

Introduction



EECS4315 Z:
Mission-Critical Systems
Winter 2023

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?

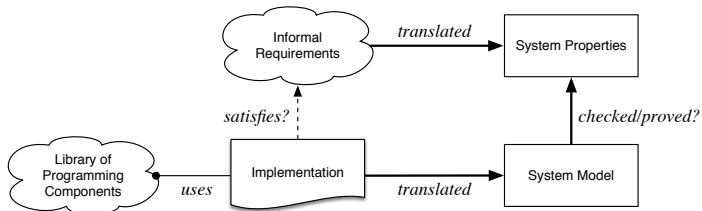
- **Critical:**
A task whose successful completion ensures the success of a larger, more complex operation.
e.g., Success of a pacemaker \Rightarrow Regulated heartbeats of a patient
 - **Safety:**
Being free from danger/injury to or loss of human lives.
 - **Mission:**
An operation or task assigned 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*"

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:

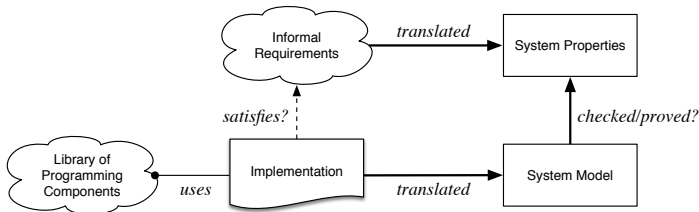
The use of **formal methods** is motivated by the expectation that, as in other engineering disciplines, performing appropriate **mathematical analyses** can contribute to establishing the **correctness** and **robustness** of a design.
- FMs, because of their mathematical basis, are capable of:
 - **Unambiguously** 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?



- o **Implementation** built via **reusable programming components**.
- o **Goal** : **Implementation Satisfies Intended Requirements**
- o 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).
- o 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 \nrightarrow 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**:
$$\langle m_0, m_1, \dots, m_i, m_j, \dots, m_n \rangle$$
 - The list starts by the **most abstract** model with least details.
 - A more **abstract** model m_i is said to be **refined by** its subsequent, more **concrete** model 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**.

- **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)

Learning Outcomes

What is a Safety-Critical System (SCS)?

Professional Engineers: Code of Ethics

Developing Safety-Critical Systems

Safety-Critical vs. Mission-Critical?

Using Formal Methods to for Certification

Verification: Building the Product Right?

Validation: Building the Right Product?

Catching Defects – When?

Model-Based Development in EECS3342

Model-Based Development in EECS4315

Index (2)

TLA+: An Industrial Strength Toolbox

Beyond this lecture ...

Review of Math



EECS4315 Z:
Mission-Critical Systems
Winter 2023

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	$\neg p$
<i>true</i>	<i>false</i>
<i>false</i>	<i>true</i>

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

p	q	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \iff q$	$p \equiv q$
<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>
<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>	<i>false</i>
<i>false</i>	<i>true</i>	<i>false</i>	<i>true</i>	<i>true</i>	<i>false</i>	<i>false</i>
<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>true</i>	<i>true</i>

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	$p \Rightarrow q$
<i>false</i>	<i>true</i>	<i>true</i>
<i>false</i>	<i>false</i>	<i>true</i>

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 $(\text{true} \Rightarrow \text{false}) \equiv \text{false}$.

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

- p **if** q [$q \Rightarrow p$]
- p **only if** q [$p \Rightarrow q$]
- p is **sufficient** for q
 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 $(\text{true} \Rightarrow \text{false}) \equiv \text{false}$.
- 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

$$\text{true} \Rightarrow p \equiv p$$

- **Theorem:** Zero of \Rightarrow

$$\text{false} \Rightarrow p \equiv \text{true}$$

- **Axiom:** De Morgan

$$\neg(p \wedge q) \equiv \neg p \vee \neg q$$

$$\neg(p \vee q) \equiv \neg p \wedge \neg q$$

- **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]
 - $\forall i, j \bullet i \in \mathbb{Z} \wedge j \in \mathbb{Z} \Rightarrow i < j \vee i > j$ [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: $\forall x \bullet (x \in \mathbb{Z} \wedge 1 \leq x \leq 10) \Rightarrow x > 0$.
All 10 integers between 1 and 10 are greater than 0.
- Prove or disprove: $\forall x \bullet (x \in \mathbb{Z} \wedge 1 \leq x \leq 10) \Rightarrow x > 1$.
Integer 1 (a witness/counterexample) in the range between 1 and 10 is *not* greater than 1.
- Prove or disprove: $\exists x \bullet (x \in \mathbb{Z} \wedge 1 \leq x \leq 10) \wedge x > 1$.
Integer 2 (a witness) in the range between 1 and 10 is greater than 1.
- Prove or disprove that $\exists x \bullet (x \in \mathbb{Z} \wedge 1 \leq x \leq 10) \wedge x > 10$?
All integers in the range between 1 and 10 are *not* greater than 10.

Predicate Logic (4): Switching Quantifications



Conversions between \forall and \exists :

$$\begin{aligned}(\forall X \bullet R \Rightarrow P) &\iff \neg(\exists X \bullet R \wedge \neg P) \\(\exists X \bullet R \wedge P) &\iff \neg(\forall X \bullet R \Rightarrow \neg P)\end{aligned}$$

Index (1)

Learning Outcomes of this Lecture

Propositional Logic (1)

Propositional Logic: Implication (1)

Propositional Logic: Implication (2)

Propositional Logic: Implication (3)

Propositional Logic (2)

Predicate Logic (1)

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

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

Predicate Logic (3): Exercises

Predicate Logic (4): Switching Quantifications

Verification by Model Checking

Readings: Chapter 3 of LICS2



EECS4315 Z:
Mission-Critical Systems
Winter 2023

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
- **ModelCheck-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 M
 - **Properties** specified as a set of formulas ϕ
 - **Decide** (automatically) that $M \models \phi$ [*decidable, algorithmic*]
 i.e., Traversal of M 's **state graph** shows that ϕ is satisfied.
- **Domain of Application**
 - Hardware vs. Software
 - Sequential vs. Concurrent
 - Reactive vs. Terminating
- **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 Temporal 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)

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

p denotes **atomic**, propositional statements

e.g., Printer $1tr2$ 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)

$\phi ::= \top$	[<i>true</i>]
\perp	[<i>false</i>]
p	[propositional atom]
$(\neg\phi)$	[logical negation]
$(\phi \wedge \phi)$	[logical conjunction]
$(\phi \vee \phi)$	[logical disjunction]
$(\phi \Rightarrow \phi)$	[logical implication]
$(\mathbf{X}\phi)$	[n e X t state]
$(\mathbf{F}\phi)$	[some F uture state]
$(\mathbf{G}\phi)$	[all future states (G lobally)]
$(\phi \mathbf{U}\phi)$	[U ntil]
$(\phi \mathbf{W}\phi)$	[W eak-untill]
$(\phi \mathbf{R}\phi)$	[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)

$\phi ::= \top$	[<i>true</i>]
\perp	[<i>false</i>]
p	[propositional atom]
$(\neg\phi)$	[logical negation]
$(\phi \wedge \phi)$	[logical conjunction]
$(\phi \vee \phi)$	[logical disjunction]
$(\phi \Rightarrow \phi)$	[logical implication]
$(\mathbf{X}\phi)$	[n e X t state]
$(\mathbf{F}\phi)$	[some F uture state]
$(\mathbf{G}\phi)$	[all future states (G lobally)]
$(\phi \mathbf{U}\phi)$	[U ntil]
$(\phi \mathbf{W}\phi)$	[W eak-untill]
$(\phi \mathbf{R}\phi)$	[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	○
Future/Eventually	F	◇
Global/Henceforth	G	□

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:

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

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

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).

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

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:

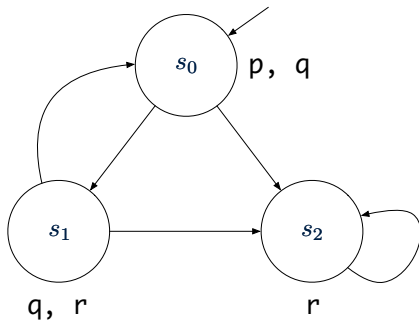
$$0 \leq c \leq 3$$

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



$M = (S, \longrightarrow, L)$:

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

LTL Semantics: Paths

Definition. A *path* in a model $M = (S, \longrightarrow, L)$ 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

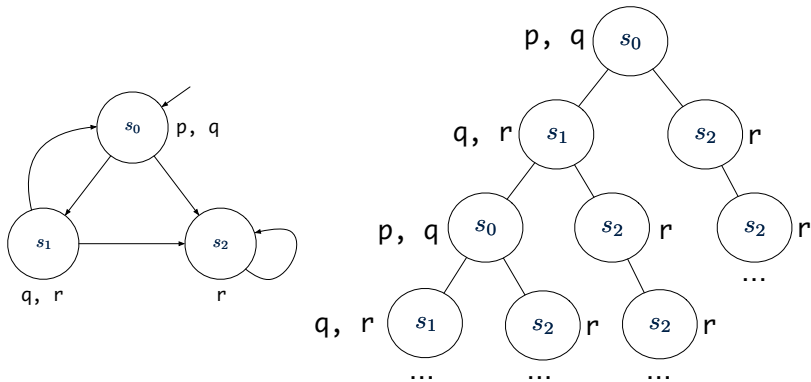
$$s_1 \longrightarrow s_2 \longrightarrow \dots$$

- **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 M .
- We write π^i for the *suffix* of path π : a path starting from state s_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* $\pi = s_1 \longrightarrow \dots$ in \mathbb{M} , whether or not path π satisfies an *LTL formula* is defined by the **satisfaction relation** \models as follows:

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

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* $\pi = s_1 \longrightarrow \dots$ in \mathbb{M} , whether or not path π satisfies an *LTL formula* is defined by the *satisfaction relation* \models as follows:

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

LTL Semantics: Model Satisfaction (1)

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

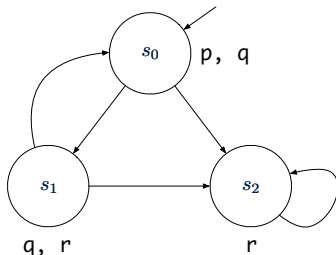
$\mathbb{M}, s \models \phi$ if and only if for **every** path π of \mathbb{M} starting at s , $\pi \models \phi$.

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

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

LTL Semantics: Model Satisfaction (2.1)

Consider the following system model:

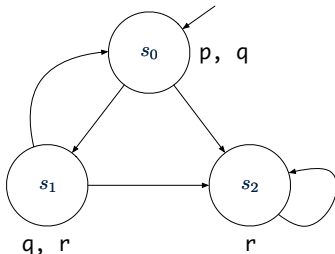


- $s_0 \models T$
- $s_0 \not\models \perp$
- $s_0 \models p \wedge q$
- $s_0 \models r$

[true]
[true]
[true]
[false]

LTL Semantics: Model Satisfaction (2.2)

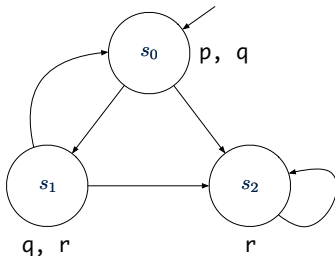
Consider the following system model:



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

LTL Semantics: Model Satisfaction (2.3)

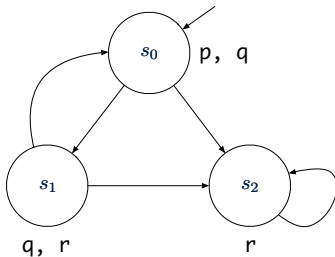
Consider the following system model:



- $s_0 \models \mathbf{G} \neg(p \wedge r)$ [true]
 $s \models \mathbf{G} \phi \iff \phi$ holds on all *reachable* states from s .
- $s_0 \models \mathbf{G} r$ [false]
Witness Path: $s_0 \longrightarrow s_2 \longrightarrow s_2 \cdots \not\models \mathbf{G} r$
- $s_2 \models \mathbf{G} r$ [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 it the case that $q \wedge r$ is eventually satisfied on every path?

• No. Witness Path: $s_0 \rightarrow s_2 \rightarrow s_2 \rightarrow \dots$

○ $s_2 \models \mathbf{F}r$

[true]

LTL Semantics: Nested G and F (1)

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

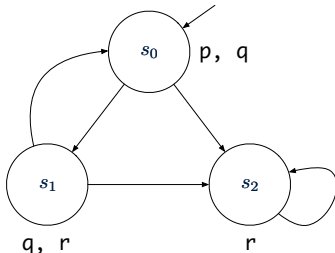
$s \models \mathbf{FG}\phi$ means that:

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

$$\exists i \bullet i \geq 1 \wedge (\forall j \bullet j \geq i \Rightarrow \pi^i \models \phi)$$
- **Q.** How to **prove** and **disprove** the above formula pattern?
- **Hint.** Structure of pattern: $\forall \pi \bullet \dots \Rightarrow (\exists i \bullet \dots \wedge (\forall j \bullet \dots \Rightarrow \phi))$

LTL Semantics: Model Satisfaction (2.5.1)

Consider the following system model:



- $s_0 \models \mathbf{FG} r$ [*false*]
Witness: $s_0 \rightarrow s_1 \rightarrow s_0 \rightarrow s_1 \rightarrow \dots$
- $s_0 \models \mathbf{FG}(p \vee q)$ [*false*]
Witness: $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_2 \rightarrow \dots$
- $s_0 \models \mathbf{FG}(p \vee r)$ [*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 $s \in S$:

$s \models \mathbf{F}\phi_1 \Rightarrow \mathbf{FG}\phi_2$ means that:

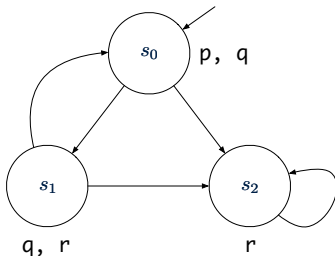
- **Each** path π starting with s is such that if ϕ_1 **eventually** holds on π , then ϕ_2 **eventually** holds **continuously** on the same π .

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

- **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:



- $s_0 \models \mathbf{F}(\neg q \wedge r) \Rightarrow \mathbf{FGR}$

[*true*]

Justification:

- $s_0 \rightarrow s_1 \rightarrow s_0 \rightarrow \dots$ never satisfies $\neg q \wedge r$.
- $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_2 \rightarrow \dots$ eventually satisfies $\neg q \wedge r$ continuously.
- $s_0 \rightarrow s_2 \rightarrow s_2 \rightarrow \dots$ eventually satisfies $\neg q \wedge r$ continuously.

- $s_0 \models \mathbf{F}(\neg q \vee r) \Rightarrow \mathbf{FGR}$

[*false*]

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

LTL Semantics: Nested G and F (3)

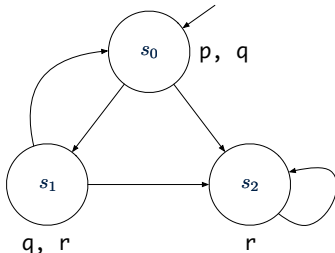
Given a model $\mathbb{M} = (S, \longrightarrow, L)$ and a state $s \in S$:

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

$$\forall i \bullet i \geq 1 \Rightarrow (\exists j \bullet j \geq i \wedge \pi^i \models \phi)$$
 - **Q.** How to *prove* and *disprove* the above formula pattern?
 - **Hint.** Structure of pattern: $\forall \pi \bullet \dots \Rightarrow (\forall i \bullet \dots \Rightarrow (\exists j \bullet \dots \wedge \phi))$

LTL Semantics: Model Satisfaction (2.6)

Consider the following system model:



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

LTL Semantics: Path Satisfaction (2.2)

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

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

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

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

LTL Semantics: Recall Model Satisfaction



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

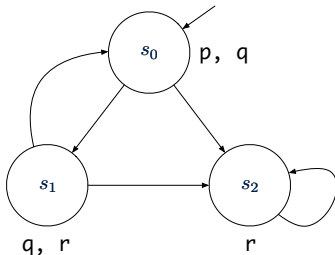
$\mathbb{M}, s \models \phi$ if and only if for **every** path π of \mathbb{M} starting at s , $\pi \models \phi$.

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

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

LTL Semantics: Model Satisfaction (3.1)

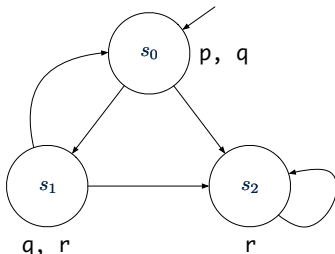
Consider the following system model:



- $s_0 \models p\mathbf{U}r$ [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 $\pi = s_0 \rightarrow s_1 \rightarrow s_0 \rightarrow s_1 \dots$: $\pi \not\models p \wedge r$ 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 $s_0 \rightarrow s_1 \rightarrow s_0 \rightarrow s_1 \dots$, $p \wedge r$ 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? [$s \mathbf{U} (t \wedge (\mathbf{G}\neg s))$]

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.

G (*requested* \Rightarrow (*noOneInCS* **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:

degReqFulfilled, *allowedForGraduation*

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

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

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)

Motivation for Formal Verification

Example of Formal Verification

Classification of Verification Methods

Verification via Model Checking

Model Checking: Temporal Logic

Linear-Time Temporal Logic (LTL)

LTL: Syntax in CFG (1)

LTL: Syntax in CFG (2)

LTL: Syntax in CFG (3)

LTL: Symbols of Unary Temporal Operators

Practical Knowledge about Parsing

Index (2)

LTL: Exercises on Parsing Formulas

LTL Formulas: More Exercises

LTL Formulas: Subformulas

LTL Semantics:

Labelled Transition Systems (LTS)

LTL Semantics: Example of LTS

LTL Semantics: More Example of LTS

LTL Semantics: Paths

LTL Semantics: All Possible Paths

LTL Semantics: Path Satisfaction (1)

LTL Semantics: Path Satisfaction (2.1)

Index (3)

LTL Semantics: Model Satisfaction (1)

LTL Semantics: Model Satisfaction (2.1)

LTL Semantics: Model Satisfaction (2.2)

LTL Semantics: Model Satisfaction (2.3)

LTL Semantics: Model Satisfaction (2.4)

LTL Semantics: Nested G and F (1)

LTL Semantics: Model Satisfaction (2.5.1)

LTL Semantics: Nested G and F (2)

LTL Semantics: Model Satisfaction (2.5.2)

LTL Semantics: Nested G and F (3)

LTL Semantics: Model Satisfaction (2.6)

Index (4)

LTL Semantics: Path Satisfaction (2.2)

LTL Semantics: Recall Model Satisfaction

LTL Semantics: Model Satisfaction (3.1)

LTL Semantics: Model Satisfaction (3.2)

Clarification on the “Until” Connective

Formulating English as LTL Formulas (1)

Formulating English as LTL Formulas (2)

Formulating English as LTL Formulas (3)

Program Verification

Readings: Chapter 4 of LICS2



EECS4315 Z:
Mission-Critical Systems
Winter 2023

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 $\{ x \mid x > 3 \} = \{ 4, 5, 6, 7, \dots \}$
 - $x > 4$ has satisfying values $\{ x \mid x > 4 \} = \{ 5, 6, 7, \dots \}$
- 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 $q(i: \text{INTEGER}) : \text{BOOLEAN}$,

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

$Q_1 : \text{Result} = (i > 0) \vee (i \bmod 2 = 0)$

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

\therefore 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*.
 ⇒ 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*: $\{Q\} S \{R\}$
 - e.g., $\{i > 3\} i := i + 9 \{i > 13\}$
 - e.g., $\{i > 5\} i := i + 9 \{i > 13\}$
 - If $\{Q\} S \{R\}$ **can** be proved **TRUE**, then the **S** is correct.
 e.g., $\{i > 5\} i := i + 9 \{i > 13\}$ can be proved TRUE.
 - If $\{Q\} S \{R\}$ **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:

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

Q is the *precondition* of f .

S is the implementation of f .

R is the *postcondition* of f .

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

Proof of Hoare Triple using wp

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

- $wp(S, R)$ 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 x_0 to denote its **pre-state (old)** value.
 - We write 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
 \therefore All variables are pre-state values in preconditions
 - We don't write “ b_0 ” in program
 \therefore there might be **multiple intermediate values** of a variable due to **sequential composition**

wp Rule: Assignments (1)

$$wp(x := e, R) = R[x := e]$$

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

wp Rule: Assignments (2)

Recall:

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

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

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

wp Rule: Assignments (3) Exercise

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

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

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

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

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 $x = 23$?

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

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

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

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

wp Rule: Assignments (4) Revisit

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

- $n > 4$ 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)

$$wp(\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ end, } R) = \left(\begin{array}{l} B \Rightarrow wp(S_1, R) \\ \wedge \\ \neg B \Rightarrow wp(S_2, R) \end{array} \right)$$

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 ∧ B} S1 {R}
else
  {Q ∧ ¬B} S2 {R}
end
{R}
  
```

$$\begin{aligned}
 & \{Q\} \text{ if } B \text{ then } S_1 \text{ else } S_2 \text{ end } \{R\} \\
 & \iff \left(\begin{array}{c} \{Q \wedge B\} S_1 \{R\} \\ \wedge \\ \{Q \wedge \neg B\} S_2 \{R\} \end{array} \right) \iff \left(\begin{array}{c} (Q \wedge B) \Rightarrow wp(S_1, R) \\ \wedge \\ (Q \wedge \neg B) \Rightarrow wp(S_2, R) \end{array} \right)
 \end{aligned}$$

wp Rule: Alternations (3) Exercise

Is this program correct?

```

{x > 0 ∧ y > 0}
if x > y then
  bigger := x ; smaller := y
else
  bigger := y ; smaller := x
end
{bigger ≥ smaller}
  
```

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

$$\wedge$$

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

wp Rule: Sequential Composition (1)



$$wp(S_1 ; S_2, R) = wp(S_1, wp(S_2, R))$$

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

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

$\therefore \text{True} \Rightarrow y > x$ does not hold in general.

\therefore The above program is not correct.

- 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}
Sinit
while (B) {
  Sbody
}
{R}
  
```

In case of C/Java/PlusCal, 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

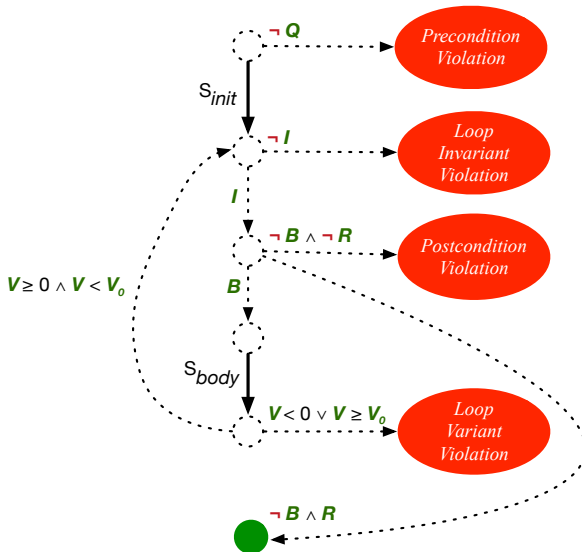
```

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



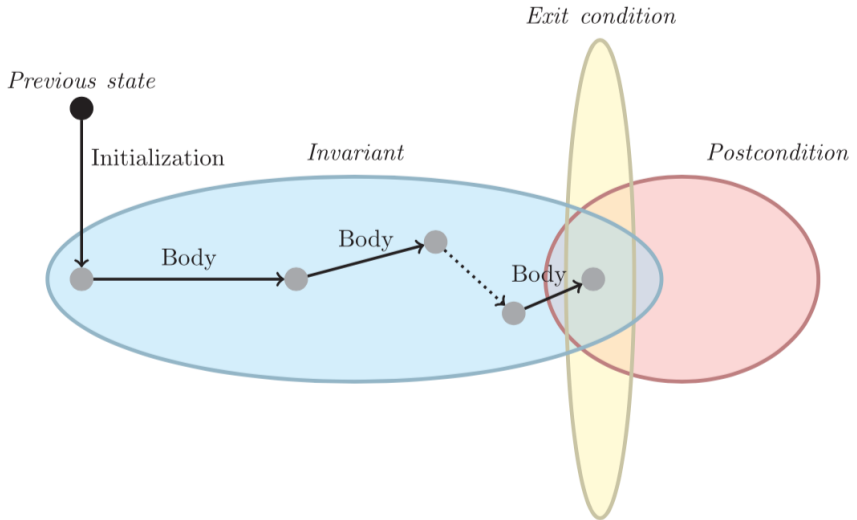
Specifying Loops: Runtime Checks (2)

```
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 $1 \leq i \wedge i \leq 5$ 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}
Sinit
assert I(...);
while( B ) {
  variantpre := V(...);
  Sbody
  variantpost := V(...);
  assert variantpost >= 0;
  assert variantpost < variantpre;
  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 .

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

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

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

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

$$I \wedge \neg B \Rightarrow R$$

- A loop *terminates* if:

- Given LI , and not yet to exit, S_{body} maintains LV V as non-negative.

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

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

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

Index (1)

Learning Objectives

Assertions: Weak vs. Strong

Assertions: Preconditions

Assertions: Postconditions

Motivating Examples (1)

Motivating Examples (2)

Software Correctness

Hoare Logic

Hoare Logic and Software Correctness

Proof of Hoare Triple using wp

Denoting Pre- and Post-State Values

Index (2)

***wp* Rule: Assignments (1)**

***wp* Rule: Assignments (2)**

***wp* Rule: Assignments (3) Exercise**

***wp* Rule: Assignments (4) Exercise**

***wp* Rule: Assignments (5) Revisit**

***wp* Rule: Alternations (1)**

***wp* Rule: Alternations (2)**

***wp* Rule: Alternations (3) Exercise**

***wp* Rule: Sequential Composition (1)**

***wp* Rule: Sequential Composition (2)**

***wp* Rule: Sequential Composition (3) Exercise**

Index (3)

Loops

Correctness of Loops

Specifying Loops

Specifying Loops: Syntax

Specifying Loops: Runtime Checks (1)

Specifying Loops: Runtime Checks (2)

Specifying Loops: Visualization

Proving Correctness of Loops (1)

Proving Correctness of Loops (2)