger \geq smaller\} \end{array}\right)$$

wp Rule: Sequential Composition (1)

$$wp(S_1 \; ; \; S_2, \boldsymbol{R}) = wp(S_1, wp(S_2, \boldsymbol{R}))$$

The *wp* of a sequential composition is such that the first phase establishes the *wp* for the second phase to establish the postcondition ***R***.

wp Rule: Sequential Composition (2)

Recall:

$$\{Q\}\ S\ \{R\} \equiv Q \Rightarrow wp(S, R)$$

How do we prove $\{Q\}\ S_1\ ;\ S_2\ \{R\}$?

$$\{Q\}\ S_1\ ;\ S_2\ \{R\} \iff Q \Rightarrow \underbrace{wp(S_1, wp(S_2, R))}_{wp(S_1\ ;\ S_2, R)}$$

wp Rule: Sequential Composition (3) Exercise

Is $\{\ \textit{True}\ \}$ `tmp := x; x := y; y := tmp` $\{\ x > y\ \}$ correct?

If and only if $\textit{True} \Rightarrow wp(\texttt{tmp := x ; x := y ; y := tmp}, x > y)$

$$
\begin{array}{ll}
 & wp(\texttt{tmp := x ; } \boxed{\texttt{x := y ; y := tmp}}, x > y) \\
= & \{\ wp \textit{ rule for seq. comp. }\} \\
 & wp(\texttt{tmp := x}, wp(\texttt{x := y ; } \boxed{\texttt{y := tmp}}, x > y)) \\
= & \{\ wp \textit{ rule for seq. comp. }\} \\
 & wp(\texttt{tmp := x}, wp(\texttt{x := y}, wp(\texttt{y := tmp}, x > \boxed{y}))) \\
= & \{\ wp \textit{ rule for assignment }\} \\
 & wp(\texttt{tmp := x}, wp(\texttt{x := y}, \boxed{x} > tmp)) \\
= & \{\ wp \textit{ rule for assignment }\} \\
 & wp(\texttt{tmp := x}, y > \boxed{tmp}) \\
= & \{\ wp \textit{ rule for assignment }\} \\
 & y > x
\end{array}
$$

$\because$ ***True*** $\Rightarrow y > x$ does not hold in general.

$\therefore$ The above program is not correct.

Loops

- A loop is a way to compute a certain result by ***successive approximations***.

  e.g. computing the maximum value of an array of integers

- Loops are needed and powerful
- But loops ***very hard*** to get right:
  - "off-by-one" error [partial correctness]
  - Not establishing the desired condition [partial correctness]
  - Improper handling of borderline cases [partial correctness]
  - Infinite loops [termination]

Correctness of Loops

How do we prove that the following loop is correct?

```
{Q}
S_init
while(B) {
  S_body
}
{R}
```

In case of C/Java/PlusCal, $\boxed{B}$ denotes the ***stay condition***.

- In TLA+ toolbox, there is **not** native, syntactic support for model-checking the ***total correctness*** of loops.
- Instead, we have to **manually** add assertions to encode:
  - LOOP INVARIANT [for establishing ***partial correctness***]
  - LOOP VARIANT [for ensuring ***termination***]

Specifying Loops

- Use of ***loop invariant (LI)*** and ***loop variant (LV)***.
  - ***LI***: Boolean expression for measuring/proving ***partial*** *correctness*
    - Typically a special case of the postcondition.
      e.g., Given postcondition "*Result is maximum of the array*":
      LI can be "*Result is maximum of the **part of array scanned so far***".
    - ***Established*** before the very first iteration.
    - ***Maintained*** TRUE after each iteration.
  - ***LV***: Integer expression for measuring/proving ***termination***
    - Denotes the "number of iterations remaining"
    - *Decreased* at the end of each subsequent iteration
    - Maintained *non-negative* at the end of each iteration.
    - As soon as value of ***LV*** reaches *zero*, meaning that no more iterations remaining, the loop must exit.
- Remember:

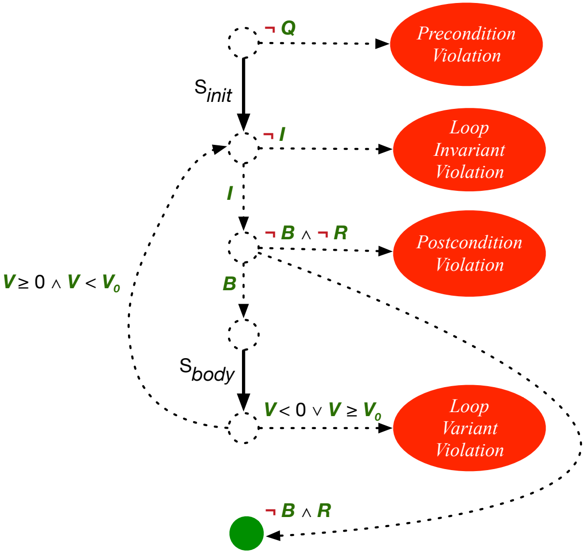
total *correctness* = ***partial*** *correctness* + ***termination***

Specifying Loops: Syntax

```
CONSTANT ... (* input list *)
I(var_list) == ...
V(var_list) == ...
--algorithm MyAlgorithm {
  variables ..., variant_pre = 0, variant_post = 0;
  {
    assert Q; (* Precondition *)
    S_init
    assert I(...); (* Is LI established? *)
    while( B ) {
      variant_pre := V(...);
      S_body
      variant_post := V(...);

      assert variant_post >= 0;
      assert variant_post < variant_pre;
      assert I(...); (* Is LI preserved? *)
    }
    assert R; (* Postcondition *)
  }
}
```

Specifying Loops: Runtime Checks (1)



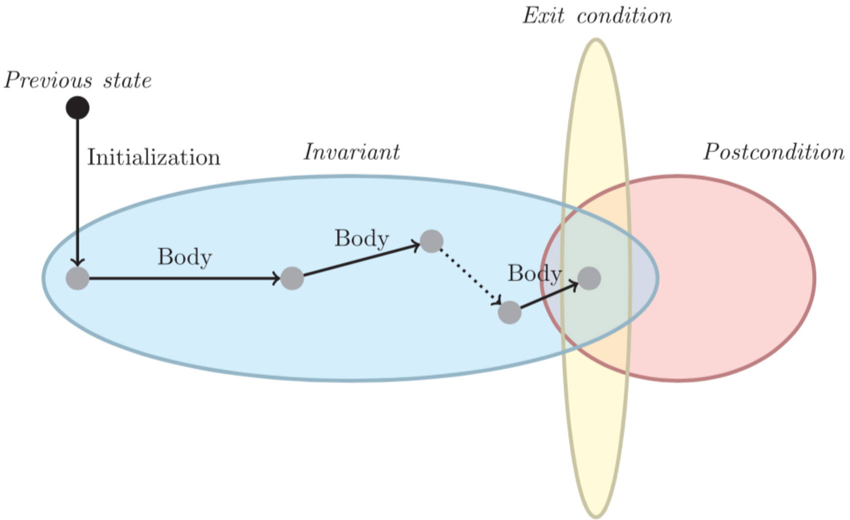
Specifying Loops: Runtime Checks (2)

```
1  I(i) == (1 <= i) /\ (i <= 6)
2  V(i) == 6 - i
3  --algorithm loop_invariant_test
4    variables i = 1, variant_pre = 0, variant_post = 0;
5    {
6      assert I(i);
7      while (i <= 5) {
8        variant_pre := V(i);
9        i := i + 1;
10       variant_post := V(i);
11       assert variant_post >= 0;
12       assert variant_post < variant_pre;
13       assert I(i);
14     } ;
15   }
```

L1: Change to `1 <= i /\ i <= 5` for a ***Loop Invariant Violation***.

L2: Change to `5 - i` for a ***Loop Variant Violation***.

Specifying Loops: Visualization



Digram Source: page 5 in *Loop Invariants: Analysis, Classification, and Examples*

Proving Correctness of Loops (1)

```
{Q}
S_init
assert I(...);
while( B ) {
  variant_pre := V(...);
  S_body
  variant_post := V(...);
  assert variant_post >= 0;
  assert variant_post < variant_pre;
  assert I(...);
}
{R}
```

- A loop is ***partially** correct* if:
  - Given precondition ***Q***, the initialization step S_{init} establishes ***LI*** I.
  - At the end of S_{body}, if not yet to exit, ***LI*** I is maintained.
  - If ready to exit and ***LI*** I maintained, postcondition ***R*** is established.
- A loop ***terminates*** if:
  - Given ***LI*** I, and not yet to exit, S_{body} maintains ***LV*** V as non-negative.
  - Given ***LI*** I, and not yet to exit, S_{body} decrements ***LV*** V.

Proving Correctness of Loops (2)

- A loop is ***partially*** *correct* if:
  - Given precondition ***Q***, the initialization step S_{init} establishes ***LI*** I.

    $$\{Q\}\ S_{init}\ \{I\}$$

  - At the end of S_{body}, if not yet to exit, ***LI*** I is maintained.

    $$\{I \land B\}\ S_{body}\ \{I\}$$

  - If ready to exit and ***LI*** I maintained, postcondition ***R*** is established.

    $$I \land \neg B \Rightarrow R$$

- A loop ***terminates*** if:
  - Given ***LI*** I, and not yet to exit, S_{body} maintains ***LV*** V as non-negative.

    $$\{I \land B\}\ S_{body}\ \{V \geq 0\}$$

  - Given ***LI*** I, and not yet to exit, S_{body} decrements ***LV*** V.

    $$\{I \land B\}\ S_{body}\ \{V < V_0\}$$

Index (1)

Index (2)

Index (3)