



Virtual University of Pakistan
FEDERAL GOVERNMENT UNIVERSITY

COMPLETE SLIDES

Formal Methods

CS636

Order Your VU Books Now

Get all Virtual University books in high-quality hard copy,
delivered right to your doorstep. Visit vubookshoppk.com



Order Now

CS636 Formal Methods

Topics Covered

Introduction to Formal Methods

Software Development Life Cycle (SDLC)

- ❑ **Software increasing in size and complexity**
- ❑ ***A Systematic approach is required called SDLC***
- ❑ ***Requirement Specification***
- ❑ ***Design***
- ❑ ***Coding***
- ❑ ***Deployment***

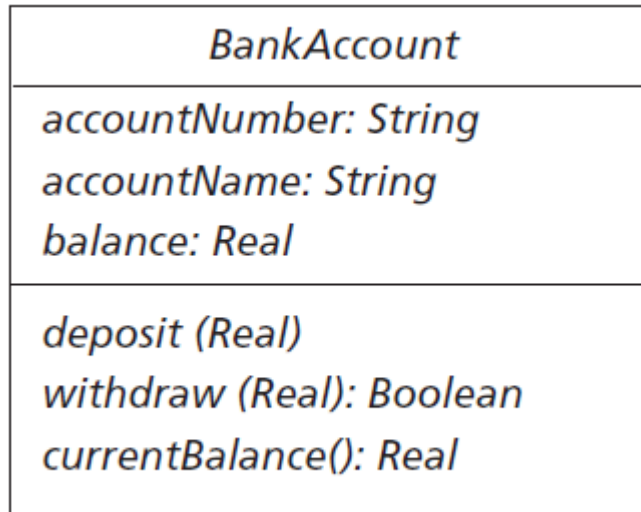
Software Errors

- ❑ **Increased size and complexity leads to software errors**
 - ❑ *Error: software does not provide the intended behavior*
- ❑ ***Errors lead to failure***
 - ❑ *Cost*
- ❑ ***High reliability software cannot tolerate errors***
 - ❑ *Incorrect specifications OR Specification were not followed (misinterpreted)*
 - ❑ *Specifications are important for error free software*

Software Specifications

- ❑ **Correctness of entire software depends on specification**
 - ❑ *Ripple affect*
- ❑ ***Currently written in natural language and diagrammatic notations (UML for example)***
 - ❑ *Ambiguous*
 - ❑ *Incomplete*
 - ❑ *Inconsistent*

Software Specifications



A typical UML diagram for the *BankAccount* class *

Withdraw: *Receives a requested amount to withdraw from the bank account and, if there are **sufficient** funds in the account, meets the request. Returns a boolean value indicating success or failure of the attempt to withdraw money from the account.*

- Is it that the bank account must contain *at least* the amount of money that is requested for withdrawal?
- Or is there a minimum balance that must be maintained?
- Or is there an agreed overdraft limit?

* Formal Software Development, from VDB to Java by Quentin Charatan and Aaron Kans

Software Specifications

- ❑ **Specification of “Withdraw” method is:**
 - ❑ **Ambiguous (different interpretations)**
 - ❑ **Incomplete (What should happen when there is insufficient amount)**
 - ❑ ***Inconsistent (Overdraft facility may be defined somewhere else)***

Software Testing

□ Testing aims to locate the errors

- *Testing cannot take place until some implementation is available*
- *Correcting errors identified by testing may introduce new errors*
- *Testing can uncover errors, does not guarantee absence of them*
- *Testing is based on specifications, ambiguous specifications may make testing ineffective*

Formal Notations

- ❑ It is desirable to use a specification notation with a fixed, unambiguous semantics.
- ❑ Notations that have a fixed semantics are known as formal notations, or formal languages.
- ❑ A fixed semantics is achieved by defining a language in a completely unambiguous way using a mathematical framework.

Formal Methods

- ❑ Formal methods are methods that use formulas. A formula is a text or diagram constructed from predefined symbols combined according to explicit rules.
- ❑ A good working definition of formula is anything whose appearance or syntax can be checked by a computer.
- ❑ *Z language*

Formal Methods

- **Formal methods constitute a branch of software engineering that incorporates the use of mathematics for software development.**
- **Formal methods use formula. A good working definition of formula is anything whose appearance or syntax can be checked by a computer.**

Advantages of formal methods

- ❑ Formal specifications can help considerably in generating suitable test cases.
- ❑ Formal specifications allow for feedback on system specifications at early development stages
- ❑ Important properties (such as internal consistency) of the initial specification can be checked mathematically
- ❑ A proof of program correctness can be constructed
- ❑ Proofs can help uncover design errors as soon as they are made

Thank You

CS636 Formal Methods

Topics Covered

Introduction to Formal Methods

When are the formal methods useful?

- ❑ **Novel Projects:** Novel Projects are about building something new, similar of which does not exist before.
 - ❑ Compare alternative ideas
 - ❑ Select the best one (through in-depth analysis)
 - ❑ Because we cannot afford building the alternative versions
- ❑ **Difficult Project:** Projects are difficult when they address something profound or deep
 - ❑ May not need to be long
 - ❑ A simple page of code can be too difficult

When are the formal methods useful?

- ❑ **Critical Projects: So much is at stake**
 - ❑ Money
 - ❑ Security
 - ❑ Safety
- ❑ **You may not need to use formal methods for simple and routine jobs which are obvious and too easy.**

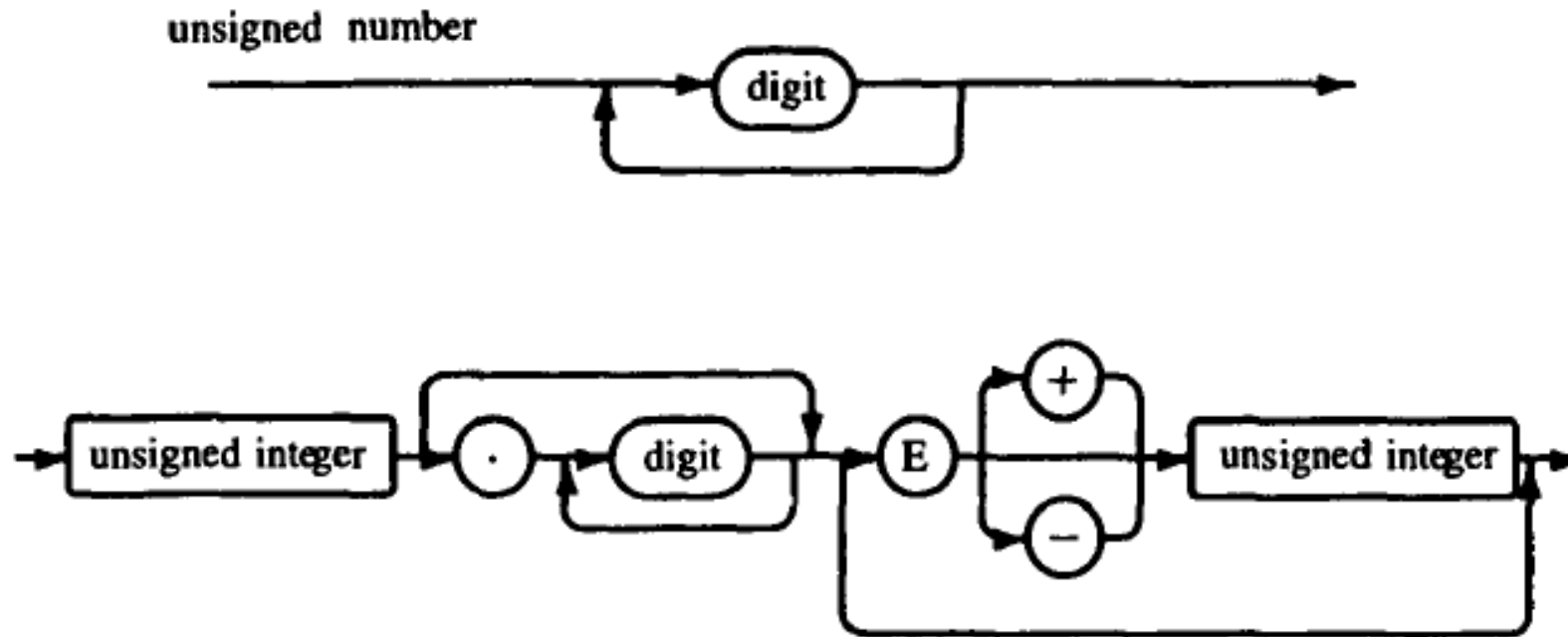
How can we use formal Methods?

□ Modeling

- Models describe the actual product behavior (before implementing it) through some symbols/notations etc.
- A model is a simplified representation. It may skip the irrelevant properties
- A mathematical model that represents the intended behavior of a program can be used as a formal specification

A formal model of how to form a number

- Specification permits 1, 0.5, 1E5, and 1.5E-5
- prohibits 1., .5, E5, 1. 5E-5.0



How can we use formal Methods?

- **Design**

- **Top Down**

- **Bottom Up**

- **Z Language**

How can we use formal Methods?

□ Verification

- The code does what it is intended for
- A convincing product of verification is a proof
- Proof provides greater confidence than testing

Thank You

CS636 Formal Methods

Topics Covered

Mathematical Notations

Notations and descriptions

$\mathbb{N}$ *Natural Numbers*

$\mathbb{Z}$ *Integers*

$\forall$ *For all*

$\exists$ *There exists*

$\nexists$ *There does not exist*

$\wedge$ *Logical AND*

$\vee$ *Logical OR*

CS636 Formal Methods

Topics Covered

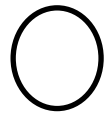
Mathematical Models

Petri nets

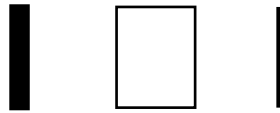
- Introduced by Carl Adam Petri in 1962
- Diagrammatic tool to model concurrency and synchronization
- Provides mathematical modeling as well

Petri nets

- Two main components are Places and transitions.
- Place (Represents condition or State) denoted by Circle



- Transition (Represents Events) denoted by Rectangle, Vertical bar or filled vertical rectangle



- Arc (connect Transitions and Places) denoted by directed arrow

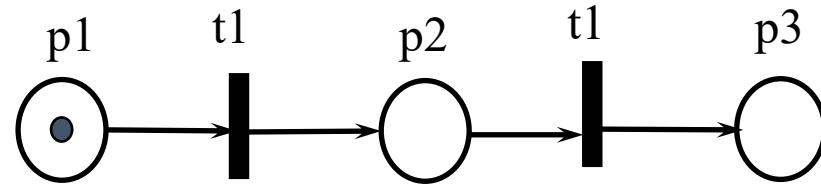


- Tokens (represents state at time t) denoted by a filled circle

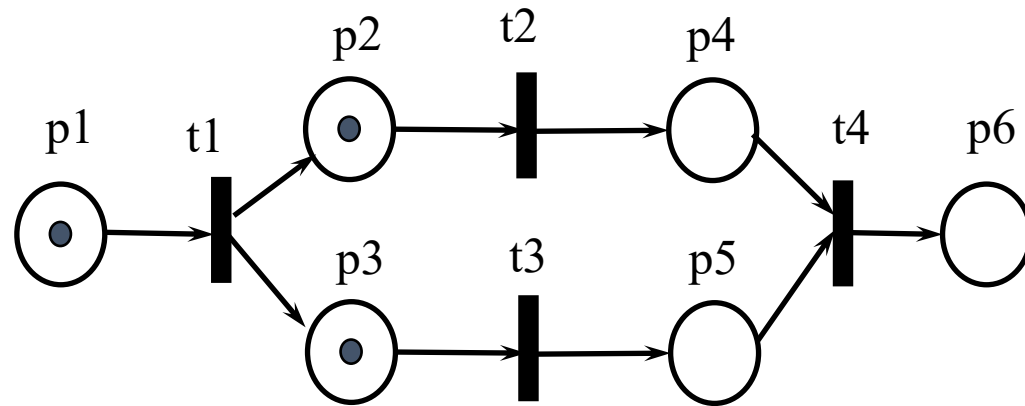


- Each transition has an input place and an output place. Input place is also called pre-condition and output place is also called post condition
- For a transition to fire, there should be at least one token at each input place
- Once a transition is fired, one token is removed from all of its input places and one token is pasted in all of its output places

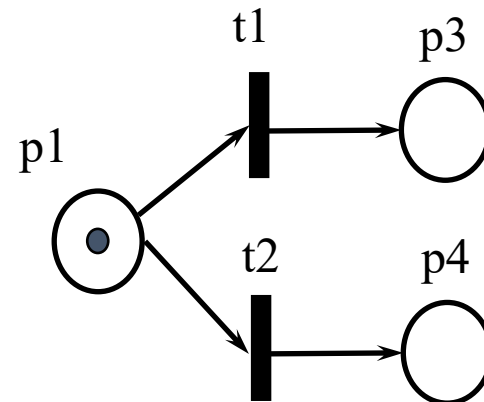
- Sequence



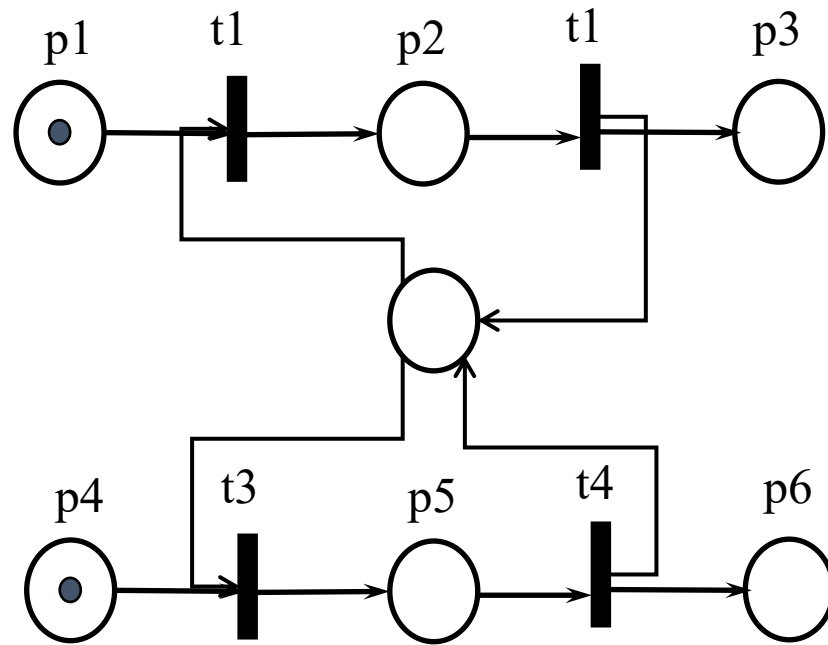
- Concurrency



- Conflict



Resource Sharing

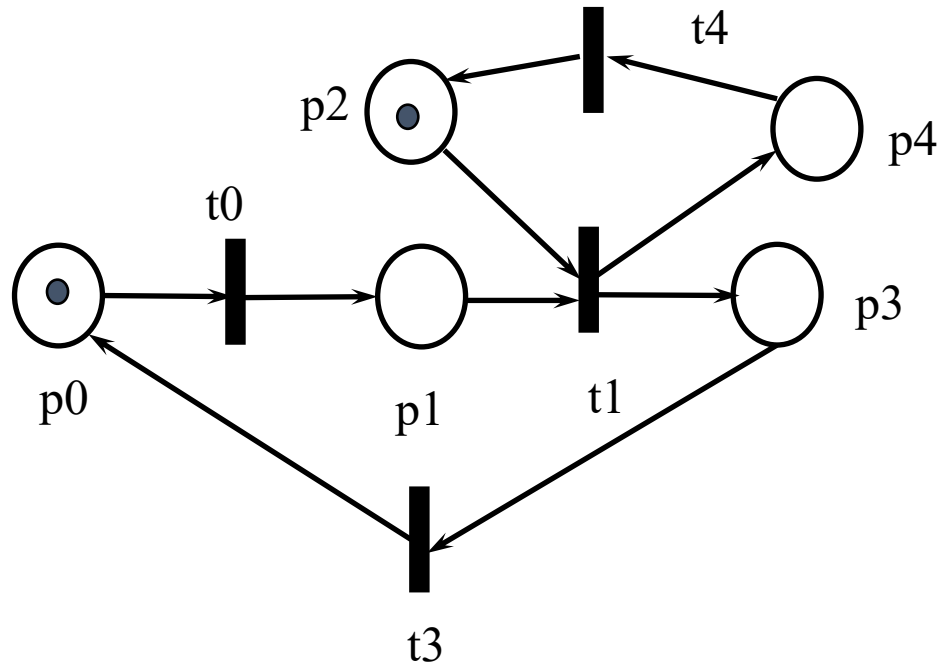


CS636 Formal Methods

Topics Covered

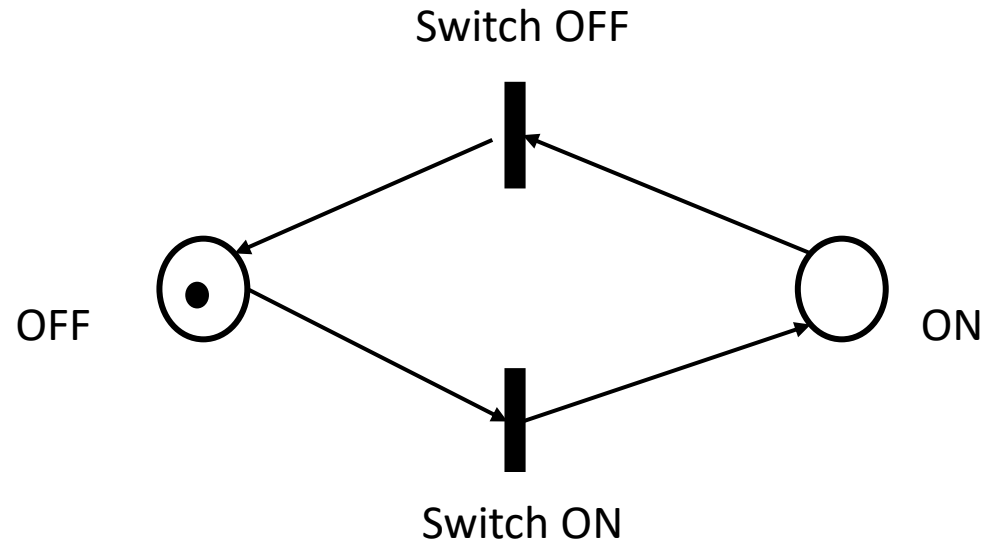
Formal Specification Techniques

Petrinets

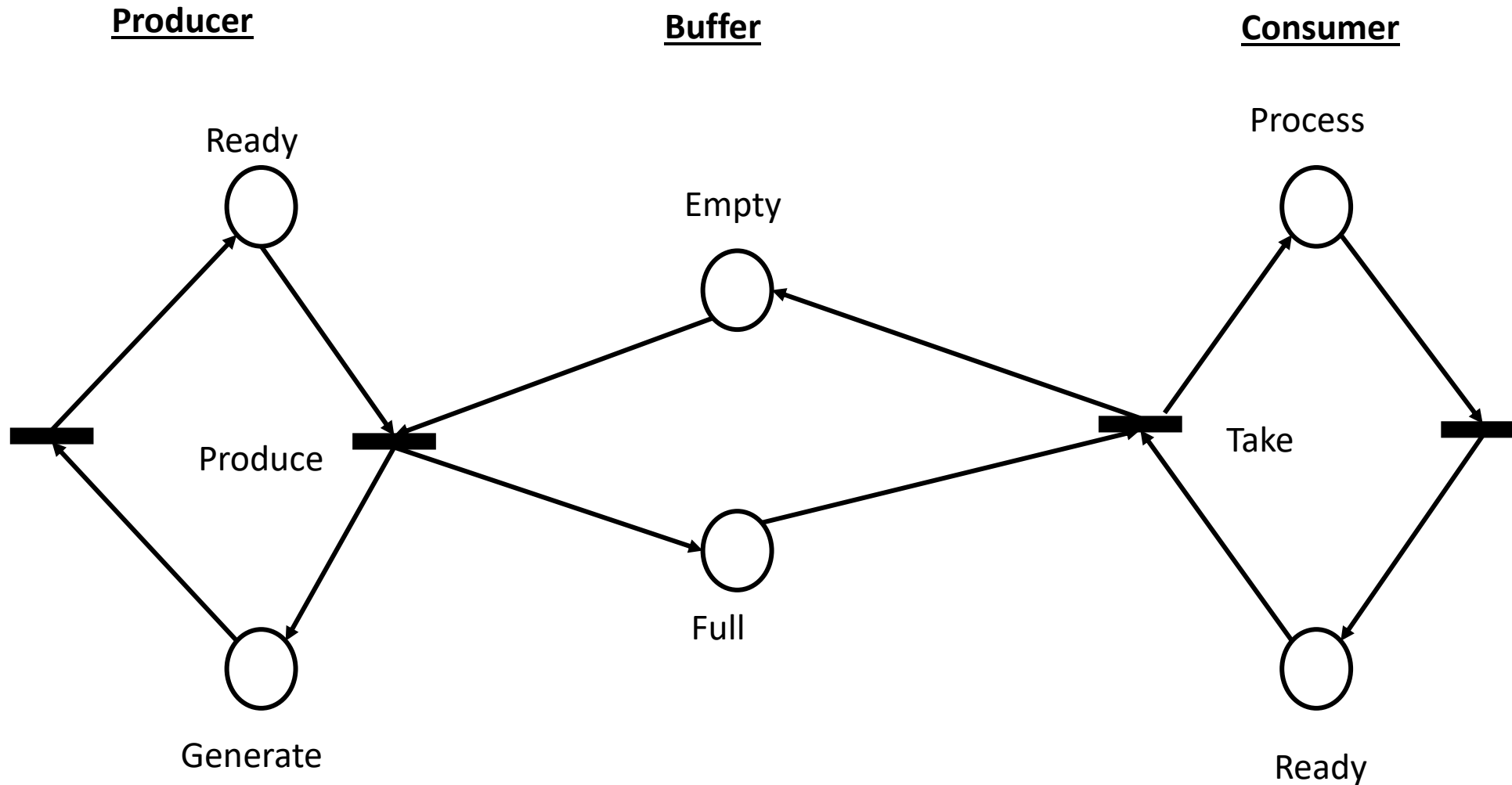


- A transition can be fired when there is at least one token on each of the transition's input places
- When a transition is fired it one token is removed from each of its input places and produced a single token on each of its output places.
- A transition that is able to fire is said to be enabled and otherwise disabled.
- If there is more than one enabled transition any one of enabled transitions may be the next one to fire.

Petri net: A simple example



Producer Consumer Problem



CS636 Formal Methods

Topics Covered

Mathematical Models

Formal Definition

- Petri net $PN = (P, T, I, O)$
- P = Set of Places
- T = Set of Transitions
- I = Inputs
- O = Outputs

Marking

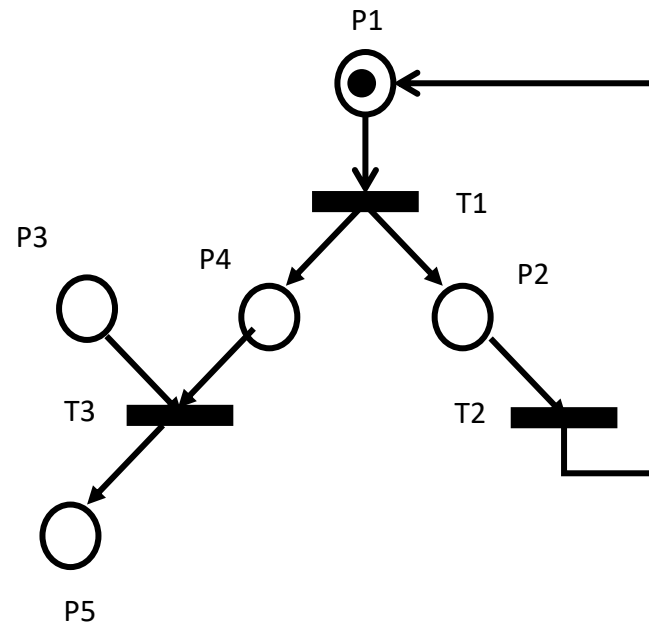
- Marking M represents the state of a petri net

Reachability

- A Marking M_i is reachable from M_o if there exists a firing sequence that transforms M_o to M_i

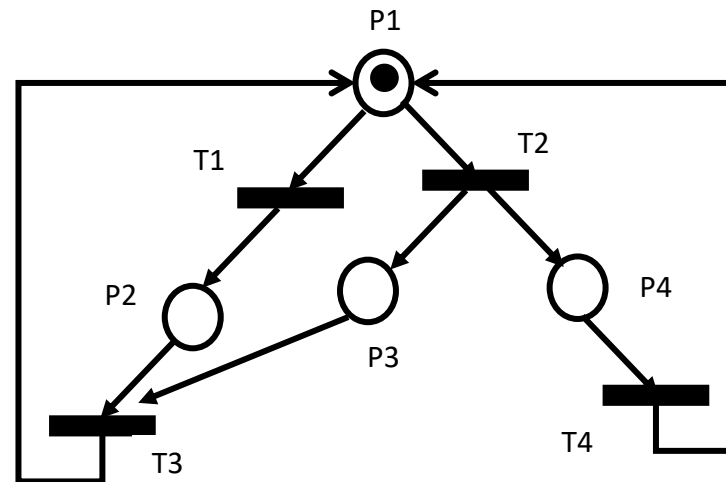
Boundedness

- A Petri net is K bounded if the number of tokens in any place does not exceed K for any marking reachable from M_0



Liveness

- A Petri net with initial marking M_0 is live if for any Marking M_i (after M_0) we can still fire a transition.
- If a petri net is live, it is deadlock safe



CS636 Formal Methods

Topics Covered

Automata

Automata

- Automata are abstract mathematical models of machines that perform computations on an input by moving through a series of states or configurations.

Language

- Language describes the conditions imposed on its words
- The language L of strings of odd length, defined over $\Sigma=\{a\}$, can be written as $L=\{a, aaa, aaaaa, \dots\}$

Kleene Star Closure

- Given Σ , then the Kleene Star Closure of the alphabet Σ , denoted by Σ^* , is the collection of all strings defined over Σ , including Λ .
- Examples:
- If $\Sigma = \{x\}$ Then
- $\Sigma^* = \{\Lambda, x, xx, xxx, xxxx, ...\}$
- If $\Sigma = \{0,1\}$ Then $\Sigma^* = \{\Lambda, 0, 1, 00, 01, 10, 11, ...\}$
- If $\Sigma = \{aaB, c\}$ Then $\Sigma^* = \{\Lambda, aaB, c, aaBaaB, aaBc, caaB, cc, ...\}$

PLUS operation

- Plus Operation is same as Kleene Star Closure except that it does not generate Λ (null string), automatically.
- Example
- If $\Sigma = \{0,1\}$ Then $\Sigma^+ = \{0, 1, 00, 01, 10, 11, ...\}$
- If $\Sigma = \{aab, c\}$ Then $\Sigma^+ = \{aab, c, aabaab, aabc, caab, cc, ...\}$

Regular Expression

- As discussed earlier that a^* generates $\Lambda, a, aa, aaa, \dots$ and
- a^+ generates $a, aa, aaa, aaaa, \dots$,
- so the language
- $L1 = \{\Lambda, a, aa, aaa, \dots\}$ and $L2 = \{a, aa, aaa, aaaa, \dots\}$ can simply be expressed by a^* and a^+ , respectively.
- a^* and a^+ are called the regular expressions (RE) for $L1$ and $L2$ respectively

Regular Expression

- Every letter of Σ including Λ is a regular expression.
- If r_1 and r_2 are regular expressions then
 - (r_1)
 - $r_1 r_2$
 - $r_1 + r_2$ and
 - r_1^* are also regular expressions.

CS636 Formal Methods

Topics Covered

Automata

Regular expressions

- $a(a+b)^*a$
- $(aa+bb)(ab)$
- $b(aa+bb)^+$

Finite Automata

- Finite automata is a state machine that takes a string of symbols as input and changes its state accordingly.
- A string is accepted if it reaches final state
- We will discuss two automata
 - Deterministic Finite Automata
 - Non Deterministic Finite Automata

Deterministic Finite Automata

□ A DFA is a 5-tuple, (S, Σ, T, s, A)

- S represents finite set of states
- Σ represents set of input symbols
- T set of transitions
- $s \in S$ represents initial state
- $A \in S$ represents final state

Deterministic Finite Automata

- $S = \{a, b, c\}$,
- $\Sigma = \{0, 1\}$,
- $s = \{a\}$,
- $A = \{c\}$

Current State	Next State for input 0	Next state for input 1
a	b	a
b	a	c
c	b	c

Non Deterministic Finite Automata

□ A DFA is a 5-tuple, (S, Σ, T, s, A)

- S represents finite set of states
- Σ represents set of input symbols
- T set of transitions
- $s \in S$ represents initial state
- $A \in S$ represents final state

Non Deterministic Finite Automata

- $S = \{a, b, c\}$,
- $\Sigma = \{0, 1\}$,
- $s = \{a\}$,
- $A = \{c\}$

Current State	Next State for input 0	Next state for input 1
a	a,b	b
b	a	a, c
c	b,c	c

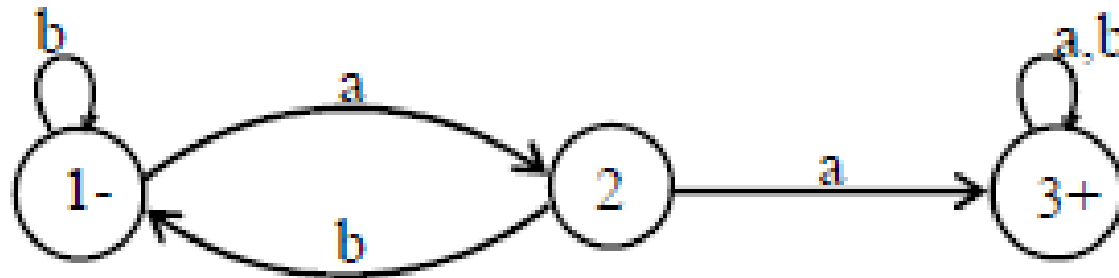
CS636 Formal Methods

Topics Covered

Automata

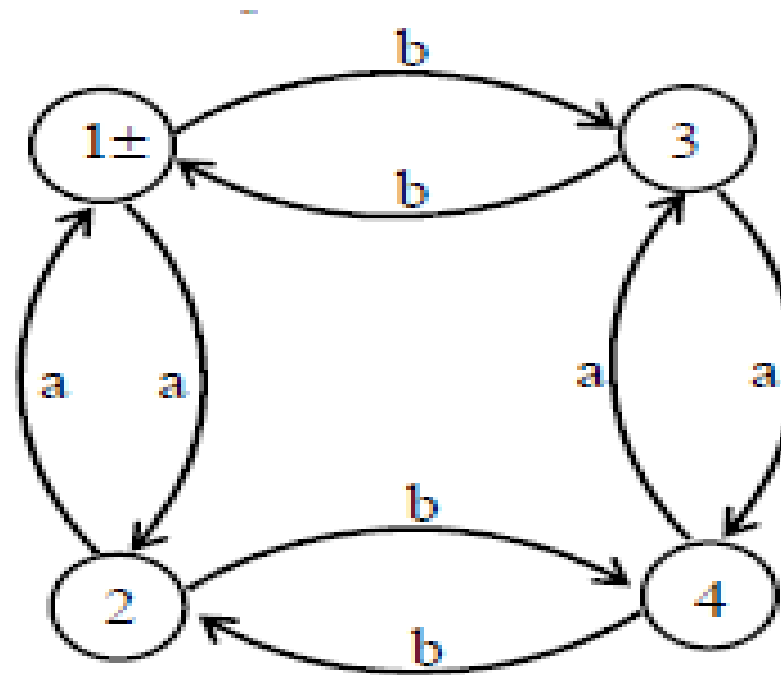
Deterministic Finite Automata: Example

- Consider the Language L of strings , defined over $\Sigma = \{a, b\}$, containing double a.
- RE: $(a+b)^*aa(a+b)^*$



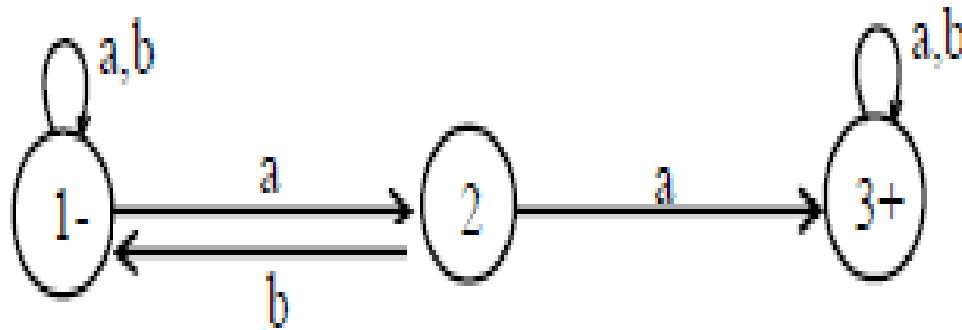
Deterministic Finite Automata: Example

- Consider the EVEN-EVEN language, defined over $\Sigma = \{a, b\}$.
- $(aa+bb+(ab+ba)(aa+bb)^*(ab+ba))^*$



Non Deterministic Finite Automata: Example

- Consider the Language L of strings , defined over $\Sigma = \{a, b\}$, containing double a.
- RE: $(a+b)^*aa(a+b)^*$



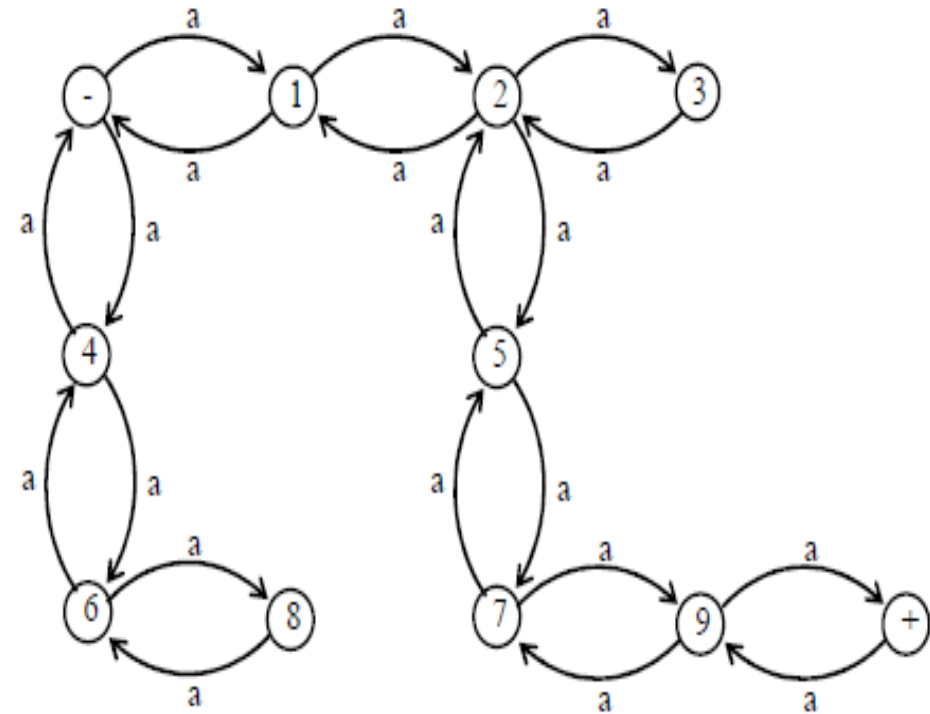
Non Deterministic Finite Automata: Example

- Maze Game
- - and + indicate the initial and final states respectively. One can move only from a box labeled by other then L, M, N, O, P to such another box.

-	1	2	3
4	L	5	O
6	M	7	P
8	N	9	+

Non Deterministic Finite Automata: Example

-	1	2	3
4	L	5	O
6	M	7	P
8	N	9	+



Deterministic Finite Automata: Example

- A Vending Machine
- Accepts \$1 and \$0.25 per soda
- Once machine receives \$1.25, it allows user to select soda
- Self-loop represents ignored input
- Soda will not be dispensed until \$1.25 is deposited
- Machine will not accept more money, once \$1.25 is deposited

$S = \{\$0.00; \$0.25; \$0.50; \$0.75; \$1.00; \$1.25; \$1.50; \$1.75; \$2.00\}$

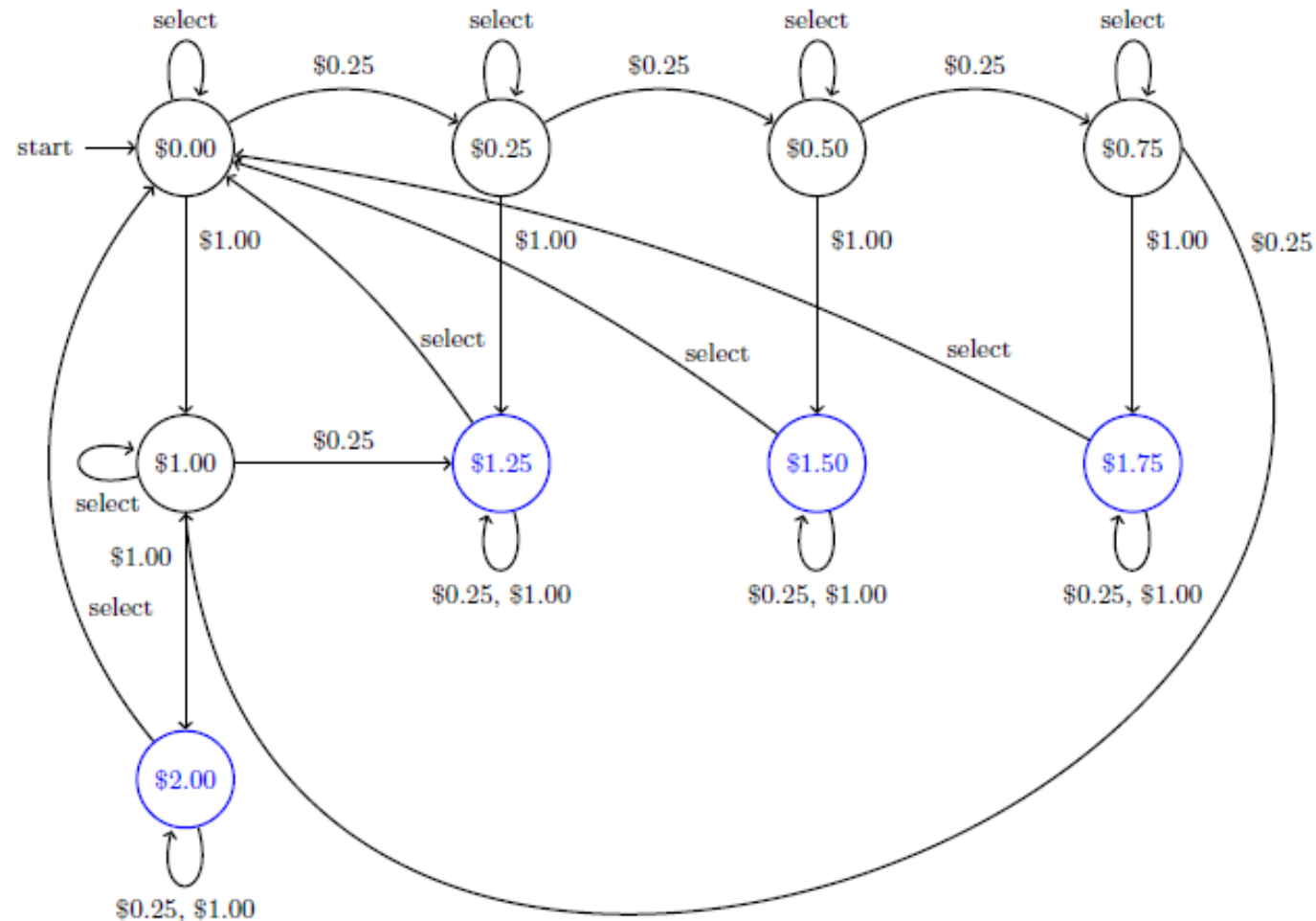
$\Sigma = \{\$0.25; \$1.00; \text{select}\}$

The transition function, is described by the state diagram.

$s = \$0.00$ is the start state

$A \in S$ is the set of accept states

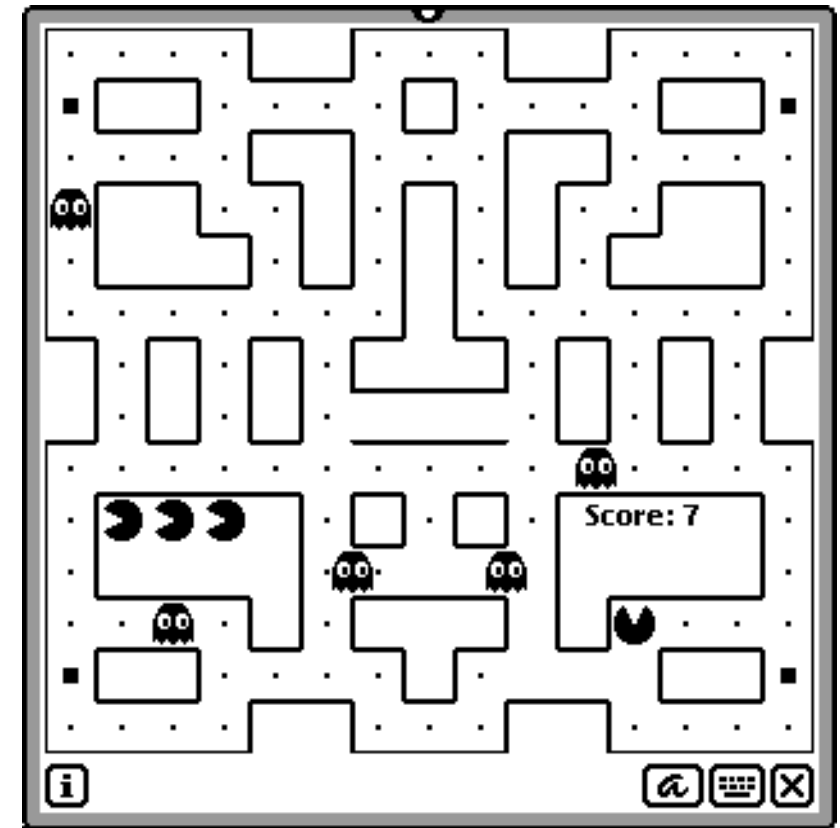
Deterministic Finite Automata: Example



Source: Gribkoff, E. (2013). Applications of deterministic finite automata. *UC Davis*, 1-9.

Deterministic Finite Automata: Example

- Pac-Man's Ghosts
- Player Navigates through the maze, eating pellets and avoiding ghost.
- If Pac-Man eats power pellet, it can eat ghosts.
- Once a Pac-Man eats power pallet, the ghost try to avoid Pac-Man

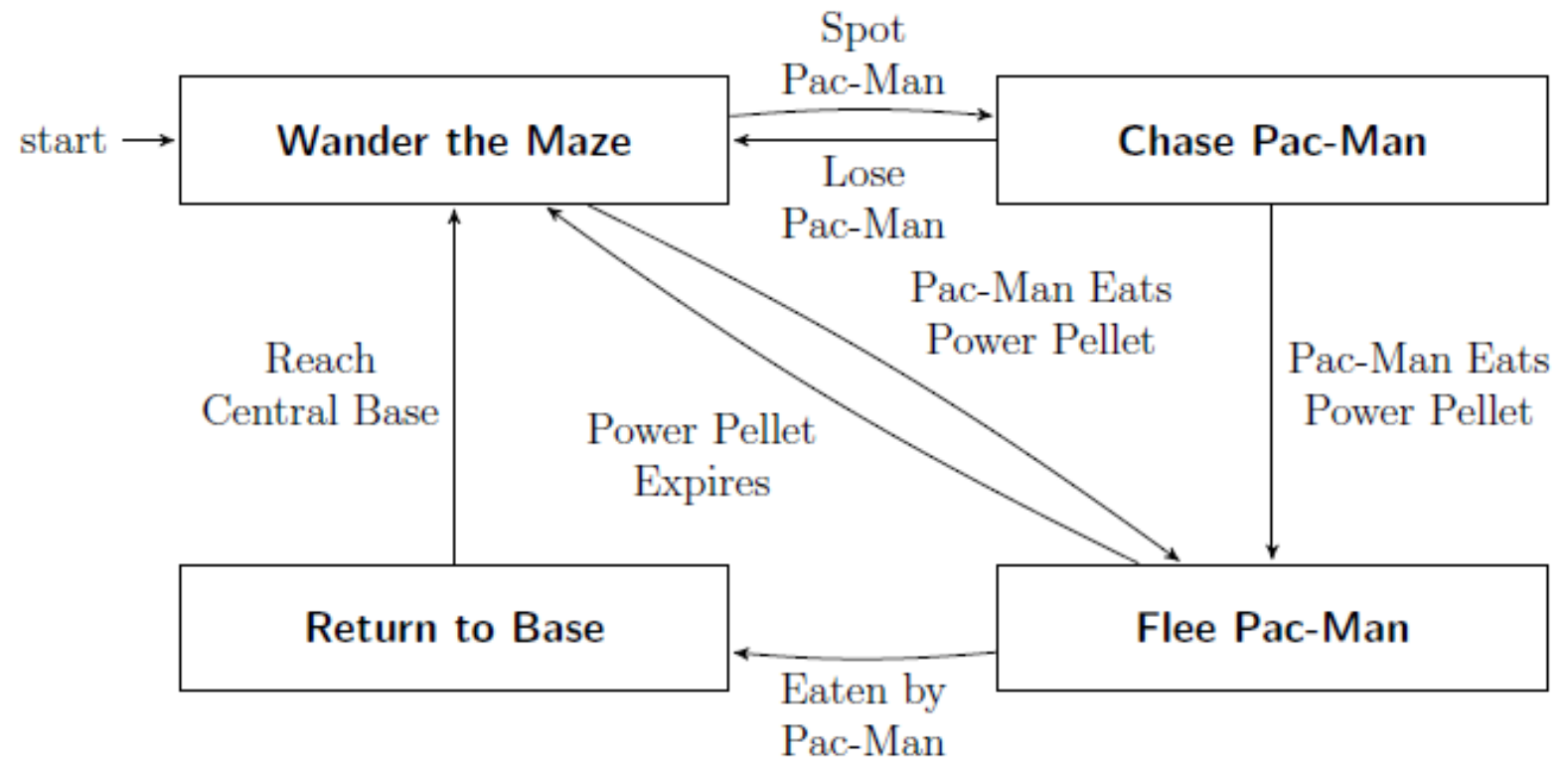


Source: Gribkoff, E. (2013). Applications of deterministic finite automata. *UC Davis*, 1-9.

Deterministic Finite Automata: Example

- **Behavior of Ghosts:**

- 1. Randomly wander the maze
- 2. Chase Pac-Man, when he is within line of sight
- 3. Flee Pac-Man, after Pac-Man has consumed a power pellet
- 4. Return to the central base to regenerate



Source: Gribkoff, E. (2013). Applications of deterministic finite automata. *UC Davis*, 1-9.

CS636 Formal Methods

Topics Covered

Formal Verification

References:

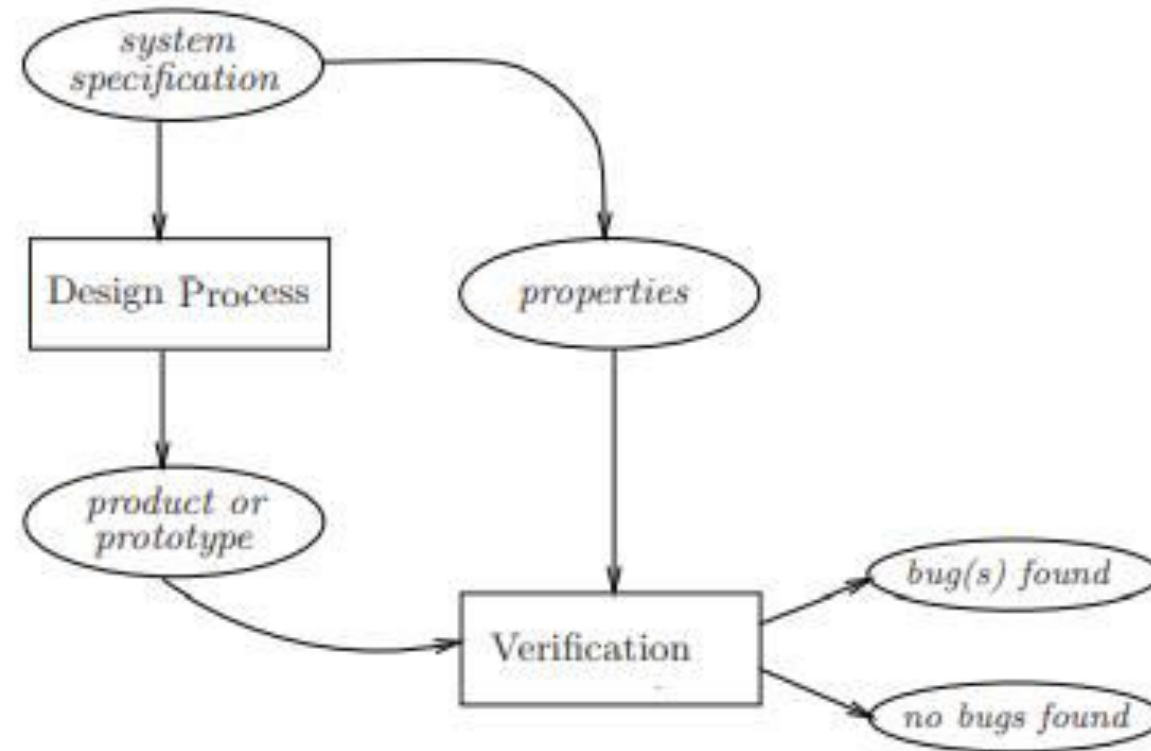
All the contents in these slides are taken from:

- Baier, Christel, and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

System Verification

- ❑ The reliability of ICT systems is a key issue in the system design process
- ❑ Reliability verification:
 - ❑ *Software System*
 - ❑ *Hardware System*
 - ❑ *Concurrent System*

System Verification



Verification of the Models

- **Model-based verification techniques are based on models describing the possible system behavior in a mathematically precise and unambiguous manner.**
- ***The accurate modeling of systems often leads to the discovery of incompleteness, ambiguities, and inconsistencies in informal system specifications***

Verification of the Models

- **The system models are accompanied by algorithms that systematically explore all states of the system model.**
 - *Model checking*
 - *Simulation*
 - *Testing*

Model Checking

- Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model.
- Model checking explores all possible system states in a brute-force manner.
- *it can be shown that a given system model truly satisfies a certain property.*

Model Checking

□ Qualitative Properties:

- Is the generated result OK?
- Can the system reach a deadlock situation?

□ *Timing properties:*

- *Can a deadlock occur within 1 hour after a system reset?*
- *Is a response always received within 8 minutes?*

Model Checking

- ❑ **Model checking requires a precise and unambiguous statement of the properties to be examined.**
- ❑ **Making an accurate system model, this step often leads to the discovery of several ambiguities and inconsistencies in the informal documentation.**

Model Checking: Example

- **Concurrency and Atomicity**
- ***proc Inc = while true do if $x < 200$ then $x := x + 1$ fi od***
- ***proc Dec = while true do if $x > 0$ then $x := x - 1$ fi od***
- ***proc Reset = while true do if $x = 200$ then $x := 0$ fi od***

CS636 Formal Methods

Topics Covered

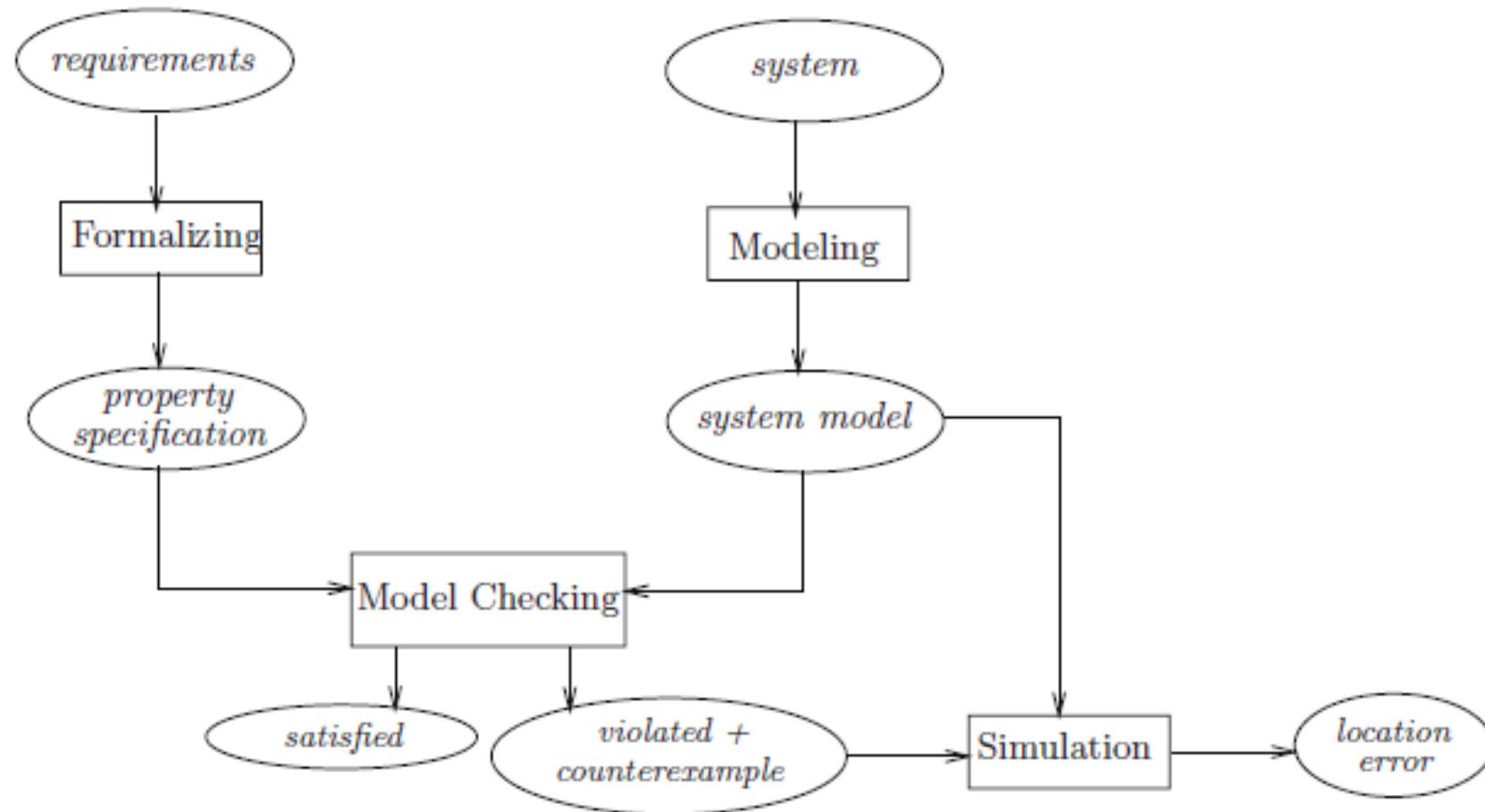
Formal Verification

References:

All the contents in these slides are taken from:

- Baier, Christel, and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

The Model-Checking Process



The Model-Checking Process

□ Modeling phase:

- model the system under consideration using the model description language of the model checker at hand;
- as a first sanity check and quick assessment of the model perform some simulations;
- formalize the property to be checked using the property specification language.

□ Running phase: run the model checker to check the validity of the property in the system model.

□ Analysis phase:

- property satisfied? → check next property (if any);
- property violated? →
 1. analyze generated counterexample by simulation;
 2. refine the model, design, or property;
 3. repeat the entire procedure.

□ – out of memory? → try to reduce the model and try again

Strengths of Model checking

- ❑ Wide range of applications
- ❑ *It supports partial verification*
- ❑ *Its not vulnerable to likelihood that an error is exposed*
- ❑ *It provides diagnostic information in case a property is invalidated*
- ❑ *It is a potential “push-button” technology*
- ❑ *It enjoys a rapidly increasing interest by industry;*

Strengths of Model checking

- ❑ It can be easily integrated in existing development cycles;
- ❑ *It has a sound and mathematical underpinning;*

Weaknesses of Model checking

- ❑ It is mainly appropriate to control-intensive applications and less suited for data intensive applications
- ❑ *Its applicability is subject to decidability issues;*
- ❑ *It verifies a system model, and not the actual system (product or prototype) itself.*
- ❑ *It checks only stated requirements*
- ❑ *Its usage requires some expertise in finding appropriate abstractions to obtain smaller system models and to state properties in the logical formalism used*

Weaknesses of Model checking

- ❑ It is not guaranteed to yield correct results: as with any tool, a model checker may contain software defects
- ❑ *It does not allow checking generalizations*

CS636 Formal Methods

Topics Covered

Transition Systems

References:

All the contents in these slides are taken from:

- Baier, Christel, and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

Transition Systems

- ❑ Transition systems are often used in computer science as models to describe the behavior of systems.
- ❑ *They are basically directed graphs where nodes represent states, and edges model transitions, i.e., state changes.*
- ❑ *A state describes some information about a system at a certain moment of its behavior.*
- ❑ *Transitions specify how the system can evolve from one state to another*

Transition Systems

- A transition system TS is a tuple $(S, Act, \rightarrow, I, AP, L)$ where
- *S is a set of states,*
- *Act is a set of actions,*
- *$\rightarrow \subseteq S \times Act \times S$ is a transition relation,*
- *$I \subseteq S$ is a set of initial states,*
- *AP is a set of atomic propositions, and*
- *$L : S \rightarrow 2^{AP}$ is a labeling function.*

Transition Systems

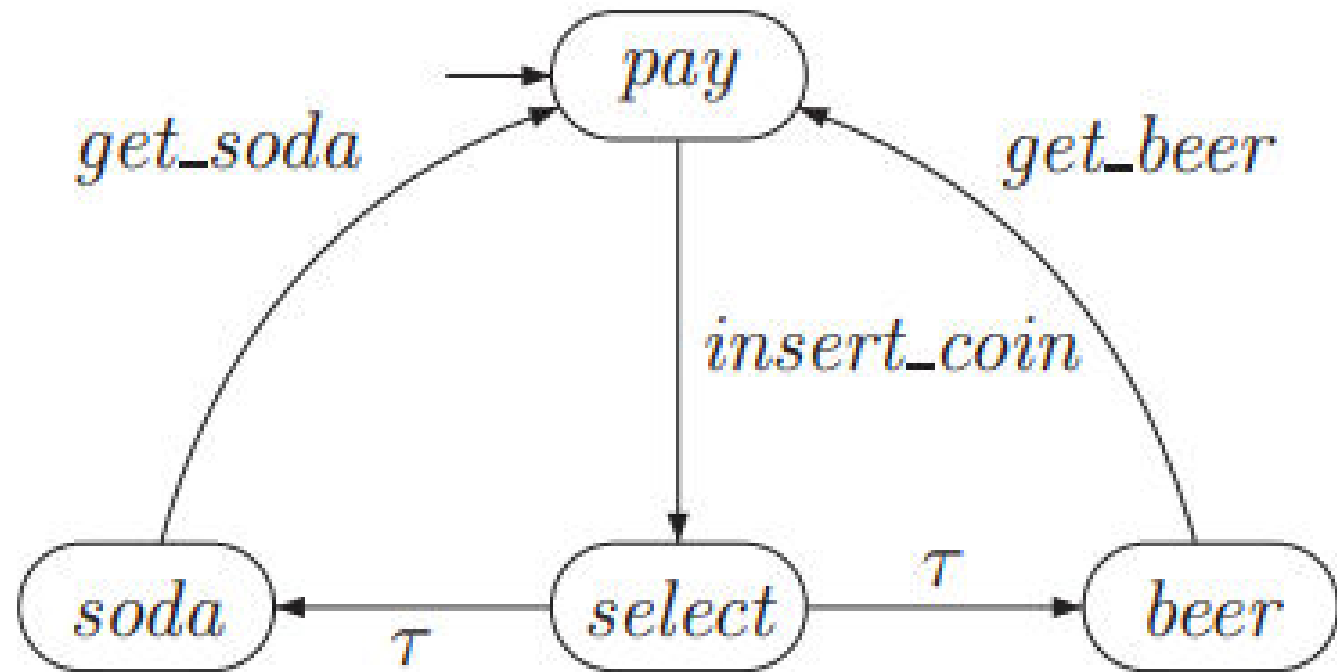
$S = \{\text{pay}, \text{select}, \text{soda}, \text{beer}\}$

$\text{Act} = \{\text{insert\_coin}, \text{get\_soda}, \text{get\_beer}, \tau\}$

$I = \{\text{pay}\}$

$\text{AP} = S$

$L(s) = \{s\}$



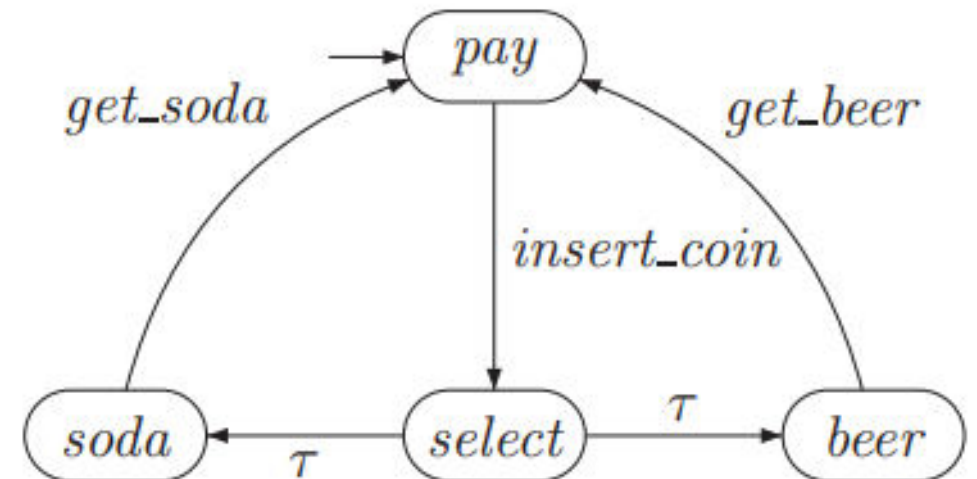
Transition Systems

The vending machine only delivers a drink after providing a coin

$AP = \{ \text{paid}, \text{drink} \}$

$L(\text{pay}) = \emptyset$, $L(\text{soda}) = L(\text{beer}) = \{ \text{paid}, \text{drink} \}$, $L(\text{select}) = \{ \text{paid} \}$.

Here, the proposition *paid* characterizes exactly those states in which the user has already paid but not yet obtained a beverage.



Direct Predecessors and Successors

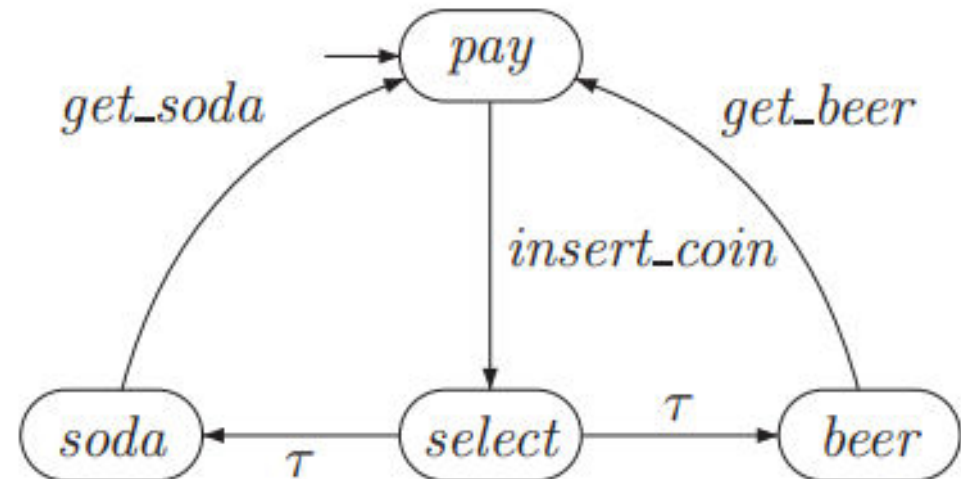
Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system. For $s \in S$ and $\alpha \in Act$, the set of direct α -successors of s is defined as:

$$Post(s, \alpha) = \{ s' \in S \mid s \xrightarrow{\alpha} s' \}, \quad Post(s) = \bigcup_{\alpha \in Act} Post(s, \alpha).$$

The set of α -predecessors of s is defined by:

$$Pre(s, \alpha) = \{ s' \in S \mid s' \xrightarrow{\alpha} s \}, \quad Pre(s) = \bigcup_{\alpha \in Act} Pre(s, \alpha).$$

$Post(\text{pay}, \text{insert\_coin}) = \{\text{select}\}$
 $Pre(\text{pay}, \text{get\_soda}) = \{\text{soda}\}$
 $Pre(\text{pay}) = \{\text{soda}; \text{beer}\}$



Terminal State

State s in transition system TS is called terminal if and only if $\text{Post}(s) = \emptyset$.

Deterministic Transition System

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system.

TS is called action-deterministic if $|I| \leq 1$ and $|\text{Post}(s, \alpha)| \leq 1$ for all states s and actions α .

TS is called AP-deterministic if $|I| \leq 1$ and $|\text{Post}(s) \cap \{s' \in S \mid L(s') = A\}| \leq 1$ for all states s and actions $A \in 2^{AP}$.

Execution Fragment

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system. A finite execution fragment ρ of TS is an alternating sequence of states and actions ending with a state

$\rho = s_0 \alpha_1 s_1 \alpha_2 \dots \alpha_n s_n$ such that $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ for all $0 \leq i \leq n$

n = length of execution fragment

Finite and Infinite Execution Fragment

Finite execution fragment

$\rho = s_0 \alpha_1 s_1 \alpha_2 \dots \alpha_n s_n$ such that $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ for all $0 \leq i \leq n$

Infinite Execution fragment

$\rho = s_0 \alpha_1 s_1 \alpha_2 \dots \alpha_n s_n$ such that $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ for all $0 \leq i$

Maximal and Initial execution fragment

- A maximal execution fragment is either a finite execution fragment that ends in a terminal state, or an infinite execution fragment. An execution fragment is called initial if it starts in an initial state

$$\rho_1 = \text{pay} \xrightarrow{\text{coin}} \text{select} \xrightarrow{\tau} \text{soda} \xrightarrow{\text{sget}} \text{pay} \xrightarrow{\text{coin}} \text{select} \xrightarrow{\tau} \text{soda} \xrightarrow{\text{sget}} \dots$$

$$\rho_2 = \text{select} \xrightarrow{\tau} \text{soda} \xrightarrow{\text{sget}} \text{pay} \xrightarrow{\text{coin}} \text{select} \xrightarrow{\tau} \text{beer} \xrightarrow{\text{bget}} \dots$$

$$\varrho = \text{pay} \xrightarrow{\text{coin}} \text{select} \xrightarrow{\tau} \text{soda} \xrightarrow{\text{sget}} \text{pay} \xrightarrow{\text{coin}} \text{select} \xrightarrow{\tau} \text{soda} .$$

Reachable States

- Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system. A state $s \in S$ is called reachable in TS if there exists an initial, finite execution fragment

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \dots \xrightarrow{\alpha_n} s_n = s .$$

CS636 Formal Methods

Topics Covered

Concurrent Systems

References:

All the contents in these slides are taken from:

- Baier, Christel, and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

Data Dependent Systems

- **The executable actions of a data-dependent system typically result from conditional branching, as in**
 - **if $x \% 2 = 1$ then $x := x + 1$ else $x := 2 \cdot x$ fi. In**



- Labels of conditional transitions are of the form $g : \alpha$ where g is a Boolean condition (called guard), and α is an action that is possible once g holds.

Data Dependent Systems

$select \xrightleftharpoons{nsoda > 0 : sget} start$ and $select \xrightleftharpoons{nbeer > 0 : bget} start$

$select \xrightleftharpoons{nsoda = 0 \wedge nbeer = 0 : ret\_coin} start$

Data Dependent Systems

Action	Effect
<i>refill</i>	$nsoda := max; nbeer := max$
<i>sget</i>	$nsoda := nsoda - 1$
<i>bget</i>	$nbeer := nbeer - 1$

- $Effect : Act \times Eval(Var) \rightarrow Eval(Var)$
- which indicates how the evaluation η of variables is changed by performing an action. If, e.g., α denotes the action $x := y+5$, where x and y are integer variables, and η is the evaluation with $\eta(x) = 17$ and $\eta(y) = -2$, then
- $Effect(\alpha, \eta)(x) = \eta(y) + 5 = -2 + 5 = 3$, and $Effect(\alpha, \eta)(y) = \eta(y) = -2$.

Program Graph

A *program graph* PG over set Var of typed variables is a tuple $(Loc, Act, Effect, \hookrightarrow, Loc_0, g_0)$ where

- Loc is a set of locations and Act is a set of actions,
- $Effect : Act \times Eval(Var) \rightarrow Eval(Var)$ is the effect function,
- $\hookrightarrow \subseteq Loc \times Cond(Var) \times Act \times Loc$ is the conditional transition relation,
- $Loc_0 \subseteq Loc$ is a set of initial locations,
- $g_0 \in Cond(Var)$ is the initial condition.

Program Graph Example

- $Var = \{ nsoda, nbeer \}$
- $Act = \{ bget, sget, coin, ret\ coin, refill \}.$
- $Effect(coin, \eta) = \eta$
- $Effect(ret\_coin, \eta) = \eta$
- $Effect(sget, \eta) = \eta[nsoda := nsoda - 1]$
- $Effect(bget, \eta) = \eta[nbeer := nbeer - 1]$
- $Effect(refill, \eta) = [nsoda := max, nbeer := max]$

Program Graph

The transition system $TS(PG)$ of program graph

$$PG = (Loc, Act, Effect, \hookrightarrow, Loc_0, g_0)$$

over set Var of variables is the tuple $(S, Act, \longrightarrow, I, AP, L)$ where

- $S = Loc \times Eval(Var)$
- $\longrightarrow \subseteq S \times Act \times S$ is defined by the following rule (see remark below):

$$\frac{\ell \xrightarrow{g:\alpha} \ell' \quad \wedge \quad \eta \models g}{\langle \ell, \eta \rangle \xrightarrow{\alpha} \langle \ell', Effect(\alpha, \eta) \rangle}$$

- $I = \{ \langle \ell, \eta \rangle \mid \ell \in Loc_0, \eta \models g_0 \}$
- $AP = Loc \cup Cond(Var)$
- $L(\langle \ell, \eta \rangle) = \{ \ell \} \cup \{ g \in Cond(Var) \mid \eta \models g \}.$

CS636 Formal Methods

Topics Covered

Modeling Concurrent Systems

References:

All the contents in these slides are taken from:

- Baier, Christel, and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

Parallelism and Communication

- Most hard- and software systems are not sequential but parallel in nature.
- Let us assume that the operational (stepwise) behavior of the processes that run in parallel are given by transition systems $TS_1, \dots, TS_n$.
- The goal is to define an operator $||$, such that:
 - $TS = TS_1 || TS_2 || \dots || TS_n$

Parallelism and Communication

- The nature of the operator $||$ will, of course, depend on the kind of communication that is supported.
- Note that the above scheme may be repeated for TS_i , i.e., TS_i may again be a transition system that is composed of several transition systems:
 - $TS_i = TS_{i,1} || TS_{i,2} || \dots || TS_{i,ni} .$

Concurrency and Interleaving

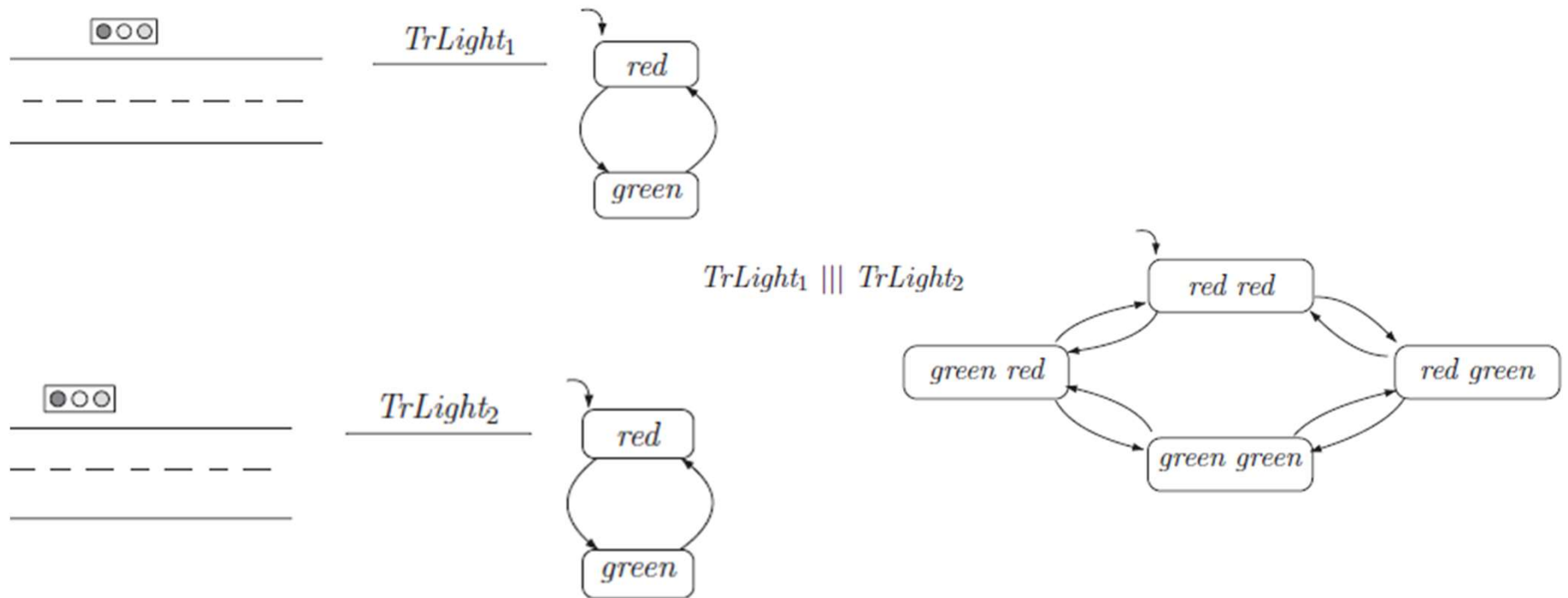
- ❑ Widely adopted paradigm for parallel systems is that of interleaving.
- ❑ The global system state – composed of the current individual states of the components – plays a key role in interleaving.
- ❑ Actions of an independent component are merged (also called weaved), or “interleaved”, with actions from other components.

Concurrency and Interleaving

<i>P</i>	<i>Q</i>	<i>P</i>	<i>Q</i>	<i>P</i>	<i>Q</i>	<i>Q</i>	<i>Q</i>	<i>P</i>	...
<i>P</i>	<i>P</i>	<i>Q</i>	<i>P</i>	<i>P</i>	<i>Q</i>	<i>P</i>	<i>P</i>	<i>Q</i>	...
<i>P</i>	<i>Q</i>	<i>P</i>	<i>P</i>	<i>Q</i>	<i>P</i>	<i>P</i>	<i>P</i>	<i>Q</i>	...

- An important justification for interleaving is the fact that the effect of concurrently executed, independent actions α and β , say, is identical to the effect when α and β are successively executed in arbitrary order. This can symbolically be stated as:
 - $\text{Effect}(\alpha \parallel \beta, \eta) = \text{Effect}((\alpha ; \beta) + (\beta ; \alpha), \eta)$

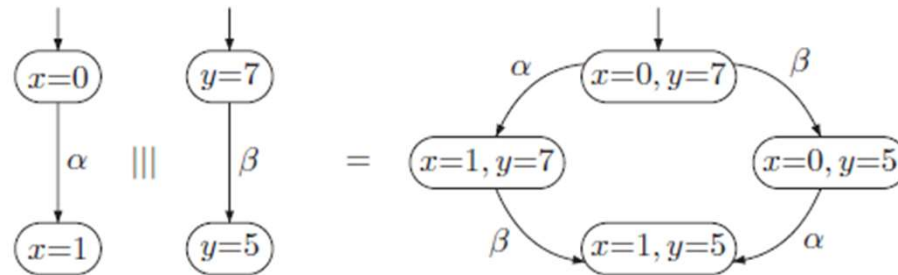
Concurrency and Interleaving



Concurrency and Interleaving

- $\text{Effect}(\alpha \parallel \beta, \eta) = \text{Effect}((\alpha ; \beta) + (\beta ; \alpha), \eta)$

$$\underbrace{x := x + 1}_{=\alpha} \parallel \underbrace{y := y - 2}_{=\beta}.$$



Interleaving of Transition Systems

- The transition system $TS_1 \parallel TS_2$ represents a parallel system resulting from the weaving (or merging) of the actions of the components as described by TS_1 and TS_2 .
- It is assumed that no communication and no contentions (on shared variables) occur at all.
- The (“global”) states of $TS_1 \parallel TS_2$ are pairs $\langle s_1, s_2 \rangle$ consisting of “local” states s_i of the components TS_i .
- The outgoing transitions of the global state $\langle s_1, s_2 \rangle$ consist of the outgoing transitions of s_1 together with those of s_2 .

Interleaving of Transition Systems

Let $TS_i = (S_i, Act_i, \rightarrow_i, I_i, AP_i, L_i)$ $i=1, 2$, be two transition systems. The transition system $TS_1 ||| TS_2$ is defined by:

$$TS_1 ||| TS_2 = (S_1 \times S_2, Act_1 \cup Act_2, \rightarrow, I_1 \times I_2, AP_1 \cup AP_2, L)$$

where the transition relation $\rightarrow$ is defined by the following rules:

$$\frac{s_1 \xrightarrow{\alpha}_1 s'_1}{\langle s_1, s_2 \rangle \xrightarrow{\alpha} \langle s'_1, s_2 \rangle} \quad \text{and} \quad \frac{s_2 \xrightarrow{\alpha}_2 s'_2}{\langle s_1, s_2 \rangle \xrightarrow{\alpha} \langle s_1, s'_2 \rangle}$$

and the labeling function is defined by $L(\langle s_1, s_2 \rangle) = L(s_1) \cup L(s_2)$.

Interleaving of Transition Systems

Let $PG_i = (Loc_i, Act_i, Effect_i, \hookrightarrow_i, Loc_{0,i}, g_{0,i})$, for $i=1, 2$ be two program graphs over the variables Var_i . Program graph $PG_1 ||| PG_2$ over $Var_1 \cup Var_2$ is defined by

$$PG_1 ||| PG_2 = (Loc_1 \times Loc_2, Act_1 \uplus Act_2, Effect, \hookrightarrow, Loc_{0,1} \times Loc_{0,2}, g_{0,1} \wedge g_{0,2})$$

where $\hookrightarrow$ is defined by the rules:

$$\frac{\ell_1 \xrightarrow{g:\alpha}_1 \ell'_1}{\langle \ell_1, \ell_2 \rangle \xrightarrow{g:\alpha} \langle \ell'_1, \ell_2 \rangle} \quad \text{and} \quad \frac{\ell_2 \xrightarrow{g:\alpha}_2 \ell'_2}{\langle \ell_1, \ell_2 \rangle \xrightarrow{g:\alpha} \langle \ell_1, \ell'_2 \rangle}$$

and $Effect(\alpha, \eta) = Effect_i(\alpha, \eta)$ if $\alpha \in Act_i$.

CS636 Formal Methods

Topics Covered

Analyzing Well formedness

References:

All the contents in these slides are taken from:

- Zhu, H., & Shan, L. (2006). Well-formedness, consistency and completeness of graphic models. Proc. of UKSIM, 6, 47-53.

Well formedness

- A model or a diagram is well-formed if its structure satisfies the syntactic rules, especially its elements are type compatible if a type system is defined.
- A Modeling Language: ML
- Model: M
- Model M has Diagrams $D \neq \phi$
- Type of Diagram: $T \neq \phi$
- T-Submodel= The subset of diagrams of type T in a model M = T.M

Consistency Constraints

- **Generally speaking, a consistency constraint C is a predicate defined on models such that $C(M) = \text{true}$ means that the model is consistent with respect to the constraint;**

Consistency Constraints: Intra diagram

- A consistency constraint C is called intra-diagram, if it is defined on a specific type T of diagram of the model in the form of: $C(M) \Leftrightarrow \forall D \in M.T. C'(D)$

Consistency Constraints: Inter diagrams

- A consistency constraint C is inter-diagram, if it is defined on two or more diagrams.
- For example, a consistency constraint C that is defined on two diagrams of type T is an inter-diagram consistency constraint, where

$$C(M) \Leftrightarrow \forall D, D' \in M.T. C'(D, D'),$$

Consistency Constraints: Inter Model

- A consistency constraint C is called inter-model, if it is defined on diagrams of more than one type, say between diagrams of types T_1 and T_2 , so that:

$$C(M) \Leftrightarrow \forall D \in M.T_1, D' \in M.T_2. C'(D, D')$$

Completeness Constraints

- A completeness constraint restricts the construction of the models so that certain types of errors due to the lack of information can be prevented and detected.

$$\forall D \in M_{uc}, \forall X \in D, \exists AD \in M_{AD}, title(X) = title(AD)$$

CS636 Formal Methods

Topics Covered
Formal Reasoning

Formal Reasoning

- It is a process of thinking, logically arguing and drawing conclusions
- It involves the application of well-defined rules, principles, and procedures to analyze and draw conclusions about a given problem or situation.
- A systematic approach to logical thinking and problem-solving using formal systems, such as mathematical or logical frameworks.
- Logic is a language of reasoning.

Logic

- Logic refers to a formal system of principles and rules that govern correct or valid reasoning.
- Logic uses symbols, notation, and formal languages to represent statements and employs logical operators and rules of inference to derive conclusions.
- It provides a framework for evaluating the logical coherence and validity of arguments.
- Application in computer science – design computer circuits, developments of computer programs, verification of the correctness of programs.
- Providing such a rigorous framework makes it possible to reason precisely about statements made in the natural language

Propositions

- Propositions are statements that are either TRUE or FALSE.
- The following are examples of propositions that evaluate to TRUE:
 - There are seven days in a week
 - There are 12 months in a year
 - $2 + 4 = 6$
- The following propositions evaluate to FALSE:
 - The angles of a triangle add up to 360 degree
 - London is the capital of France
 - $2 - 4 = 7$

Propositions

- In mathematics we often represent a proposition symbolically by a variable name
- such as P or Q
- For example:
 - P: I go shopping on Wednesdays
 - Q: $102.001 > 101.31$

Propositions

- In the worlds of computing and mathematics, occasions arise when it is not possible to evaluate expressions precisely – maybe a program terminated incorrectly, or perhaps somebody tried to evaluate the square root of a negative integer.
- it is possible to account for such situations by defining a three-valued logic, which allows a proposition to take the value UNDEFINED as well as TRUE or FALSE.

CS636 Formal Methods

Topics Covered

Propositional Logic

References

- All the contents of this video lecture and PPT slides are taken from:

Theory and Problems of DISCRETE MATHEMATICS, Third Edition, by SEYMOUR LIPSCHUTZ and MARC LARS LIPSON

Propositions

- Propositions are statements that are either TRUE or FALSE.
- The following are examples of propositions that evaluate to TRUE:
 - There are 12 months in a year
 - $3 + 6 = 9$
- The following propositions evaluate to FALSE:
 - Lahore is Capital of Pakistan
 - $4 - 7 = 36$
- In mathematics we often represent a proposition symbolically by a variable name
- such as P or Q
- For example:
 - P: $3 + 6 = 9$
 - Q: I am a citizen of Pakistan

Compound Propositions

- Many propositions are composite, that is, composed of sub-propositions and various connectives. Such composite propositions are called compound propositions.
- A proposition is said to be primitive if it cannot be broken down into simpler propositions, that is, if it is not composite.
- Primitive: $2+2=4$
- Primitive: Ice floats in water
- Compound: Roses are red and violets are blue

Compound Propositions

- The fundamental property of a compound proposition is that its truth value is completely determined by the truth values of its subpropositions together with the way in which they are connected to form the compound propositions.

Basic Logical Operations

- **Conjunction:** Corresponds to English word “and”
- Any two propositions can be combined by the word “and” to form a compound proposition called the conjunction of the original propositions. Symbolically:
 - $p \wedge q$
- Definition: If p and q are true, then $p \wedge q$ is true; otherwise $p \wedge q$ is false.

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

Basic Logical Operators (Conjunction)

1. Ice floats in water and $2 + 2 = 4$.
2. Ice floats in water and $2 + 2 = 5$.
3. China is in Europe and $2 + 2 = 4$.
4. China is in Europe and $2 + 2 = 5$.

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

- Only the first statement is true. Each of the others is false since at least one of its sub statements is false.

Basic Logical Operators

- **Disjunction:** Corresponds to English word “or”
- Any two propositions can be combined by the word “or” to form a compound proposition called the disjunction of the original propositions. Symbolically:
 - $p \vee q$
- Definition: If p and q are false, then $p \vee q$ is false; otherwise $p \vee q$ is true.

p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

Basic Logical Operators (Disjunction)

1. Ice floats in water or $2 + 2 = 4$.
 2. Ice floats in water or $2 + 2 = 5$.
 3. China is in Europe or $2 + 2 = 4$.
 4. China is in Europe or $2 + 2 = 5$.
- Only the last statement (iv) is false. Each of the others is true since at least one of its sub-statements is true.

Basic Logical Operators

- **Negation:** Corresponds to English word “not”
- Given any proposition p , another proposition, called the negation of p , can be formed by writing “It is not true that . . .” or “It is false that. . .” before p or, if possible, by inserting in p the word “not.” Symbolically: $\neg p$
- Definition: If p is true, then $\neg p$ is false; and if p is false, then $\neg p$ is true.

p	$\neg p$
T	F
F	T

Basic Logical Operators (Negation)

(*a*1) Ice floats in water.

(*a*2) It is false that ice floats in water.

(*a*3) Ice does not float in water.

(*b*1) $2 + 2 = 5$

(*b*2) It is false that $2 + 2 = 5$.

(*b*3) $2 + 2 \neq 5$

(*a*2) and (*a*3) are each the negation of (*a*1); and (*b*2) and (*b*3) are each the negation of (*b*1). Since (*a*1) is true, (*a*2) and (*a*3) are false; and since (*b*1) is false, (*b*2) and (*b*3) are true.

CS636 Formal Methods

Topics Covered

Propositional Logic

References

- All the contents of this video lecture and PPT slides are taken from:

Theory and Problems of DISCRETE MATHEMATICS, Third Edition, by SEYMOUR LIPSCHUTZ and MARC LARS LIPSON

PROPOSITIONS AND TRUTH TABLES

- Let $P(p, q, \dots)$ denote an expression constructed from logical variables $p, q, \dots$, and the logical connectives $\wedge, \vee$, and $\neg$, Such an expression $P(p, q, \dots)$ will be called a proposition.
- The truth value of a proposition is known once the truth value of each of its variables is known.

How to construct a Truth Table

- Consider, for example, the proposition $\neg(p \wedge \neg q)$. Following is the truth table of $\neg(p \wedge \neg q)$

p	q	$\neg q$	$p \wedge \neg q$	$\neg(p \wedge \neg q)$
T	T	F	F	T
T	F	T	T	F
F	T	F	F	T
F	F	T	F	T

- First columns of the table are for the variables $p, q, \dots$ and that there are enough rows in the table, to allow for all possible combinations of T and F for these variables. (For 2 variables, as above, 4 rows are necessary; for 3 variables, 8 rows are necessary; and, in general, for n variables, 2^n rows are required.)

How to construct a Truth Table

p	q	$\neg q$	$p \wedge \neg q$	$\neg(p \wedge \neg q)$
T	T	F	F	T
T	F	T	T	F
F	T	F	F	T
F	F	T	F	T

- There is then a column for each “elementary” stage of the construction of the proposition, the truth value at each step being determined from the previous stages by the definitions of the connectives $\wedge$, $\vee$, $\neg$.
- Finally we obtain the truth value of the proposition, which appears in the last column.

How to construct a Truth Table (alternate method)

p	q	$\neg$	$(p$	$\wedge$	$\neg$	$q)$
T	T					
T	F					
F	T					
F	F					
Step						

p	q	$\neg$	$(p$	$\wedge$	$\neg$	$q)$
T	T		T		F	T
T	F		T		T	F
F	T		F		F	T
F	F		F		T	F
Step			1		2	1

(a)

p	q	$\neg$	$(p$	$\wedge$	$\neg$	$q)$
T	T		T	F	F	T
T	F		T	T	T	F
F	T		F	F	F	T
F	F		F	F	T	F
Step			1	3	2	1

(b)

p	q	$\neg$	$(p$	$\wedge$	$\neg$	$q)$
T	T	T	T	F	F	T
F	T	F	T	T	T	F
F	F	T	F	F	F	T
F	F	T	F	F	T	F
Step		4	1	3	2	1

(c)

p	q	$\neg(p \wedge \neg q)$
T	T	T
T	F	F
F	T	T
F	F	T

LOGICAL EQUIVALENCE

- Two propositions $P(p, q, \dots)$ and $Q(p, q, \dots)$ are said to be logically equivalent, or simply equivalent or equal, if they have identical truth tables.
- Denoted by: $P(p, q, \dots) \equiv Q(p, q, \dots)$
- $\neg(p \wedge q) \equiv \neg p \vee \neg q$

p	q	$p \wedge q$	$\neg(p \wedge q)$
T	T	T	F
T	F	F	T
F	T	F	T
F	F	F	T

(a) $\neg(p \wedge q)$

p	q	$\neg p$	$\neg q$	$\neg p \vee \neg q$
T	T	F	F	F
T	F	F	T	T
F	T	T	F	T
F	F	T	T	T

(b) $\neg p \vee \neg q$

LOGICAL EQUIVALENCE

- $\neg(p \wedge q) \equiv \neg p \vee \neg q$
- Let p be “Roses are red” and q be “Violets are blue.”
- Let S be the statement: “It is not true that roses are red and violets are blue.”
- Then $S = \neg(p \wedge q)$.
- However, as noted above, $\neg(p \wedge q) \equiv \neg p \vee \neg q$. Accordingly, S has the same meaning as the statement:
- “Roses are not red, or violets are not blue.”

CS636 Formal Methods

Topics Covered

Propositional Logic

References

- All the contents of this video lecture and PPT slides are taken from:

Theory and Problems of DISCRETE MATHEMATICS, Third Edition, by SEYMOUR LIPSCHUTZ and MARC LARS LIPSON

Laws of the algebra of propositions

Idempotent laws:	(1a) $p \vee p \equiv p$	(1b) $p \wedge p \equiv p$
Associative laws:	(2a) $(p \vee q) \vee r \equiv p \vee (q \vee r)$	(2b) $(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$
Commutative laws:	(3a) $p \vee q \equiv q \vee p$	(3b) $p \wedge q \equiv q \wedge p$
Distributive laws:	(4a) $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$	(4b) $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$
Identity laws:	(5a) $p \vee F \equiv p$ (6a) $p \vee T \equiv T$	(5b) $p \wedge T \equiv p$ (6b) $p \wedge F \equiv F$
Involution law:	(7) $\neg\neg p \equiv p$	
Complement laws:	(8a) $p \vee \neg p \equiv T$ (9a) $\neg T \equiv F$	(8b) $p \wedge \neg p \equiv F$ (9b) $\neg F \equiv T$
DeMorgan's laws:	(10a) $\neg(p \vee q) \equiv \neg p \wedge \neg q$	(10b) $\neg(p \wedge q) \equiv \neg p \vee \neg q$

CS636 Formal Methods

Topics Covered

Propositional Logic

References

- All the contents of this video lecture and PPT slides are taken from:
- Formal Software Development From VDM to Java, by Quentin Charatan and Aaron Kans

The implication operator

- In defining an implication operator we attempt to give meaning to the expression P implies Q .
- The implication operator is represented by the symbol $\Rightarrow$.
- The statement P implies Q is therefore represented by: $P \Rightarrow Q$
- An alternative way of expressing implication is *if P then Q* .
- Thus if P represented the statement *It is rainy day* and Q represented the statement *I go the outdoors* then $P \Rightarrow Q$ would represent the statement *If it is rainy day I go the outdoors*.

The implication operator (Truth table)

P	Q	$P \Rightarrow Q$
T	T	T
T	F	F
F	T	T
F	F	T

The equivalence operator

- It is represented by the symbol $\Leftrightarrow$.
- It states: if P is TRUE then Q is TRUE, otherwise Q is FALSE; in other words, P is equivalent to Q, which is represented by: $P \Leftrightarrow Q$

P	Q	$P \Leftrightarrow Q$
T	T	T
T	F	F
F	T	F
F	F	T

The exclusive or operator

- The 'or' operator evaluates to TRUE if either of p or q or both are true.
- The natural language 'or' usually implies that only one or other of the statements is TRUE but not both.
- This corresponds to the logical operator known as exclusive or (sometimes referred to as xor), which is represented by the symbol: $\oplus$.

P	Q	$P \oplus Q$
T	T	F
T	F	T
F	T	T
F	F	F

CS636 Formal Methods

Topics Covered

Propositional Logic

References

- All the contents of this video lecture and PPT slides are taken from:

Theory and Problems of DISCRETE MATHEMATICS, Third Edition, by SEYMOUR LIPSCHUTZ and MARC LARS LIPSON

TAUTOLOGIES AND CONTRADICTIONS

- Some propositions $P(p, q, \dots)$ contain only T in their evaluation or, in other words, they are true for any truth values of their variables. Such propositions are called tautologies.
- Analogously, a proposition $P(p, q, \dots)$ is called a contradiction if it contains only F in its evaluation or, in other words, if it is false for any truth values of its variables.

p	$\neg p$	$p \vee \neg p$
T	F	T
F	T	T

p	$\neg p$	$p \wedge \neg p$
T	F	F
F	T	F

ARGUMENTS

- An argument is an assertion that a given set of propositions $P_1, P_2, \dots, P_n$, yields another proposition Q .
- An argument provides support or evidence in favor of one of the others.
- An argument $P_1, P_2, \dots, P_n \vdash Q$ is said to be valid if Q is TRUE whenever all the premises $P_1, P_2, \dots, P_n$ are true.
- Eg: valid argument

$p, p \rightarrow q \vdash q$

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

ARGUMENTS

- Now the propositions $P_1, P_2, \dots, P_n$ are true simultaneously if and only if the proposition $P_1 \wedge P_2 \wedge \dots \wedge P_n$ is true.
- Thus the argument $P_1, P_2, \dots, P_n \vdash Q$ is valid if and only if Q is true whenever $P_1 \wedge P_2 \wedge \dots \wedge P_n$ is true, or, equivalently, if the proposition $(P_1 \wedge P_2 \wedge \dots \wedge P_n) \rightarrow Q$ is a tautology.

ARGUMENTS

Lets determine the validity of the following argument:

$$p \rightarrow q, \neg q \vdash \neg p.$$

p	q	$[(p \rightarrow q) \wedge \neg q] \rightarrow \neg p$								
T	T	T	T	T	F	F	T	T	F	T
T	F	T	F	F	F	T	F	T	F	T
F	T	F	T	T	F	F	T	T	T	F
F	F	F	T	F	T	T	F	T	T	F
Step		1	2	1	3	2	1	4	2	1

ARGUMENTS

- An argument which is not valid is called fallacy.
- Eg: fallacy argument
- $p \rightarrow q, q \vdash p$

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

ARGUMENTS

Show that the following argument is a fallacy: $p \rightarrow q, \neg p \vdash \neg q$.

p	q	$p \rightarrow q$	$\neg p$	$(p \rightarrow q) \wedge \neg p$	$\neg q$	$[(p \rightarrow q) \wedge \neg p] \rightarrow \neg q$
T	T	T	F	F	F	T
T	F	F	F	F	T	T
F	T	T	T	T	F	F
F	F	T	T	T	T	T

CS636 Formal Methods

Topics Covered

Propositional Logic

References

- All the contents of this video lecture and PPT slides are taken from:
- Formal Software Development From VDM to Java, by Quentin Charatan and Aaron Kans

THREE-VALUED LOGIC

- Classical logic assumes that all expressions evaluate to TRUE or FALSE.
- In reality, this is not always the case when evaluating an expression, because sometimes an expression can be undefined
 - For example, the expression $0/0$.
- Undefined terms are very common in programming situations – for example, when a variable is first declared and has not yet been assigned a value.
- In this system of logic a proposition could have the value TRUE, FALSE or UNDEFINED.

AND Operator

P	Q	$P \wedge Q$
T	T	T
T	F	F
T	UNDEFINED	UNDEFINED
F	T	F
F	F	F
F	UNDEFINED	F
UNDEFINED	T	UNDEFINED
UNDEFINED	F	F
UNDEFINED	UNDEFINED	UNDEFINED

OR Operator

<i>P</i>	<i>Q</i>	<i>P ∨ Q</i>
T	T	T
T	F	T
T	UNDEFINED	T
F	T	T
F	F	F
F	UNDEFINED	UNDEFINED
UNDEFINED	T	T
UNDEFINED	F	UNDEFINED
UNDEFINED	UNDEFINED	UNDEFINED

IMPLICATION Operator

P	Q	$P \Rightarrow Q$
T	T	T
T	F	F
T	UNDEFINED	UNDEFINED
F	T	T
F	F	T
F	UNDEFINED	T
UNDEFINED	T	T
UNDEFINED	F	UNDEFINED
UNDEFINED	UNDEFINED	UNDEFINED

Equivalence Operator

P	Q	$P \Leftrightarrow Q$
T	T	T
T	F	F
T	UNDEFINED	UNDEFINED
F	T	F
F	F	T
F	UNDEFINED	UNDEFINED
UNDEFINED	T	UNDEFINED
UNDEFINED	F	UNDEFINED
UNDEFINED	UNDEFINED	UNDEFINED

Exclusive OR Operator

P	Q	$P \oplus Q$
T	T	F
T	F	T
T	UNDEFINED	UNDEFINED
F	T	T
F	F	F
F	UNDEFINED	UNDEFINED
UNDEFINED	T	UNDEFINED
UNDEFINED	F	UNDEFINED
UNDEFINED	UNDEFINED	UNDEFINED

Negation Operator

P	$\neg P$
T	F
F	T
UNDEFINED	UNDEFINED

CS636 Formal Methods

Topics Covered

Predicate Logic

References

- All the contents of this video lecture and PPT slides are taken from:
- Formal Software Development From VDM to Java, by Quentin Charatan and Aaron Kans

Predicate Logic

- One of the limitations with the propositional logic is that, while it allows us to argue about individual values, it does not give us the ability to argue about sets of values.
- A set is any well-defined, unordered, collection of objects.
 - The set of whole numbers from 1 to 10
 - The set of the days of the week
- In mathematics, we often represent a set elements by lower-case letters. For example:
- $A = \{s, d, f, h, k\}$
- $B = \{a, b, c, d, e, f\}$
- The symbol $\in$ means 'is an element of'. Therefore the statement 'd is an element of A' is written: $d \in A$

Predicate Logic

- For the purpose of reasoning about sets of values, a more powerful tool than the propositional logic has been devised, namely the predicate logic.
- A predicate is a truth-valued expression containing free variables. These allow the expression to be evaluated by giving different values to the variables.
- Once the variables are evaluated they are said to be bound.

Example of predicates

- Predicates can be named with either a single letter, or with a word that expresses the meaning of the predicate; the variables are placed in brackets after the name.
- This is made clear in the following examples:
 - $C(x)$: x is a cat
 - $\text{Studies}(x,y)$: x studies y
 - $\text{Prime}(n)$: n is a prime number
- A statement such as $C(x)$ can be read C of x.
- Predicates such as those above do not yet have a value – they only have a value when the variables themselves are given a value.

Binding variables

- Predicates do not yet have a value – they only have a value when the variables themselves are given a value.
- Two ways:
 - Substitution
 - Quantification

Binding (Substitution)

- For example, using the previous three predicates:
- $C(\text{Simba})$: Simba is a cat
- $\text{Studies}(\text{Ali}, \text{physics})$: Ali studies physics
- $\text{Prime}(3)$: 3 is a prime number

The above expressions now have a value of TRUE or FALSE.

Binding (Quantification)

- A quantifier is a mechanism for specifying an expression about a set of values. There are three quantifiers that we can use, each with its own symbol:
- Universal Quantifier
- Existential Quantifier
- Unique Existential Quantifier

Quantification

- The universal quantifier $\forall$:
- This quantifier enables a predicate to make a statement about all the elements in a particular set. For example, if $M(x)$ is the predicate x chases mice, we could write:
- $\forall x \in \text{Cats} \bullet M(x)$
- This reads *For all the x which are members of the set Cats, x chases mice*, or, more simply, *All cats chase mice*.

Quantification

- – The existential quantifier $\exists$
- In this case, a statement is made about whether or not at least one element of a set meets a particular criterion.
- For example, if, as above, $P(n)$ is the predicate *n is a prime number*, then we could write:
- $\exists n \in \mathbb{N} \bullet P(n)$
- This reads *There exists an n in the set of natural numbers such that n is a prime number*, or, put another way, *There exists at least one prime number in the set of natural numbers*.

Quantification

- –The unique existential quantifier $\exists!$
- This quantifier modifies a predicate to make a statement about whether or not precisely one element of a set meets a particular criterion.
- For example, if $G(x)$ is the predicate *x is green*, we could write
- $\exists!x \in \text{Cats} \bullet G(x)$
- This would mean *There is one and only one cat that is green.*

CS636 Formal Methods

Topics Covered: 058 – 065

**Introduction to Specification in VDM-SL
(Part 1)**

Intoduction to Specification in VDM-SL

At the end of this chapter you should be able to:

- write a formal specification of a system in VDM-SL
- correlate the components of a UML class diagram with those of a VDM specification
- declare constants and specify functions to enhance the specification
- explain the use of a state invariant to place a global constraint on the system
- explain the purpose of the nil value in VDM

Formal Methods

Topic#058

**The Case Study: Requirements
Analysis**

The Case Study: Requirements Analysis

- The example we will use throughout this chapter will be that of an incubator, the temperature of which needs to be carefully controlled and monitored in order to provide the correct conditions for a particular biological experiment to be undertaken.
- We will specify the software needed to monitor and control the incubator temperature.

- In developing any software system the first stage in the process involves an analysis of the system and an initial statement of the requirements.
- It is very important, in any requirements definition, to be clear about the system boundaries, and we should make it clear here that in this initial version, control of the hardware lies outside of our system.
- In other words, for the time being we will be specifying a system that simply monitors the temperature of the incubator.

The Incubator - Case Study

- The hardware increments or decrements the temperature of the incubator in response to instructions (from someone or something outside of our system), and each time a change of one degree has been achieved, the software is informed of the change, which it duly records. However, safety requirements dictate that the temperature of the incubator must never be allowed to rise above 10 degree celsius, nor fall below -10 degree celsius.

Formal Methods

Topic#058

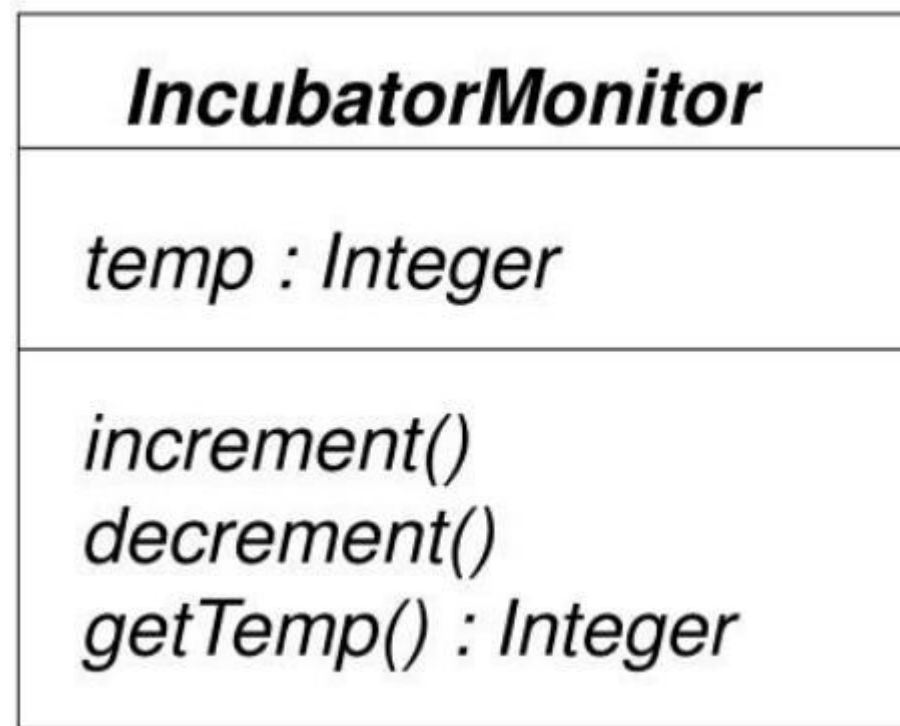
END!

Formal Methods

Topic#059

The UML Specification

The UML Specification



The Specification of the IncubatorMonitor Class

- In the simple system described in the previous slide, we can identify a single class, IncubatorMonitor.
- We have identified one attribute and three methods.
- The single attribute records the temperature of the system and will be of type integer.
- With regard to the methods, the first two do not involve any input or output (since they merely record an increase or decrease of one degree).
- The final method reads the value of the temperature, and therefore will output an integer.

Formal Methods

Topic#059

END!

Formal Methods

Topic#060

Specifying the State

Specifying the State

- The first thing we will consider for our formal specification is what is known as the state of the system.
- In VDM-SL the state refers to the permanent data that must be stored by the system, and which can be accessed by means of operations.

The VDM state refers to the permanent data stored by the system.

<i>IncubatorMonitor</i>
<i>temp : Integer</i>
<i>increment() decrement() getTemp() : Integer</i>

Intrinsic types available in VDM-SL

N: natural numbers (positive whole numbers)

N_1 : natural numbers excluding zero

Z: integers (positive and negative whole numbers)

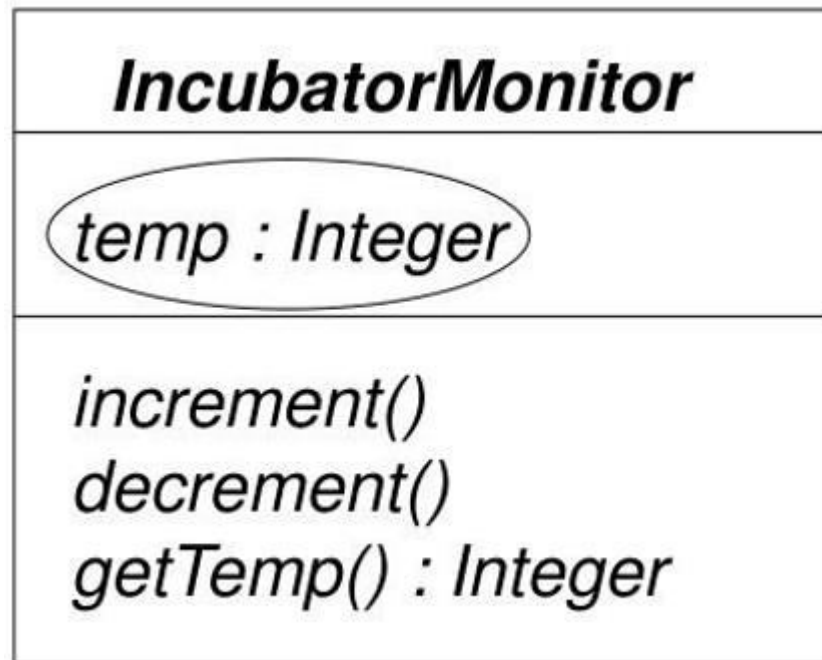
R: real numbers (positive and negative numbers that can include a fractional part)

B: boolean values (TRUE or FALSE)

Char: the set of alphanumeric characters

Specifying the state of Incubator Monitor System

UML



VDM-SL



Formal Methods

Topic#060

END!

Formal Methods

Topic#061

Specifying the Operations

Specifying the Operations

- We need to specify a number of operations that the system should be able to perform and by which means the data (that is the state) can be accessed.
- In VDM we tend to use the word operation, whereas in most object-oriented texts you will tend to see the word method.
- In VDM, operations by definition access the state in some way, either by reading or writing the data, or both.

Operations in VDM-SL

In VDM-SL an operation consists of four sections:

- the operation header
- the external clause
- the precondition
- the postcondition

We need to consider: an operation that records an increment in the temperature; an operation that records a decrement; and one that simply reads the value of the temperature.

<i>IncubatorMonitor</i>
<i>temp : Integer</i>
<i>increment()</i> <i>decrement()</i> <i>getTemp() : Integer</i>

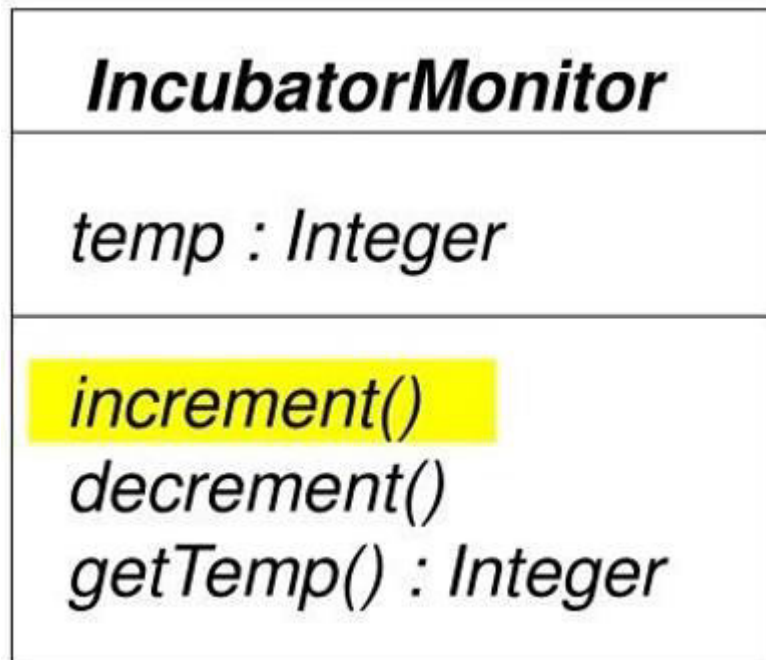
Each operation specified in VDM-SL as follows:



the operation header
the external clause
the precondition
the postcondition

Increment Operation

UML

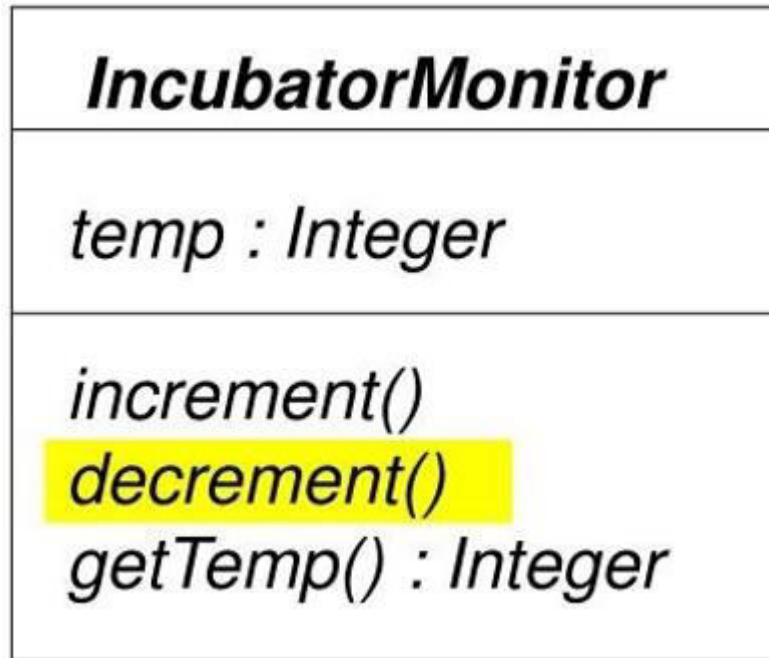


VDM-SL



Decrement Operation

UML

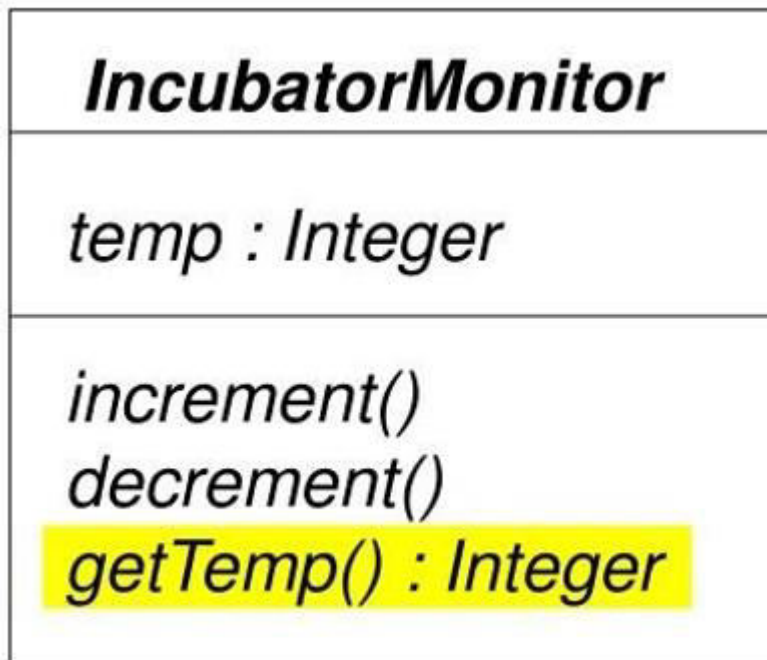


VDM-SL

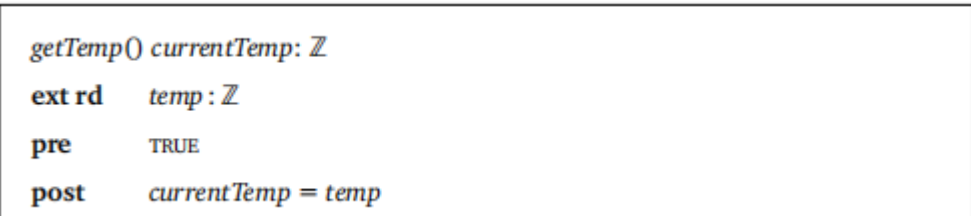
```
decrement()  
ext wr   temp :  $\mathbb{Z}$   
pre      temp > -10  
post     temp =  $\overline{\text{temp}} - 1$ 
```

getTemp Operation

UML



VDM-SL



Formal Methods

Topic#061

END!

Formal Methods

Topic#062

Declaring Constants

Declaring Constants

- As with many programming languages, it is possible in VDM-SL to specify constants.
- This is something that is not essential to any specification, but can greatly enhance its readability.
- It is done by using the keyword **values**, and the declaration would come immediately before the state definition.

- In the case of the IncubatorMonitor it would look like this:

values

MAX: $\mathbb{Z}$ = 10

MIN: $\mathbb{Z}$ = -10

- These values could then be used in our functions and operations, so, for example, the precondition of the decrement operation would now look like this:

pre *temp > MIN*

Formal Methods

Topic#062

END!

Formal Methods

Topic#063

Specifying Functions

What is a Function?

A function is a set of assignments from one set to another. Thus, the function receives an input value (or values) and maps this to an output value according to some rule – for example it could accept an integer and output the square of that integer, or it could accept the name of a person and output that person's telephone number.

Specifying a Function in VDM-SL

There are two ways in which we can specify a function in VDM-SL:

- Explicitly
- Implicitly

Explicit Function

- The style of this specification is algorithmic, and we explicitly define the method of transforming the inputs to the output. This is illustrated in the following very simple function that adds two numbers together:

```
add:  $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$   
add(x, y)  $\triangleq x + y$ 
```

- The first line is called the **function signature**.
- The second part is the **function definition**.

Implicit Function

- In this method, we use a pre- and postcondition in the same way as we described for operations – a function, of course, does not access the state variables. Here is the add function defined implicitly.

```
add( $x:\mathbb{R}, y:\mathbb{R}$ )  $z:\mathbb{R}$ 
```

```
pre      TRUE
```

```
post     $z = x + y$ 
```

Formal Methods

Topic#063

END!

Formal Methods

Topic#064

Specifying a State Invariant

Specifying a State Invariant

- You have seen that our requirements definition states that the temperature of the incubator must stay within the range -10 to +10 degree celsius. In VDM-SL there is a mechanism by which we can incorporate such a restriction into the specification of the state. This mechanism involves specifying a function known as a state **invariant**.
- By specifying such a function, we are creating a global constraint, rather than just a local constraint as we did with our preconditions.

- The invariant definition uses the keyword **inv**
- In previous topic you studied about function signature. So, in the case of an invariant function, *inv*, its signature will be:

inv: State $\rightarrow$ B

- The function maps a value of the state onto a boolean – either TRUE or FALSE; and by specifying such a function we are saying that the state variables must be such that the result of the function is TRUE.

- For the IncubatorMonitor system the invariant is specified as:

inv mk-IncubatorMonitor(t) $\triangleq MIN \leq t \leq MAX$

- After the keyword *inv*, we have the expression *mk-IncubatorMonitor(t)*, which effectively is the input to the *inv* function. This expression is itself a function, and is known as a make function (the *mk* is pronounced 'make').

Invariant gives the opportunity to think about the idea of mathematical proof and integrity checking in connection with software development.

Formal Methods

Topic#064

END!

Formal Methods

Topic#065

**Specifying an
Initialization Function**

Specifying an Initialization Function

- You may already have identified one of the shortcomings of the above specification, namely that we have not yet made any statement about what the value of the temperature should be when the system is first brought into being. It is all very well having operations that increment and decrement the temperature, but if we do not know what the initial value of the temperature was, then they are not very meaningful.
- This problem can be solved by specifying an **initialization function**, which is given the name **init**.
- This function is specified after the declaration of the invariant, and prescribes the conditions that the system must satisfy when it is first brought into being.

- Let us illustrate this with our IncubatorMonitor example. We will assume that the system works in the following way: when the incubator is turned on, its temperature is adjusted until a steady 5 degree celsius is obtained. At this point the software system is activated. Thus, our initialization function should state that when the system is first invoked, the temperature should be set to 5.
- We write the initialization function like this:

```
init mk-IncubatorMonitor(t)  $\Delta$  t = 5
```

It is very important to note that the initialization function – as with the operations must preserve the invariant.

Formal Methods

Topic#065

END!

CS636 Formal Methods

Topics Covered: 066 – 073

**Introduction to Specification in VDM-SL
(Part 2)**

Formal Methods

Topic#066

User-defined Types

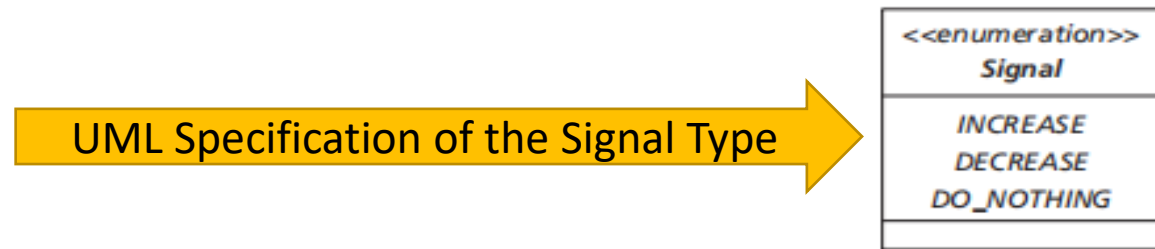
Components of VDM-SL

As we studied, the components of a complete VDM-SL specification are:

- Declaration of constants
- State definition
- Functions
- Operations

User-defined Types

- There is one more component to the specification, which must come at the beginning. This is the declaration of any user-defined types.



- User-defined type such as signal can be defined in a VDM specification.

- The **types** clause is the appropriate place to define new types.
- The *Signal* type is defined as follows:

types

Signal = <INCREASE> | <DECREASE> | <DO_NOTHING>

- Here we are defining a type by **type construction**. This form of type construction allows enumerated types to be specified formally in VDM-SL.
- Values such as <INCREASE>, <DECREASE> and <DO_NOTHING> are called **quote types**, and a type such as *Signal* is a **union of quote types** in VDM.
- A quote type defines a single value, and at the same time defines a type containing just that value. These quote types correspond to the values specified in the UML diagram at previous slide.
- Byconvention, **type names begin with an upper-case letter**.

Formal Methods

Topic#066

END!

Formal Methods

Topic#067

The nil Value

The nil Value

- It is common in the programming world for a value to be undefined. VDM-SL allows for this concept by including the possibility of a term or expression having the value **nil**, meaning that it is undefined.
- Of course, if we want to allow for this possibility, then we need to slightly modify the type of the variable. We do that by placing square brackets around the type name – for example [N] or [Z] – meaning that a variable of that type can take the value of nil.
- Effectively we are extending the type to include the **nil** value.

Formal Methods

Topic#067

END!

Formal Methods

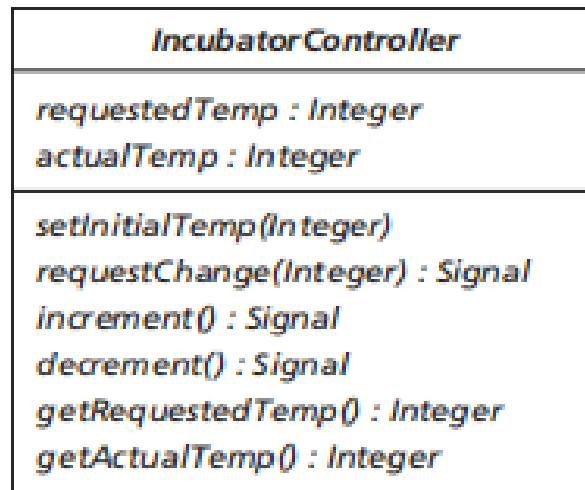
Topic#068

**Improving the Incubator
System**

Improving the Incubator System

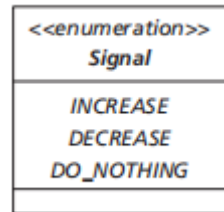
- Make the software more realistic.
- In our enhanced system, the software will not only record the current temperature of the system, but will also control the hardware.
- Our new system will also behave a bit more realistically in regard to the initial temperature of the incubator.

UML diagram for our new system



Specification of the IncubatorController

UML specification of the Signal type



UML specification of the Signal type

Formal Methods

Topic#068

END!

Formal Methods

Topic#069

Specifying the State of the IncubatorController System

Specifying the State of the IncubatorController System

There now need to be two components of the state:

- one to hold the actual temperature
- one to hold the temperature that has been requested

New state definition

```
state IncubatorController of  
  requestedTemp : [ $\mathbb{Z}$ ]  
  actualTemp : [ $\mathbb{Z}$ ]
```

Invariant comprising two conjuncts

inv *mk-IncubatorController* (*r*, *a*) $\triangleq$

$(MIN \leq r \leq MAX \vee r = \mathbf{nil}) \wedge (MIN \leq a \leq MAX \vee a = \mathbf{nil})$

Now there are two inputs to the make function.

inRange Function

- The purpose of this function is to check whether an integer value, *val*, is within the range *MIN* and *MAX* as defined earlier.
- You can see that the use of the equivalence connective ensures that the output is true if the input is in range, but is otherwise false.

inRange(val: Z) result: B

pre TRUE

post $result \Leftrightarrow MIN \leq val \leq MAX$

Use of inRange Function in the invariant

$$\text{inv } \text{mk-IncubatorController } (r, a) \triangleq (\text{inRange}(r) \vee r = \text{nil}) \wedge (\text{inRange}(a) \vee a = \text{nil})$$

This makes the invariant much more readable, and also has the advantage that we can re-use our function throughout the specification.

Initialization Function

- Since both the requested temperature and the actual temperature will be undefined at the point when the system is created, these should both be set to nil. Hence the initialization function is:

```
init mk-IncubatorController (r, a)  $\triangleq$  r = nil  $\wedge$  a = nil
```

Formal Methods

Topic#069

END!

Formal Methods

Topic#070

Specifying the Operations of the IncubatorController System

Specifying the Operations of the IncubatorController System

- It is going to be necessary to provide an operation that can be used to set the initial temperature of the system – this will be invoked by the hardware when the incubator has established a steady initial temperature.

setInitialTemp operation

- The operation is specified below:

```
setInitialTemp(tempIn :  $\mathbb{Z}$ )  
ext wr actualTemp : [ $\mathbb{Z}$ ]  
pre    inRange(tempIn)  $\wedge$  actualTemp = nil  
post   actualTemp = tempIn
```

requestChange operation

- The operation is specified as follows:

```
requestChange(tempIn :  $\mathbb{Z}$ ) signalOut : Signal
ext wr requestedTemp : [ $\mathbb{Z}$ ]
    rd actualTemp : [ $\mathbb{Z}$ ]
pre    inRange(tempIn)  $\wedge$  actualTemp  $\neq$  nil
post   requestedTemp = tempIn  $\wedge$ 
        (tempIn > actualTemp  $\wedge$  signalOut = <INCREASE>
          $\vee$  tempIn < actualTemp  $\wedge$  signalOut = <DECREASE>
          $\vee$  tempIn = actualTemp  $\wedge$  signalOut = <DO_NOTHING>)
```

increment operation

- The operation is specified as follows:

```
increment () signalOut : Signal
ext rd requestedTemp : [Z]
wr actualTemp : [Z]
pre    actualTemp < requestedTemp ∧ actualTemp ≠ nil ∧ requestedTemp ≠ nil
post   actualTemp = actualTemp + 1 ∧
       (actualTemp < requestedTemp ∧ signalOut = <INCREASE>
        ∨ actualTemp = requestedTemp ∧ signalOut = <DO_NOTHING>)
```

- The decrement operation is similar and does not require further explanation.

Read Operations

requested temperature

```
getRequestedTemp() currentRequested : [Z]  
ext rd    requestedTemp : [Z]  
pre      TRUE  
post    currentRequested = requestedTemp
```

actual temperature

```
getActualTemp() currentActual : [Z]  
ext rd    actualTemp : [Z]  
pre      TRUE  
post    currentActual = actualTemp
```

Formal Methods

Topic#070

END!

Formal Methods

Topic#071

A Standard Template for VDM-SL Specifications

Template for VDM-SL Specifications

- Generalized template for a VDM-SL specification:

```
types
  SomeType = ....
values
  constantName : ConstantType = someValue
state SystemName of
  attribute1 : Type
  :
  :
  attributen : Type

  inv mk-SystemName(i1:Type, ..., in:Type)  $\Delta$  Expression(i1, ..., in)
  init mk-SystemName(i1:Type, ..., in:Type)  $\Delta$  Expression(i1, ..., in)

end

functions
  specification of functions .....

operations
  specification of operations .....
```

- Not every clause would necessarily appear in every specification.

Formal Methods

Topic#071

END!

Formal Methods

Topic#072

Including Comments

Including Comments

- As with program code, the readability of a VDM-SL specification is greatly enhanced by the inclusion of comments. This is done by introducing the comment with the symbol `--`. A new line ends the comment.

```
types
Signal = <INCREASE> | <DECREASE> | <DO_NOTHING>

values
MAX:  $\mathbb{Z}$  = 10
MIN:  $\mathbb{Z}$  = -10

state IncubatorController of
  requestedTemp:  $\mathbb{Z}$ 
  actualTemp:  $\mathbb{Z}$ 
  -- both requested and actual temperatures must be in range or equal to nil
  inv mk-IncubatorController (r, a)  $\triangleq$  (inRange(r)  $\vee$  r = nil)  $\wedge$  (inRange(a)  $\vee$ 
                                                                    a = nil)

  -- both requested and actual temperatures are undefined when the system is initialized
  init mk-IncubatorController (r, a)  $\triangleq$  r = nil  $\wedge$  a = nil
end
```

Formal Methods

Topic#072

END!

Formal Methods

Topic#073

The Complete Specification of the IncubatorController System

Complete Specification of the IncubatorController System

```
types
Signal = <INCREASE> | <DECREASE> | <DO_NOTHING>

values
MAX: Z = 10
MIN: Z = -10

state IncubatorController of
  requestedTemp: [Z]
  actualTemp: [Z]
  -- both requested and actual temperatures must be in range or equal to nil
  inv mk-IncubatorController (r, a)  $\Delta$  (inRange(r)  $\vee$  r = nil)  $\wedge$  (inRange(a)  $\vee$  a = nil)

  -- both requested and actual temperatures are undefined when the system is initialized
  init mk-IncubatorController (r, a)  $\Delta$  r = nil  $\wedge$  a = nil
end

functions
inRange(val: Z) result: B
pre TRUE
post result  $\Leftrightarrow$  MIN  $\leq$  val  $\leq$  MAX

operations
-- an operation that records the initial temperature of the system
setInitialTemp(tempIn: Z)
ext wr actualTemp: [Z]
pre inRange(tempIn)  $\wedge$  actualTemp = nil
post actualTemp = tempIn

-- an operation that records the requested temperature and signals the hardware to increase
-- or decrease the temperature as appropriate
requestChange(tempIn: Z) signalOut: Signal
ext wr requestedTemp: [Z]
rd actualTemp: [Z]
pre inRange(tempIn)  $\wedge$  actualTemp  $\neq$  nil
post requestedTemp = tempIn  $\wedge$ 
  (tempIn > actualTemp  $\wedge$  signalOut = <INCREASE>
    $\vee$  tempIn < actualTemp  $\wedge$  signalOut = <DECREASE>
    $\vee$  tempIn = actualTemp  $\wedge$  signalOut = <DO_NOTHING>)

-- an operation that records a one degree increase and instructs the hardware
-- either to continue increasing the temperature or to stop
```

```
increment() signalOut: Signal
ext rd requestedTemp: [Z]
wr actualTemp: [Z]
pre actualTemp < requestedTemp  $\wedge$  actualTemp  $\neq$  nil  $\wedge$  requestedTemp  $\neq$  nil
post actualTemp =  $\overline{\text{actualTemp}}$  + 1  $\wedge$ 
  (actualTemp < requestedTemp  $\wedge$  signalOut = <INCREASE>
    $\vee$  actualTemp = requestedTemp  $\wedge$  signalOut = <DO_NOTHING>)

-- an operation that records a one degree decrease and instructs the hardware
-- either to continue decreasing the temperature or to stop
decrement() signalOut: Signal
ext rd requestedTemp: [Z]
wr actualTemp: [Z]
pre actualTemp > requestedTemp  $\wedge$  actualTemp  $\neq$  nil  $\wedge$  requestedTemp  $\neq$  nil
post actualTemp =  $\overline{\text{actualTemp}}$  - 1  $\wedge$ 
  (actualTemp > requestedTemp  $\wedge$  signalOut = <DECREASE>
    $\vee$  actualTemp = requestedTemp  $\wedge$  signalOut = <DO_NOTHING>)

getRequestedTemp() currentRequested: [Z]
ext rd requestedTemp: [Z]
pre TRUE
post currentRequested = requestedTemp

getActualTemp() currentActual: [Z]
ext rd actualTemp: [Z]
pre TRUE
post currentActual = actualTemp
```

Formal Methods

Topic#073

END!

CS636 Formal Methods

Topics Covered

Sets

Sets for System Modelling

- Many systems deal with data collections
- *For such data collections, VDM-SL provides a number of collection types*
 - *Sets*
 - *Sequences*

Sets for System Modelling

- A set is an unordered collection of objects in which repetition is not significant.
- *A collection of objects that are considered unique, and in which ordering is unimportant, the set type is a good candidate.*
- *A collection of patients registered on the books of a doctor's surgery.*

Sets in VDM-SL

- The type constructor **–set** is appended to the type associated with the elements of the set.
- **aNumber: $\mathbb{N}$**
- **someNumbers: $\mathbb{N}$ -set** ($\mathbb{N}$ means Natural Numbers)
- **someOtherNumbers: $\mathbb{Z}$ -set**

Set Declaration

types

Day =<MON> | <TUE> | <WED> | <THU> | <FRI> | <SAT> | <SUN>

importantDays: Day-set

Defining sets

```
someNumbers = {2, 4, 28, 19, 10}  
importantDays = {<FRI>, <SAT>, <SUN>}
```

```
someNumbers = {28, 2, 10, 4, 19}  
importantDays = {<SUN>, <FRI>, <SAT>}
```

```
someNumbers = {28, 2, 10, 2, 4, 19, 2}  
importantDays = {<SUN>, <FRI>, <SAT>, <FRI>, <FRI>}
```

Sets in VDM-SL

- **Subranges:** used when a set of continuous integers is required

```
someRange = {5,...,15}
```

- ***A subrange returns the set of all numbers from the first to the last number inclusive.***

```
someRange = {5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}
```

Sets in VDM-SL

- **When the second number in the range is smaller than the first, the empty set is returned. The empty set is represented in VDM-SL as an empty pair of braces**

$\{7, \dots, 6\} = \{\}$

$\{6, \dots, 4\} = \{\}$

Sets in VDM-SL

- **Comprehension:** A set is defined by means of an expression and/or a test that each element in the set must satisfy.

```
someNumbers = {x | x ∈ {2,...,6} • isEven(x)}
```

Sets in VDM-SL

- *In general, set comprehension takes the following form:*

```
someSet = {expression (x) | binding (x) • test(x)}
```

- *“|” means “Such that”*
- *“binding” & “test” determine acceptable values for the free variable (*x* in this case).*
- *The bullet (•) separates the binding from the test.*
- *This free variable is then used in the expression to determine the final value of elements in the new set.*

Sets in VDM-SL

- ***expressions can be complex:***

```
someOtherNumbers = {x2 | x ∈ {2,...,6}}
```

```
someOtherNumbers = {22, 32, 42, 52, 62}  
                  = {4, 9, 16, 25, 36}
```

- ***type can be used in the binding instead of a set. When using a type the ‘is of type’ symbol (:) is to be used***

```
smallNumbers {x | x: ℕ • 1 ≤ x ≤ 10}
```

Finite and Infinite sets

- ❑ *Sets having infinite elements e.g. set of all real numbers are described as infinite sets*
- ❑ *As sets in VDM-SL need to be implemented in machine which is not possible, so in VDM-SL, sets are finite*

```
smallNumbers = {x | x:  $\mathbb{Z}$  • x < 0}
```

- ❑ *The above set is not permissible in VDM-SL*

Thank You

CS636 Formal Methods

Topics Covered

Set Operations

Set Operations

- ***Three operations that take two sets as input and return a new set as output***
 - ***Set union***
 - ***Set intersection***
 - ***Set difference***
- ***In each case, the types of elements in each set are assumed to be the same.***

Set Operations: Union

- The union of two sets, j and k returns a set that contains all the elements of the set j and all the elements of the set k .

$J \cup K$ (J Union K)

- Example:

```
if      j = {<MON>, <TUE>, <WED>, <SUN>}
and     k = {<MON>, <FRI>, <TUE>}
then    j ∪ k = {<MON>, <TUE>, <WED>, <SUN>, <FRI>}
```

Set Operations: Intersection

- The intersection of two sets j and k returns a set that contains all the elements that are common to both j and k .

$J \cap K$ (J intersection K)

- Example:

```
if      j = {<MON>, <TUE>, <WED>, <SUN>}  
and     k = {<MON>, <FRI>, <TUE>}  
then    j ∩ k = {<MON>, <TUE>}
```

Set Operations: Difference

- The difference of j and k is the set that contains all the elements that belong to j but do not belong to k .

$J \setminus K$ (J difference K)

- Example:

if $j = \{\text{<MON>, <TUE>, <WED>, <SUN>}\}$
and $k = \{\text{<MON>, <FRI>, <TUE>}\}$
then $j \setminus k = \{\text{<WED>, <SUN>}\}$

Incorrect: $\{\text{<MON>, <TUE>, <WED>}\} \setminus \text{<TUE>}$ Correct: $\{\text{<MON>, <TUE>, <WED>}\} \setminus \{\text{<TUE>}\}$

Set Operations: Equal

- Two sets i and j are equal if they have same elements

$$i = \{a, b, c\} \quad j = \{b, a, c\}$$

- Here x and y are not equal:

$$x = \{a, b, c\} \quad j = \{b, a, c, d\}$$

Set Operations

- A set containing just a single element is referred to as a singleton set. Example:

$$J = \{<SUN>\}$$

- **Commutative operators:** same result regardless of the order of the parameters

$$\begin{aligned} J \cap K &= K \cap J \\ J \cup K &= K \cup J \end{aligned}$$

Set Operations

- ❑ **Set difference is not commutative: the order of the parameters is significant to the result.**

$$j \setminus k \neq k \setminus j$$

$j = \{\langle \text{MON} \rangle, \langle \text{TUE} \rangle, \langle \text{WED} \rangle, \langle \text{SUN} \rangle\}$

$k = \{\langle \text{MON} \rangle, \langle \text{FRI} \rangle, \langle \text{TUE} \rangle\}$

$j \setminus k = \{\langle \text{WED} \rangle, \langle \text{SUN} \rangle\}$

$k \setminus j = \{\langle \text{FRI} \rangle\}$

Set Operations: Membership

- This operator checks whether or not a particular element is present in a particular set.

Symbol: $\in$

- Non-Membership:

Symbol: $\notin$

Example:

If $A = \{1, 3, 5, 7\}$, then $1 \in A$, but $2 \notin A$

Set Operations: Subsets

□ It returns **TRUE** if all the elements in the first set are also elements of the second set and **FALSE** otherwise

□ Example:

$\{a, d, e\} \subseteq \{a, b, c, d, e, f\}$... *TRUE*

$\{a, b, c, d, e, f\} \subseteq \{a, d, e\}$... *FALSE*

□ Returns **TRUE** if both sets are equal

$\{a, d, e\} \subseteq \{d, a, e\}$... *TRUE*

Set Operations: Subsets

□ It returns **TRUE** if all the elements in the first set are also elements of the second set and **FALSE** otherwise

□ Example:

$\{a, d, e\} \subseteq \{a, b, c, d, e, f\}$... *TRUE*

$\{a, b, c, d, e, f\} \subseteq \{a, d, e\}$... *FALSE*

□ Returns **TRUE** if both sets are equal

$\{a, d, e\} \subseteq \{d, a, e\}$... *TRUE*

Set Operations: Cardinality

- ❑ The cardinality operator of VDM-SL (**card**) returns the number of elements in a given set.

```
card {7, 2, 12} = 3  
card {4,...,10} = 7  
card {} = 0
```

- ❑ As repetition is not significant in sets, repeated elements are counted only once when calculating the cardinality

```
card {7, 2, 12, 2, 2} = card {7, 2, 12} = 3
```

Thank You

CS636 Formal Methods

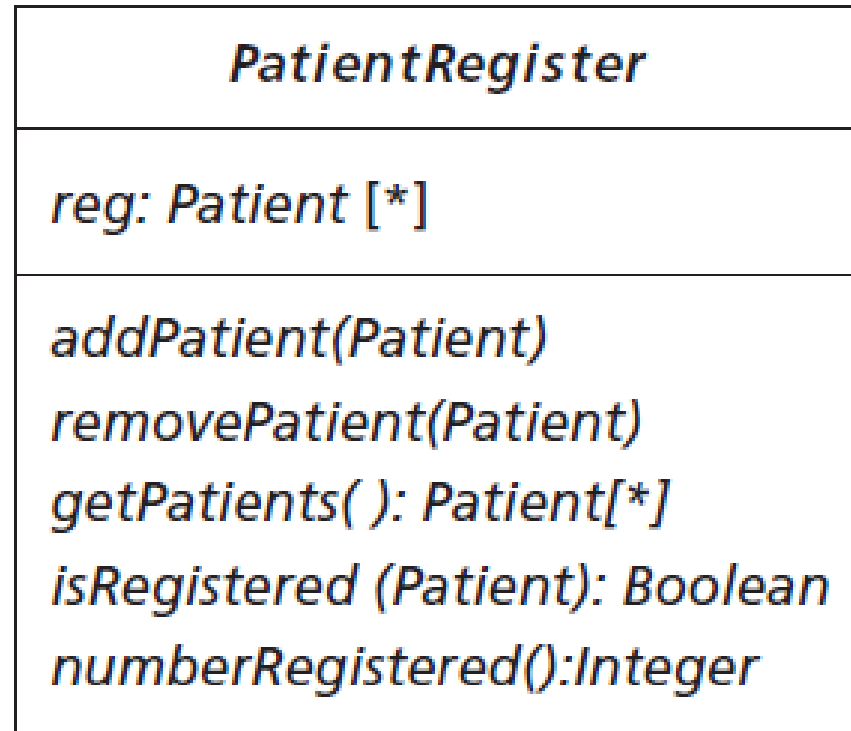
Topics Covered

Patient Register Class

Patient Surgery Clinic

- ❑ **At maximum 200 patients can register**
- ❑ ***Patients can be added/Removed***
- ❑ ***List of Patients and Number of Patients should be returned***
- ❑ ***Check if a patient is registered or not***

UML representation



Implementation in VDM-SL

types

Patient TOKEN

values

LIMIT: $\mathbb{N}$ = 200

state *PatientRegister* of

reg: *Patient-set*

PatientRegister

reg: *Patient* [*]

addPatient(*Patient*)

removePatient(*Patient*)

getPatients(): *Patient* [*]

isRegistered (*Patient*): *Boolean*

numberRegistered():*Integer*

Implementation in VDM-SL

types

Patient = TOKEN

values

LIMIT: $\mathbb{N}$ = 200

state *PatientRegister* of

reg: *Patient*-set

PatientRegister

reg: *Patient* [*]

addPatient(*Patient*)

removePatient(*Patient*)

getPatients(): *Patient* [*]

isRegistered (*Patient*): *Boolean*

numberRegistered():*Integer*

Implementation in VDM-SL

inv mk-PatientRegister (r) $\triangle$ card r $\leq$ LIMIT

inv mk-PatientRegister (r) $\triangle$ r = {}

<i>PatientRegister</i>
<i>reg: Patient [*]</i>
<i>addPatient(Patient)</i> <i>removePatient(Patient)</i> <i>getPatients(): Patient[*]</i> <i>isRegistered (Patient): Boolean</i> <i>numberRegistered():Integer</i>

Implementation in VDM-SL

types

Patient = TOKEN

values=

LIMIT: $\mathbb{N}$ = 200

state *PatientRegister* of

reg: *Patient-set*

inv *mk-PatientRegister* (*r*) $\triangle \text{card } r \leq \textit{LIMIT}$

inv *mk-PatientRegister* (*r*) $\triangle r = \{\}$

end

PatientRegister

reg: *Patient* [*]

addPatient(*Patient*)

removePatient(*Patient*)

getPatients(): *Patient* [*]

isRegistered (*Patient*): *Boolean*

numberRegistered():*Integer*

Implementation in VDM-SL

```
addPatient (patientIn: Patient)  
ext wr reg: Patient-set  
pre patientIn  $\notin$  reg  $\wedge$  card reg < LIMIT  
post reg =  $\overline{reg} \cup \{patientIn\}$ 
```

<i>PatientRegister</i>
<i>reg</i> : <i>Patient</i> [*]
<i>addPatient</i> (<i>Patient</i>) <i>removePatient</i> (<i>Patient</i>) <i>getPatients</i> (): <i>Patient</i> [*] <i>isRegistered</i> (<i>Patient</i>): <i>Boolean</i> <i>numberRegistered</i> (): <i>Integer</i>

Implementation in VDM-SL

```
removePatient (patientIn: Patient)  
ext wr reg: Patient-set  
pre patientIn  $\in$  reg  
post reg =  $\overline{reg} \setminus \{patientIn\}$ 
```

<i>PatientRegister</i>
<i>reg</i> : <i>Patient</i> [*]
<i>addPatient</i> (<i>Patient</i>) <i>removePatient</i> (<i>Patient</i>) <i>getPatients</i> (): <i>Patient</i> [*] <i>isRegistered</i> (<i>Patient</i>): <i>Boolean</i> <i>numberRegistered</i> (): <i>Integer</i>

Implementation in VDM-SL

```
getPatients ( ) output: Patient-set  
ext rd reg: Patient-set  
pre TRUE  
post output = reg
```

<i>PatientRegister</i>
<i>reg: Patient [*]</i>
<i>addPatient(Patient)</i> <i>removePatient(Patient)</i> <i>getPatients(): Patient[*]</i> <i>isRegistered (Patient): Boolean</i> <i>numberRegistered():Integer</i>

Implementation in VDM-SL

```
isRegistered (patientIn: Patient) query:  $\mathbb{B}$   
ext rd reg: Patient-set  
pre TRUE  
post query  $\Leftrightarrow$  patientIn  $\in$  reg
```

<i>PatientRegister</i>
<i>reg</i> : <i>Patient</i> [*]
<i>addPatient</i> (<i>Patient</i>) <i>removePatient</i> (<i>Patient</i>) <i>getPatients</i> (): <i>Patient</i> [*] <i>isRegistered</i> (<i>Patient</i>): <i>Boolean</i> <i>numberRegistered</i> (): <i>Integer</i>

Implementation in VDM-SL

```
numberRegistered ( ) total:  $\mathbb{N}$   
ext rd reg: Patient-set  
pre TRUE  
post total = card reg
```

<i>PatientRegister</i>
<i>reg</i> : <i>Patient</i> [*]
<i>addPatient</i> (<i>Patient</i>) <i>removePatient</i> (<i>Patient</i>) <i>getPatients</i> (): <i>Patient</i> [*] <i>isRegistered</i> (<i>Patient</i>): <i>Boolean</i> <i>numberRegistered</i> (): <i>Integer</i>

Thank You

CS636 Formal Methods

Topics Covered

**The Airport Class and Implementation in
VDM-SL**

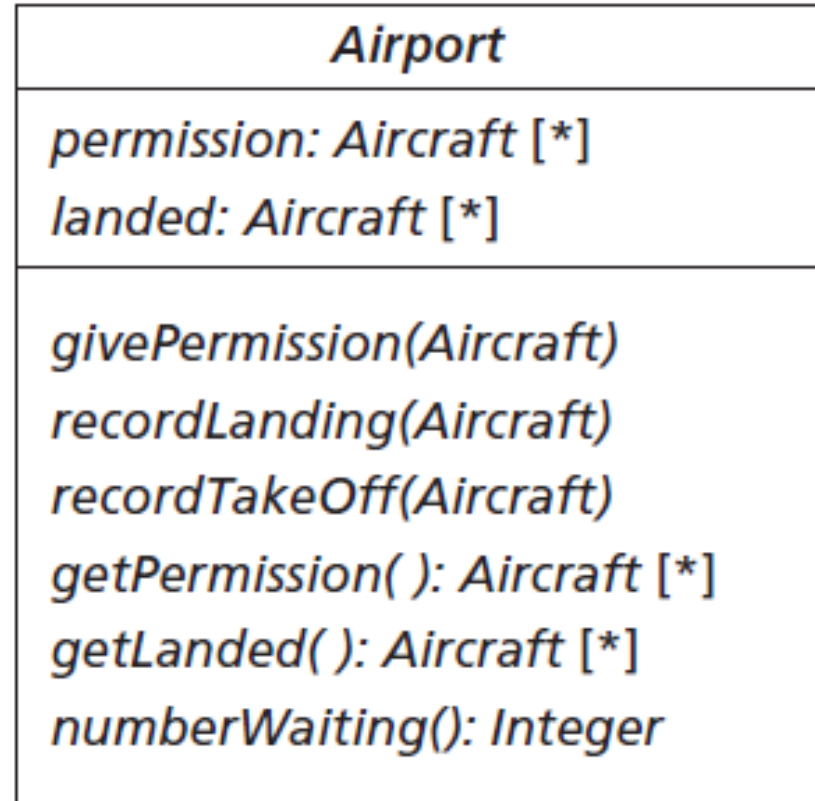
Airport Class

- ❑ **Consider a system that keeps track of aircraft that are allowed to land at a particular airport. Aircraft must apply for permission to land at the airport prior to landing. When an aircraft arrives to land at the airport it should only have done so if it had previously been given permission. When an aircraft leaves the airport its permission to land is also removed.**

Airport Class

- ❑ **givePermission:** records the fact that an aircraft has been granted permission to land at the airport.
- ❑ **recordLanding:** records an aircraft as having landed at the airport.
- ❑ **recordTakeOff:** records an aircraft as having taken off from the airport.
- ❑ **getPermission:** returns the aircrafts currently recorded as having permission to land
- ❑ **getLanded:** returns the aircrafts currently recorded as having landed
- ❑ **numberWaiting:** returns the number of aircrafts granted permission to land but not yet landed.

UML representation



Implementation in VDM-SL

types

Aircraft = TOKEN

state *Airport* of

permission: *Aircraft*-set

landed: *Aircraft* -set

inv *mk-Airport*(*p*, *l*) $\triangleq l \subseteq p$

init *mk-Airport* (*p*, *l*) $\triangleq p = \{ \} \wedge l = \{ \}$

end

Airport

permission: *Aircraft* [*]

landed: *Aircraft* [*]

givePermission(*Aircraft*)

recordLanding(*Aircraft*)

recordTakeOff(*Aircraft*)

getPermission(): *Aircraft* [*]

getLanded(): *Aircraft* [*]

numberWaiting(): *Integer*

Implementation in VDM-SL

```
givePermission (craftIn: Aircraft)  
ext wr permission: Aircraft -set  
pre craftIn  $\notin$  permission  
post permission = permission  $\cup$  {craftIn}
```

<i>Airport</i>
<i>permission</i> : <i>Aircraft</i> [*] <i>landed</i> : <i>Aircraft</i> [*]
<i>givePermission</i> (<i>Aircraft</i>) <i>recordLanding</i> (<i>Aircraft</i>) <i>recordTakeOff</i> (<i>Aircraft</i>) <i>getPermission</i> (): <i>Aircraft</i> [*] <i>getLanded</i> (): <i>Aircraft</i> [*] <i>numberWaiting</i> (): <i>Integer</i>

Implementation in VDM-SL

```
recordLanding (craftIn: Aircraft)  
ext rd permission: Aircraft -set  
wr landed: Aircraft -set  
pre craftIn  $\in$  permission  $\wedge$  craftIn  $\notin$  landed  
post landed =  $\overline{\text{landed}}$   $\cup$  {craftIn}
```

<i>Airport</i>
<i>permission</i> : <i>Aircraft</i> [*] <i>landed</i> : <i>Aircraft</i> [*]
<i>givePermission</i> (<i>Aircraft</i>) <i>recordLanding</i> (<i>Aircraft</i>) <i>recordTakeOff</i> (<i>Aircraft</i>) <i>getPermission</i> (): <i>Aircraft</i> [*] <i>getLanded</i> (): <i>Aircraft</i> [*] <i>numberWaiting</i> (): <i>Integer</i>

Implementation in VDM-SL

```
recordTakeOff (craftIn: Aircraft)  
ext wr permission: Aircraft -set  
wr landed: Aircraft -set  
pre craftIn ∈ landed  
post landed =  $\overline{\textit{landed}}$  \ {craftIn} ∧ permission =  $\overline{\textit{permission}}$  \ {craftIn}
```

<i>Airport</i>
<i>permission</i> : <i>Aircraft</i> [*] <i>landed</i> : <i>Aircraft</i> [*]
<i>givePermission</i> (<i>Aircraft</i>) <i>recordLanding</i> (<i>Aircraft</i>) <i>recordTakeOff</i> (<i>Aircraft</i>) <i>getPermission</i> (): <i>Aircraft</i> [*] <i>getLanded</i> (): <i>Aircraft</i> [*] <i>numberWaiting</i> (): <i>Integer</i>

Implementation in VDM-SL

```
getPermission ( ) out: Aircraft -set  
ext rd permission: Aircraft -set  
pre TRUE  
post out = permission
```

<i>Airport</i>
<i>permission: Aircraft [*]</i> <i>landed: Aircraft [*]</i>
<i>givePermission(Aircraft)</i> <i>recordLanding(Aircraft)</i> <i>recordTakeOff(Aircraft)</i> <i>getPermission(): Aircraft [*]</i> <i>getLanded(): Aircraft [*]</i> <i>numberWaiting(): Integer</i>

Implementation in VDM-SL

```
getLanded ( ) out: Aircraft -set  
ext rd landed: Aircraft -set  
pre TRUE  
post out = landed
```

<i>Airport</i>
<i>permission: Aircraft [*]</i> <i>landed: Aircraft [*]</i>
<i>givePermission(Aircraft)</i> <i>recordLanding(Aircraft)</i> <i>recordTakeOff(Aircraft)</i> <i>getPermission(): Aircraft [*]</i> <i>getLanded(): Aircraft [*]</i> <i>numberWaiting(): Integer</i>

Implementation in VDM-SL

```
numberWaiting( ) total: ℕ  
ext rd permission: Aircraft -set  
rd landed: Aircraft -set  
post total = card (permission \ landed)
```

<i>Airport</i>
<i>permission: Aircraft [*]</i> <i>landed: Aircraft [*]</i>
<i>givePermission(Aircraft)</i> <i>recordLanding(Aircraft)</i> <i>recordTakeOff(Aircraft)</i> <i>getPermission(): Aircraft [*]</i> <i>getLanded(): Aircraft [*]</i> <i>numberWaiting(): Integer</i>

Thank You

CS636 Formal Methods

Topics Covered

Sequences

Sequences

- **Sequence is a collection of objects, however**
 - *A sequence is an ordered collection of objects.*
 - *In a sequence, repetitions are significant.*
 - **Notation:**
 - *A sequence is specified by enclosing its members in square brackets.*
- $s = [a, d, f, a, d, d, c]$*
- $queue = [MICHAEL, VARINDER, ELIZABETH, WINSTON, JUDITH]$*

Sequences

- As sequence is an ordered collection, so:
 - $[a, d, f] \neq [a, f, d]$
- Empty Sequence is expressed as:
 - $[]$
- The elements of a sequence are numbered, starting from 1, from left to right. We can refer to a particular element of a sequence by placing the position of the element in brackets
- $s(3)=f$ $queue(4) = WINSTON$

Sequence Operators

$s = [a, d, f, a, d, d, c]$

$queue = [MICHAEL, VARINDER, ELIZABETH, WINSTON, JUDITH]$

- The **len** operator gives us the length of the sequence.
 - $len\ s = 7$
 - $len\ queue = 5$
- The **elems** operator returns a set that contains all the members of the sequence (removes the duplicates):
 - $elems\ s = \{a, d, f, c\}$
 - $elems\ queue = \{MICHAEL, VARINDER, ELIZABETH, WINSTON, JUDITH\}$

Sequence Operators

$s = [a, d, f, a, d, d, c]$

$queue = [MICHAEL, VARINDER, ELIZABETH, WINSTON, JUDITH]$

- The head (**hd**) operator gives us the first element in the sequence;
- The tail (**tl**) operator gives us a sequence containing all but the first element
 - $hd\ s = s(1) = a$
 - $tl\ s = [d, f, a, d, d, c]$
 - $hd\ queue = MICHAEL$
 - $tl\ queue = [VARINDER, ELIZABETH, WINSTON, JUDITH]$

Sequence Operators

- The concatenation operator (^) operates on two sequences, and returns a sequence that consists of the two sequences joined together

```
if    first = [ w, e, r, w ]  
    and second = [ t, w, q ]  
then  
    first^second = [ w, e, r, w, t, w, q ]
```

Sequence Operators

- ❑ The override operator, $\dagger$, takes a sequence and gives us a new sequence with a particular element of the old sequence overridden by a new element.
- ❑ The generalized form is “ $s \dagger m$ ”
 - ❑ s is a sequence and m is a map
 - ❑ $[a, c, d, e] \dagger \{2 \rightarrow x, 4 \rightarrow y\} = [a, x, d, y]$
- ❑ The override operator is undefined if any index is invalid.

Sequence Operators

$s = [a, d, f, a, d, d, c]$

queue = [MICHAEL, VARINDER, ELIZABETH, WINSTON, JUDITH]

□ The **inds** operator returns a set of all the indices of the sequence.

□ ***inds s = {1, 2, 3, 4, 5, 6, 7}***

□ ***inds queue = {1, 2, 3, 4, 5}***

Sequence Operators

$s = [a, d, f, a, d, d, c]$

$queue = [MICHAEL, VARINDER, ELIZABETH, WINSTON, JUDITH]$

□ A subsequence operator is defined to allow us to extract a part of a sequence between two indices.

□ $subseq(s, 2, 5) = [d, f, a, d]$

□ *The language allows us to write this in the following, more convenient, way*

□ $s(2, \dots, 5) = [d, f, a, d]$

Sequence Operators

$s = [a, d, f, a, d, d, c]$

$queue = [MICHAEL, VARINDER, ELIZABETH, WINSTON, JUDITH]$

- The subsequence operator is undefined if either index is out of range, or if the first index is greater than the second
 - $s(1, \dots, 0) = []$
 - $s(8, \dots, 7) = []$
 - $s(2, \dots, 2) = [d]$

Thank You

CS636 Formal Methods

Topics Covered

Defining a Sequence by Comprehension

Defining a Sequence by Comprehension

- We can define a sequence by comprehension
- sequence of odd numbers from 1 to 20
 - $[a \mid a \in \{1, \dots, 20\} \bullet \text{is-odd}(a)]$
 - *is-odd* is a function that returns TRUE if a is odd and FALSE if a is even
- Generic Form:
 - $[\text{expression}(a) \mid a \in \text{SomeSet} \bullet \text{test}(a)]$
- When constructing the sequence, these values are considered in order, smallest first.

Defining a Sequence by Comprehension

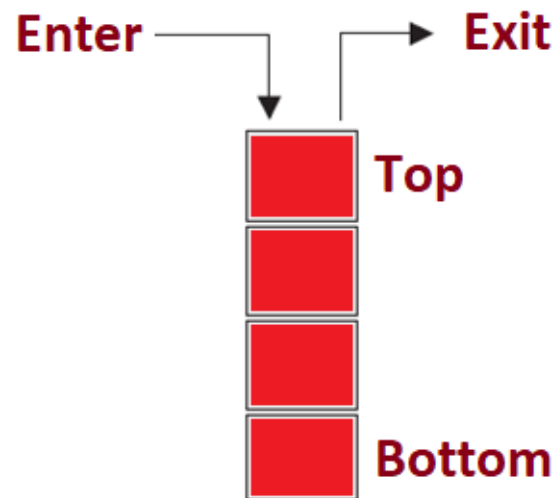
- Often sequence comprehension is used to ‘filter’ a sequence.
 - $s1 = [2, 3, 4, 7, 9, 11, 6, 7, 8, 14, 39, 45, 3]$
 - $s2 = [s1(i) \mid i \in \text{inds } s1 \bullet s1(i) > 10]$
 - $s2$ would evaluate to the sequence $[11, 14, 39, 45]$.

Sequence Type in VDM-SL

- ❑ Set is declared by appending the word –set at the end of the type contained in set
- ❑ To declare a variable to be of type sequence we place an asterisk after the name of the type contained within the sequence.
 - ❑ $\text{seq} : \mathbb{Z}^*$
 - ❑ $\text{convoy} : \text{SpaceCraft}^*$

Applications of Sequence: Stack

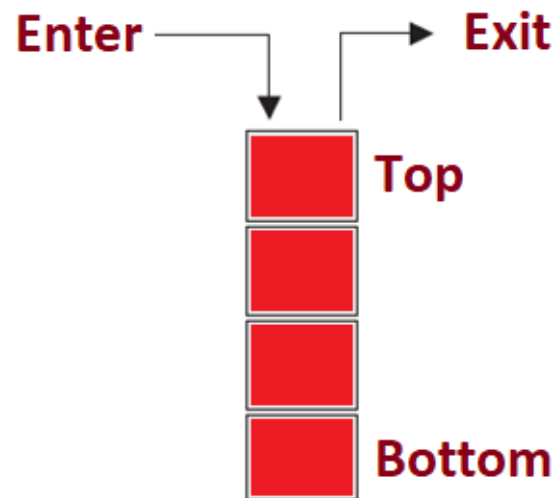
- A stack is a data structure, an ordered list that follows last-in-first-out (LIFO) protocol



<i>Stack</i>
<i>stack: Element[*]</i>
<i>push(Element)</i> <i>pop(): Element</i> <i>isEmpty(): Boolean</i>

Applications of Sequence: Stack

- A stack is a data structure, an ordered list that follows last-in-first-out (LIFO) protocol



<i>Stack</i>
<i>stack: Element[*]</i>
<i>push(Element)</i> <i>pop(): Element</i> <i>isEmpty(): Boolean</i>

Specifying the State of the Stack

types

Element = TOKEN

state *Stack* of

stack : *Element**

init mk-Stack(*s*) $\triangle$ *s* = []

end

Stack

stack: *Element*[*]

push(*Element*)

pop(): *Element*

isEmpty(): *Boolean*

Specifying the Operations on the Stack

```
push(itemIn : Element)  
ext wr stack : Element*  
pre TRUE  
post stack = [itemIn] ^ stack
```

<i>Stack</i>
<i>stack: Element[*]</i>
<i>push(Element)</i> <i>pop(): Element</i> <i>isEmpty(): Boolean</i>

Specifying the Operations on the Stack

pop() *itemRemoved* : *Element*
ext *wr* *stack* : *Element**
pre *stack* $\neq$ []
post *stack* = **tl** $\overline{\text{stack}}$ $\wedge$ *itemRemoved* = **hd** $\overline{\text{stack}}$

isEmpty() *query* : $\mathbb{B}$
ext *rd* *stack* : *Element**
pre TRUE
post *query* $\Leftrightarrow$ *stack* = []

<i>Stack</i>
<i>stack</i> : <i>Element</i> [*]
<i>push</i> (<i>Element</i>) <i>pop</i> (): <i>Element</i> <i>isEmpty</i> (): <i>Boolean</i>

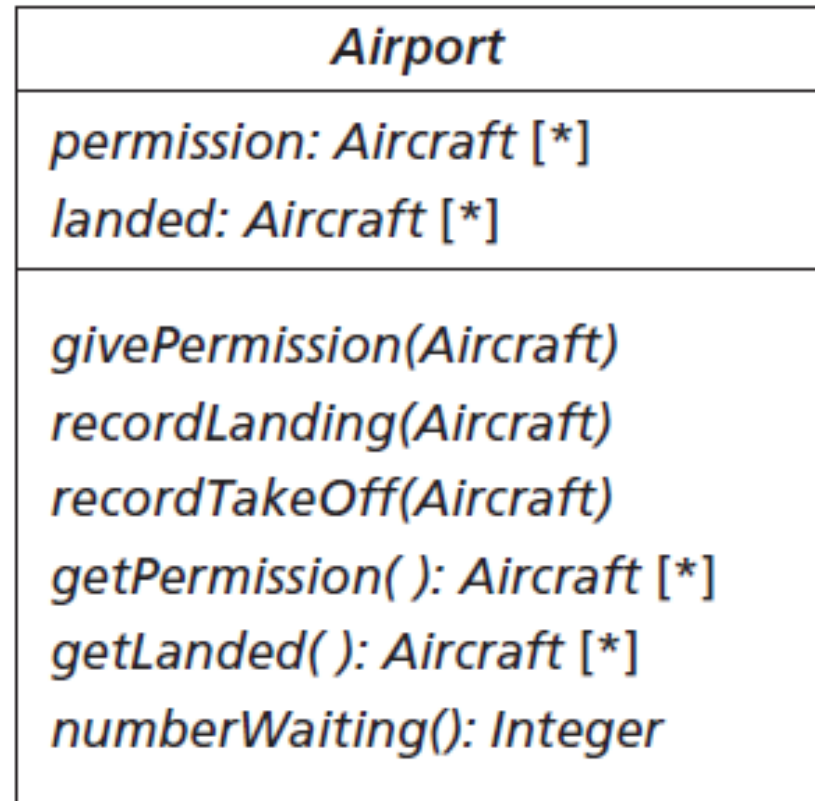
Thank You

CS636 Formal Methods

Topics Covered

Rethinking the Airport System

UML representation



Implementation in VDM-SL

```
types
Aircraft = TOKEN
state Airport2 of
  permission: Aircraft-set
  landed: Aircraft -set
  circling : Aircraft*
init mk-Airport2 (p, l, c)  $\triangleq$  p = { }  $\wedge$  l = { }  $\wedge$  c = []
end
```

Implementation in VDM-SL

```
isUnique(seqIn : Aircraft*) query :  $\mathbb{B}$   
pre TRUE  
post query  $\Leftrightarrow \forall i_1, i_2 \in \text{inds seqIn} \bullet i_1 \neq i_2 \Rightarrow \text{seqIn}(i_1) \neq \text{seqIn}(i_2)$ 
```

```
inv mk-Airport2 (p,l,c)  $\triangle$   $l \subseteq p$   
     $\wedge \text{elems } c \subseteq p$   
     $\wedge \text{elems } c \cap l = \{ \}$   
     $\wedge \text{isUnique}(c)$ 
```

Implementation in VDM-SL

```
allowToCircle (craftIn : Aircraft)  
ext wr circling : Aircraft*  
rd permission : Aircraft-set  
rd landed : Aircraft-set  
pre craftIn ∈ permission ∧ craftIn ∉ elems circling ∧ craftIn ∉ landed  
post circling = circling ^ [craftIn]
```


Implementation in VDM-SL

```
recordLanding( )  
ext wr circling : Aircraft*  
wr landed : Aircraft-set  
pre circling ≠ []  
post landed =  $\overline{landed} \cup \{ \mathbf{hd} \, \overline{circling} \} \wedge circling = \mathbf{tl} \, \overline{circling}$ 
```

Thank You

CS636 Formal Methods

Topics Covered

Sequences: Some useful functions

Sequences: Some useful functions

- ❑ Sometimes it is useful to have a function that returns the last element in a sequence
- ❑ Also sometimes you need a function that returns the sequence with the last element removed.
- ❑ *These are not standard VDM-SL functions, so here we will see description of few of these functions*
- ❑ *You can call them custom build functions and can include anywhere you need in your VDM-SL specification*

Sequences: Some useful functions

```
last(sequenceIn : Element*) elementOut : Element  
pre sequenceIn ≠ []  
post elementOut = sequenceIn(len sequenceIn)
```

```
allButLast(sequenceIn : Element*) sequenceOut : Element*  
pre sequenceIn ≠ []  
post sequenceOut sequenceIn(1, ... , (len sequenceIn - 1))
```

Sequences: Some useful functions

find(sequenceIn : Element, element : Element) position : $\mathbb{N}$*
pre *element $\in$ elems sequenceIn*
post *sequenceIn(position) = element*

findFirst(sequenceIn : Element, element : Element) position : $\mathbb{N}$*
pre *element $\in$ sequenceIn*
post *sequenceIn(position) = element $\wedge \forall i \in \text{inds sequenceIn} \bullet \text{sequenceIn}(i) = \text{element} \Rightarrow \text{position} \leq i$*

Thank You

CS636 Formal Methods

Topics Covered

Composite objects

Composite Objects

- ❑ There will be occasions when you need to associate more than one type with an object.
- ❑ *e.g. the “Car” object => registration number, make, price etc.*
- ❑ *The appropriate type for the object as a whole would then be a composite of all the types of its internal data.*
- ❑ *We call such a type a composite object type*

composite type definition

```
TypeName :: fieldname1 : Type1  
          fieldname2 : Type2  
          :
```

'::' symbol is called *“Composed of”*
Individual values are called fields

Example:

```
Time:: hour: N  
      minute: N  
      second: N
```

Composite Object Operators

- **make function: creates new object of given composite type**
 - ***mk-CompositeObjectName (parameter list)***

inv mk-Airport2 (p,l,c) $\triangleq$ $l \subseteq p \wedge \text{elems } c \subseteq p \wedge \text{elems } c \cap l = \{ \} \wedge \text{isUnique}(c)$

mk-Airport2: Aircraft-set x Aircraft-set x Aircraft* $\rightarrow$ Airport2

mk-Time: $\mathbb{N}$ x $\mathbb{N}$ x $\mathbb{N}$ $\rightarrow$ Time

Composite Object Operators

□ How to use mk-Time function

```
someTime = mk-Time (16, 20, 44)
```

□ ***How to specify the invariants***

```
Time:: hour: ℕ  
        minute: ℕ  
        second: ℕ  
inv mk-Time (h, m, s)  $\triangleq h < 24 \wedge m < 60 \wedge s < 60$ 
```

Composite Object Operators

- Individual fields of composite objects are selected (read) by using dot operator '.' followed by the name of the field.

```
someTime.minute = 20  
someTime.hour = 16
```

Composite Object Operators

- **mu (μ) function: returns one composite object from another but with one or more fields changed.**

newTime = μ (someTime, hour $\mapsto$ 15)

thisTime = μ (someTime, minute $\mapsto$ 0, second $\mapsto$ 0)

thisTime = mk-Time(someTime.hour, 0, 0)

Thank You

CS636 Formal Methods

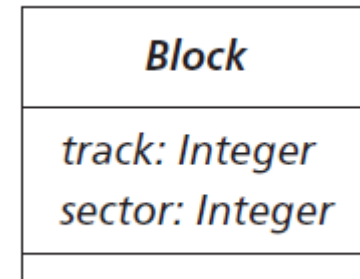
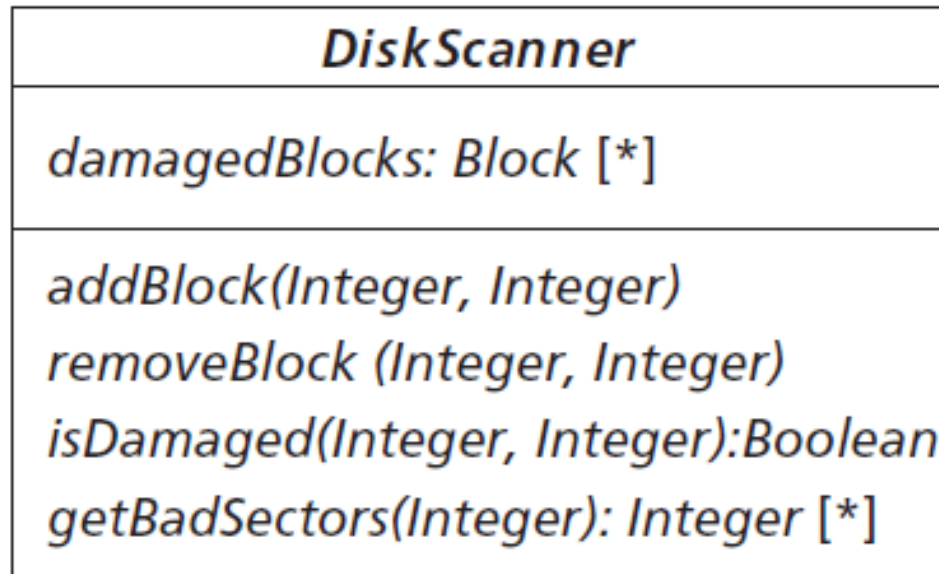
Topics Covered

DiskScanner System

DiskScanner System

- ❑ software designed to keep track of damaged blocks on the surface of a disk.
- ❑ *A disk is divided into a number of tracks*
- ❑ *each track into a number of sectors.*
- ❑ *block = track + sector number*
- ❑ *DiskScanner class*

UML representation



Implementation in VDM-SL

types

Block : : *track*: $\mathbb{N}$
sector: $\mathbb{N}$

state *DiskScanner* of
damagedBlocks: *Block-set*

init *mk-DiskScanner* (*dB*) $\triangleq$ *dB* = { }

Block

track: *Integer*
sector: *Integer*

Implementation in VDM-SL

```
addBlock (trackIn:  $\mathbb{N}$  , sectorIn:  $\mathbb{N}$ )  
ext wr damagedBlocks: Block-set  
pre mk-Block (trackIn, sectorIn)  $\notin$  damagedBlocks  
post damagedBlocks = damagedBlocks  $\cup$  {mk-Block (trackIn,  
sectorIn)}
```

```
removeBlock(trackIn:  $\mathbb{N}$  , sectorIn:  $\mathbb{N}$ )  
ext wr damagedBlocks: Block-set  
pre mk-Block (trackIn, sectorIn)  $\in$  damagedBlocks  
post damagedBlocks = damagedBlocks  $\setminus$  {mk-Block (trackIn,  
sectorIn)}
```

<i>DiskScanner</i>
<i>damagedBlocks</i> : Block [*]
<i>addBlock</i> (Integer, Integer) <i>removeBlock</i> (Integer, Integer) <i>isDamaged</i> (Integer, Integer): Boolean <i>getBadSectors</i> (Integer): Integer [*]

Implementation in VDM-SL

```
isDamaged (trackIn: $\mathbb{N}$ , sectorIn: $\mathbb{N}$ ) query:  $\mathbb{B}$   
ext rd damagedBlocks: Block-set  
pre TRUE  
post query  $\Leftrightarrow$  mk-Block (trackIn, sectorIn)  $\in$  damagedBlocks
```

```
getBadSectors (trackIn: $\mathbb{N}$ ) list:  $\mathbb{N}$ -set  
ext rd damagedBlocks: Block-set  
pre TRUE  
post list = {b.sector | b  $\in$  damagedBlocks  $\bullet$  b.track = trackIn}
```

<i>DiskScanner</i>
<i>damagedBlocks</i> : <i>Block</i> [*]
<i>addBlock</i> (<i>Integer</i> , <i>Integer</i>) <i>removeBlock</i> (<i>Integer</i> , <i>Integer</i>) <i>isDamaged</i> (<i>Integer</i> , <i>Integer</i>): <i>Boolean</i> <i>getBadSectors</i> (<i>Integer</i>): <i>Integer</i> [*]

Thank You

CS636 Formal Methods

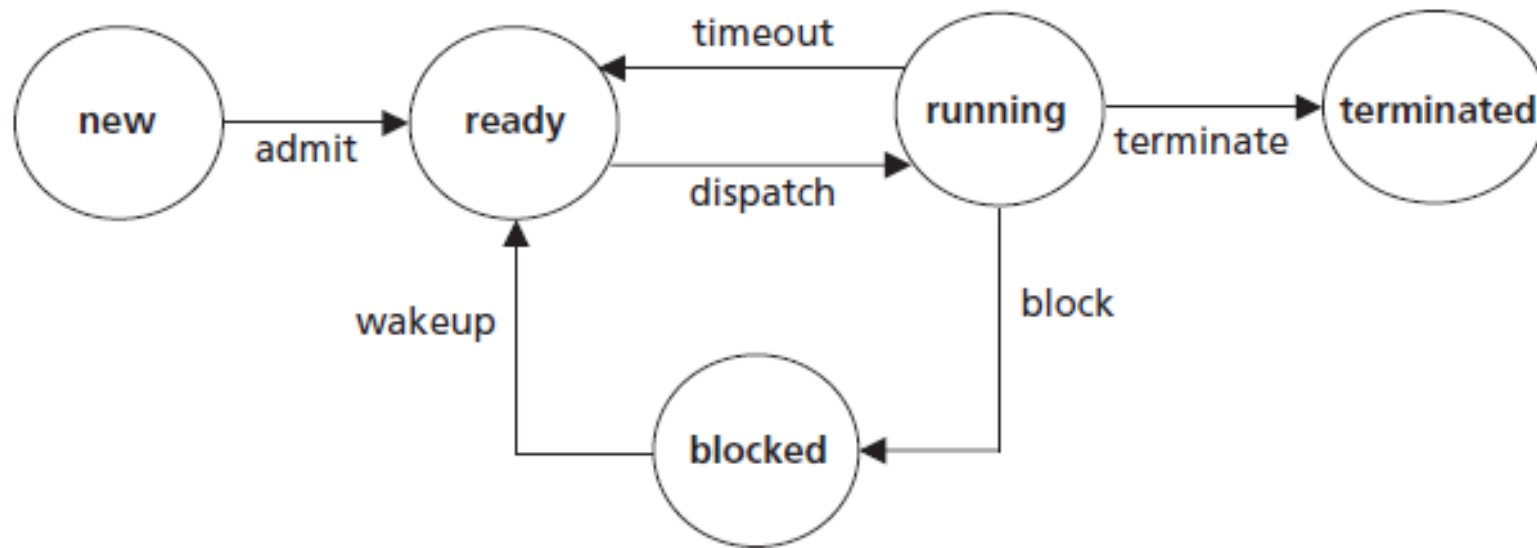
Topics Covered

A Process Management System

Process Management System

- **Process management system for a multitasking operating system.**
- ***Processes are identified by a unique process identification number (pid).***
- ***We will specify a simple first-in-first-out policy***

UML representation



<i>ProcessManagement</i>
<i>running</i> : String <i>waiting</i> : Process[*]
<i>admit</i> (String) <i>dispatch</i> () <i>timeOut</i> () <i>block</i> () <i>wakeUp</i> (String) <i>terminate</i> ()

<i>Process</i>
<i>id</i> : String <i>status</i> : Status

<<enumeration>> <i>Status</i>
<i>READY</i> <i>BLOCKED</i>

Implementation in VDM-SL

types

String = Char*

Status = <READY> | <BLOCKED>

Process :: *id* : *String*

status : *Status*

state *ProcessManagement* **of**

running : [*String*]

waiting : *Process**

inv *mk-ProcessManagement* (*run*, *wait*) $\triangle$ (*run* = nil $\vee$ $\neg \exists i \in \text{inds } \text{wait} \bullet \text{wait}(i).id = \text{run}$)

$\wedge \forall i, j \in \text{inds } \text{wait} \bullet i \neq j \implies \text{wait}(i).id \neq \text{wait}(j).id$

init *mk-ProcessManagement* (*run*, *wait*) $\triangle$ *run* = nil $\wedge$ *wait* = []

end

ProcessManagement

running: *String*

waiting: *Process*[*]

admit(*String*)

dispatch()

timeOut()

block()

wakeUp(*String*)

terminate()

Process

id: *String*

status: *Status*

<<enumeration>>
Status

READY

BLOCKED

Some Useful Functions

```
findPos(qIn : Process*, idIn : String) pos : ℕ
pre ∃p ∈ elems qIn • p.id = idIn
post qIn(pos).id = idIn
```

```
findNext(qIn : Process*) pos : ℕ
pre ∃p ∈ elems qIn • p.status = <READY>
post qIn(pos).status = <READY> ∧ ¬∃i ∈ {1,...,pos-1} • qIn(i).status = <READY>
```

```
remove(qIn : Process*, posIn: ℕ) qOut : Process*
pre posIn ∈ inds qIn
post qOut = qIn(1,..., posIn-1) ∧ qIn(posIn + 1,..., len qIn)
```

<i>ProcessManagement</i>
<i>running: String</i> <i>waiting: Process[*]</i>
<i>admit(String)</i> <i>dispatch()</i> <i>timeOut()</i> <i>block()</i> <i>wakeUp(String)</i> <i>terminate()</i>

<i>Process</i>
<i>id: String</i> <i>status: Status</i>

<i><<enumeration>></i> <i>Status</i>
<i>READY</i> <i>BLOCKED</i>

Thank You

CS636 Formal Methods

Topics Covered

**A Process Management System: Modeling
operations in VDM-SL**

Implementation in VDM-SL

```
admit(idIn: String)
ext wr waiting: Process*
rd running: [String]
pre (running = nil  $\vee$  idIn  $\neq$  running)  $\wedge$   $\forall p \in$  elems waiting  $\bullet$  p.id  $\neq$  idIn
post waiting = waiting ^ [mk-Process(idIn, <READY>)]
```

```
dispatch( )
ext wr running: [String]
    wr waiting: Process*
pre  running = nil  $\wedge$   $\exists p \in$  elems waiting  $\bullet$  p.status = <READY>
post running = waiting (findNext(waiting)).id
     $\wedge$  waiting = remove(waiting, findNext(waiting))
```

ProcessManagement
running: String waiting: Process[*]
admit(String) dispatch() timeOut() block() wakeUp(String) terminate()

Process
id: String status: Status

<<enumeration>> Status
READY BLOCKED

Implementation in VDM-SL

```
timeOut( )  
ext wr running: [String]  
    wr waiting: Process*  
pre running ≠ nil  
post waiting = waiting ∧ [mk-Process(running, <READY>)] ∧ running = nil
```

```
block( )  
ext wr running: [String]  
    wr waiting: Process*  
pre running ≠ nil  
post waiting = waiting ∧ [mk-Process(running, <BLOCKED>)] ∧ running = nil
```

<i>ProcessManagement</i>
<i>running</i> : String <i>waiting</i> : Process[*]
<i>admit</i> (String) <i>dispatch</i> () <i>timeOut</i> () <i>block</i> () <i>wakeUp</i> (String) <i>terminate</i> ()

<i>Process</i>
<i>id</i> : String <i>status</i> : Status

<<enumeration>> <i>Status</i>
READY BLOCKED

Implementation in VDM-SL

```
wakeUp(idIn: String)
ext wr waiting: Process*
pre waiting(findPos(waiting, idIn)).status = <BLOCKED>
post waiting = waiting † {findPos(waiting, idIn) ↦ mk-Process(idIn, <READY>)}
```

```
terminate(( ))
ext wr running: [String]
pre running ≠ nil
post running = nil
```

<i>ProcessManagement</i>
<i>running</i> : String <i>waiting</i> : Process[*]
<i>admit</i> (String) <i>dispatch</i> () <i>timeOut</i> () <i>block</i> () <i>wakeUp</i> (String) <i>terminate</i> ()

<i>Process</i>
<i>id</i> : String <i>status</i> : Status

<<enumeration>> <i>Status</i>
READY BLOCKED

THE LET... IN CLAUSE

```
let name = sub-expression
in expression(name)
```

□ Example

```
post running = waiting (findNext(waiting)).id
  ∧ waiting = remove(waiting, findNext(waiting))
```

```
post let next = findNext (waiting)
  in running = waiting(next).id
  ∧ waiting = remove(waiting, next)
```

<i>ProcessManagement</i>
<i>running: String</i> <i>waiting: Process[*]</i>
<i>admit(String)</i> <i>dispatch()</i> <i>timeOut()</i> <i>block()</i> <i>wakeUp(String)</i> <i>terminate()</i>

<i>Process</i>
<i>id: String</i> <i>status: Status</i>

<i><<enumeration>></i> <i>Status</i>
<i>READY</i> <i>BLOCKED</i>

THE LET... IN CLAUSE

□ Example

```
pre waiting(findPos(waiting, idIn)).status = <BLOCKED>
```

```
pre let pos = findPos(waiting, idIn)  
in waiting(pos).status = BLOCKED
```

```
post waiting =  $\overline{\text{waiting}}$  † {findPos( $\overline{\text{waiting}}$ , idIn)  
↦ mk-Process(idIn, <READY>)}
```

```
post let pos = findPos( $\overline{\text{waiting}}$ , idIn)  
in waiting =  $\overline{\text{waiting}}$  † {pos ↦ mk-Process(idIn, <READY>)}
```

```
post waiting =  $\overline{\text{waiting}}$  † {findPos( $\overline{\text{waiting}}$ , idIn)  
↦ mk-Process(idIn, <READY>)}
```

```
post let pos = findPos( $\overline{\text{waiting}}$ , idIn)  
in let wakeProcess = mk-Process(idIn, <READY>)  
In waiting =  $\overline{\text{waiting}}$  † {pos ↦ wakeProcess}
```

Thank You

CS636 Formal Methods

Topics Covered

Maps

Maps

- ❑ **Computing systems often involve relating two types of value together**
- ❑ ***A map is a special sort of set, which contains a set of maplets.***
- ❑ ***Each maplet connects an element of one set to an element of another set***
 - ❑ ***The first set is referred to as the domain***
 - ❑ ***The second is referred to as the range.***

Maps

Sensor	Condition
A	LOW
B	NORMAL
C	NORMAL
D	HIGH
E	NORMAL
F	NORMAL

$sensors = \{A \mapsto \langle LOW \rangle, B \mapsto \langle NORMAL \rangle, C \mapsto \langle NORMAL \rangle, D \mapsto \langle HIGH \rangle, E \mapsto \langle NORMAL \rangle, F \mapsto \langle NORMAL \rangle\}$

$m = \{a \mapsto y, b \mapsto x, C \mapsto x, d \mapsto z\}$

Maps

- By definition, all the domain elements in a map are unique.
- The ordering of the maplets is not significant – the map m above could be specified, without changing the meaning, as:

$$m = \{a \mapsto y, b \mapsto x, c \mapsto x, d \mapsto z\}$$

$$m = \{d \mapsto z, a \mapsto y, c \mapsto x, b \mapsto x\}$$

$$\textit{Empty map } \{\mapsto\}$$

Map Operators

□ Domain and Range Operator

The *domain* operator, **dom**, returns the set of all the domain elements of the maplets.

The *range* operator, **rng**, returns the set of all the range elements.

$$m1 = \{a \mapsto 1, b \mapsto 2, c \mapsto 2, d \mapsto 3, e \mapsto 4\}$$

$$m2 = \{a \mapsto 2, f \mapsto 1, c \mapsto 7\}$$

$$m3 = \{f \mapsto 2, g \mapsto 6\}$$

- **dom** $m1$ = $\{a, b, c, d, e\}$
- **rng** $m1$ = $\{1, 2, 3, 4\}$
- **dom** $m2$ = $\{a, f, c\}$
- **rng** $m2$ = $\{1, 2, 7\}$

Map Operators

□ Union Operator

$$m1 = \{a \mapsto 1, b \mapsto 2, c \mapsto 2, d \mapsto 3, e \mapsto 4\}$$

$$m2 = \{a \mapsto 2, f \mapsto 1, c \mapsto 7\}$$

$$m3 = \{f \mapsto 2, g \mapsto 6\}$$

$$m1 \cup m3 = \{a \mapsto 1, b \mapsto 2, c \mapsto 2, d \mapsto 3, e \mapsto 4, f \mapsto 2, g \mapsto 6\}$$

- **The union operator is defined only if no two domain elements are the same; if this is not the case, then union is undefined**

“ $m1 \cup m2$ ” and “ $m2 \cup m3$ ” are undefined

Map Operators

Override ($\dagger$) Operator:

- In the case where two or more domain elements are the same in both maps, we can use the override operator ($\dagger$).
- If the domain element of a maplet is the same in both sets, the second maplet wins.

$$m1 = \{a \mapsto 1, b \mapsto 2, c \mapsto 2, d \mapsto 3, e \mapsto 4\}$$

$$m2 = \{a \mapsto 2, f \mapsto 1, c \mapsto 7\}$$

$$m3 = \{f \mapsto 2, g \mapsto 6\}$$

$$m1 \dagger m2 = \{a \mapsto 2, b \mapsto 2, c \mapsto 7, d \mapsto 3, e \mapsto 4, f \mapsto 1\}$$

$$m3 \dagger m2 = \{f \mapsto 1, g \mapsto 6, a \mapsto 2, c \mapsto 7\}$$

Map Operators

Domain Restriction ($\triangleleft$) Operator:

- Defined with two operands. The first is a set and the second is a map.
- The result yields a map that contains only those maplets whose domain element is in the set. For example

$$m1 = \{a \mapsto 1, b \mapsto 2, c \mapsto 2, d \mapsto 3, e \mapsto 4\}$$

$$m2 = \{a \mapsto 2, f \mapsto 1, c \mapsto 7\}$$

$$m3 = \{f \mapsto 2, g \mapsto 6\}$$

$$\{a, c, e\} \triangleleft m1 = \{a \mapsto 1, c \mapsto 2, e \mapsto 4\}$$

$$\{e, f\} \triangleleft m2 = \{f \mapsto 1\}$$

$$\{\} \triangleleft m3 = \{\mapsto\}$$

Map Operators

Domain Deletion ($\Leftarrow$) Operator:

- ❑ Behaves similar to Domain Restriction operator but it deletes the maplet in questions.

$$m1 = \{a \mapsto 1, b \mapsto 2, c \mapsto 2, d \mapsto 3, e \mapsto 4\}$$

$$m2 = \{a \mapsto 2, f \mapsto 1, c \mapsto 7\}$$

$$m3 = \{f \mapsto 2, g \mapsto 6\}$$

$$\{a, c, e\} \Leftarrow m1 = \{b \mapsto 2, d \mapsto 3\}$$

$$\{e, f\} \Leftarrow m2 = \{a \mapsto 2, c \mapsto 7\}$$

$$\{\} \Leftarrow m3 = \{f \mapsto 2, g \mapsto 6\}$$

Map Application

- If we apply our map to a particular domain element, then the result is the range element.

$$m1 = \{a \mapsto 1, b \mapsto 2, c \mapsto 2, d \mapsto 3, e \mapsto 4\}$$

$$m2 = \{a \mapsto 2, f \mapsto 1, c \mapsto 7\}$$

$$m3 = \{f \mapsto 2, g \mapsto 6\}$$

$$m1(d) = 3$$

$$m2(f) = 1$$

$$m3(f) = 2$$

$$m3(x) \text{ is undefined}$$

Using the Map Type in VDM-SL

- To declare a variable to be of type Map we use a special

arrow $\xrightarrow{m}$

- For example, to declare a variable m that maps characters to natural numbers we would write:

$c: Char \xrightarrow{m} \mathbb{N}$

Thank You

CS636 Formal Methods

Topics Covered

Specifying a High-Security Building

a High-Security Building

Only authorized employees are allowed entry to the building and each one consists of a user name (which is unique) and a password, both of which must be supplied when the individual wishes to enter the building. If the details are correct, a signal is sent to the hardware instructing it to open the door, and the member of staff is recorded as being inside the building. When the member of staff wishes to leave the building, the individual supplies his or her user name, and as long as the user is recorded as being currently inside the building, a signal is sent to the hardware to open the door, and the employee is recorded as having left.

a High-Security Building

<i>SecuritySys</i>
<i>authorized: Employee[*]</i> <i>inside: String[*]</i>
<i>addEmployee(String, String)</i> <i>removeEmployee(String)</i> <i>enter(String, String): Signal</i> <i>leave(String): Signal</i>

<i>Employee</i>
<i>name: String</i> <i>password: String</i>

<i><<enumeration>></i> <i>Signal</i>
<i>OPEN_DOOR</i> <i>ACTIVATE_ALARM</i>

VDM-SL Specification

types

String = Char*

Signal = <OPEN_DOOR> | <ACTIVATE_ALARM>

authorized: string $\xrightarrow{m}$ *string*

state *SecuritySys* of

authorized: string $\xrightarrow{m}$ *string*
inside : String-set

<<enumeration>>
Signal

OPEN_DOOR
ACTIVATE_ALARM

Employee

name: String
password: String

SecuritySys

authorized: Employee[]*
inside: String[]*

addEmployee(String, String)
removeEmployee(String)
enter(String, String): Signal
leave(String): Signal

VDM-SL Specification

inv $\text{mk-SecuritySys}(a,i) \ i \subseteq \text{dom } a$
init $\text{mk-SecuritySys}(a,i) \triangle a = \{\mapsto\} \wedge i = \{\}$

$\text{addEmployee}(\text{nameIn} : \text{String}, \text{passwordIn} : \text{String})$
 $\text{ext wr } \text{authorized} : \text{string} \xrightarrow{m} \text{string}$
 $\text{pre } \text{nameIn} \notin \text{dom } \text{authorized}$
 $\text{post } \text{authorized} = \overline{\text{authorized}} \cup \{\text{nameIn} \mapsto \text{passwordIn}\}$

$\text{removeEmployee}(\text{nameIn} : \text{String})$
 $\text{ext wr } \text{authorized} : \text{string} \xrightarrow{m} \text{string}$
 $\text{rd } \text{inside} : \text{String-Set}$
 $\text{pre } \text{nameIn} \in \text{dom } \text{authorized} \wedge \text{nameIn} \notin \text{inside}$
 $\text{post } \text{authorized} = \{\text{nameIn}\} \triangleleft \overline{\text{authorized}}$

VDM-SL Specification

```
enter(nameIn : String, passwordIn : String) signal : Signal
  ext rd authorized : String  $\xrightarrow{m}$  String
    wr inside : String-set
  pre TRUE
  post (authorized(nameIn) = passwordIn  $\wedge$  nameIn  $\notin$   $\overline{\text{inside}}$ )
       $\wedge$  (inside =  $\overline{\text{inside}}$   $\cup$  {nameIn}  $\wedge$  signal = <OPEN_DOOR>)
       $\vee$  (authorized(nameIn)  $\neq$  passwordIn  $\vee$  nameIn  $\in$  inside)
       $\wedge$  (inside =  $\overline{\text{inside}}$   $\wedge$  signal = <ACTIVATE_ALARM>)
```

```
leave(nameIn : String) signal : Signal
  ext wr inside : String-set
  pre TRUE
  post nameIn  $\in$   $\overline{\text{inside}}$   $\wedge$  inside =  $\overline{\text{inside}} \setminus \{\text{nameIn}\}$   $\wedge$  signal = <OPEN_DOOR>
       $\vee$  nameIn  $\notin$  inside  $\wedge$  inside =  $\overline{\text{inside}}$   $\wedge$  signal = <ACTIVATE_ALARM>
```

Thank You

CS636 Formal Methods

Topics Covered

Robot Monitoring System

Robot Monitoring System

- ❑ **A software system that monitors a number of robots working at a space station.**
 - ❑ *Each robot will have a unique name and a mode, which can be WORKING, IDLE or BROKEN.*
 - ❑ *There are two sectors, A and B, in which a robot can be set to work.*

Robot Monitoring System

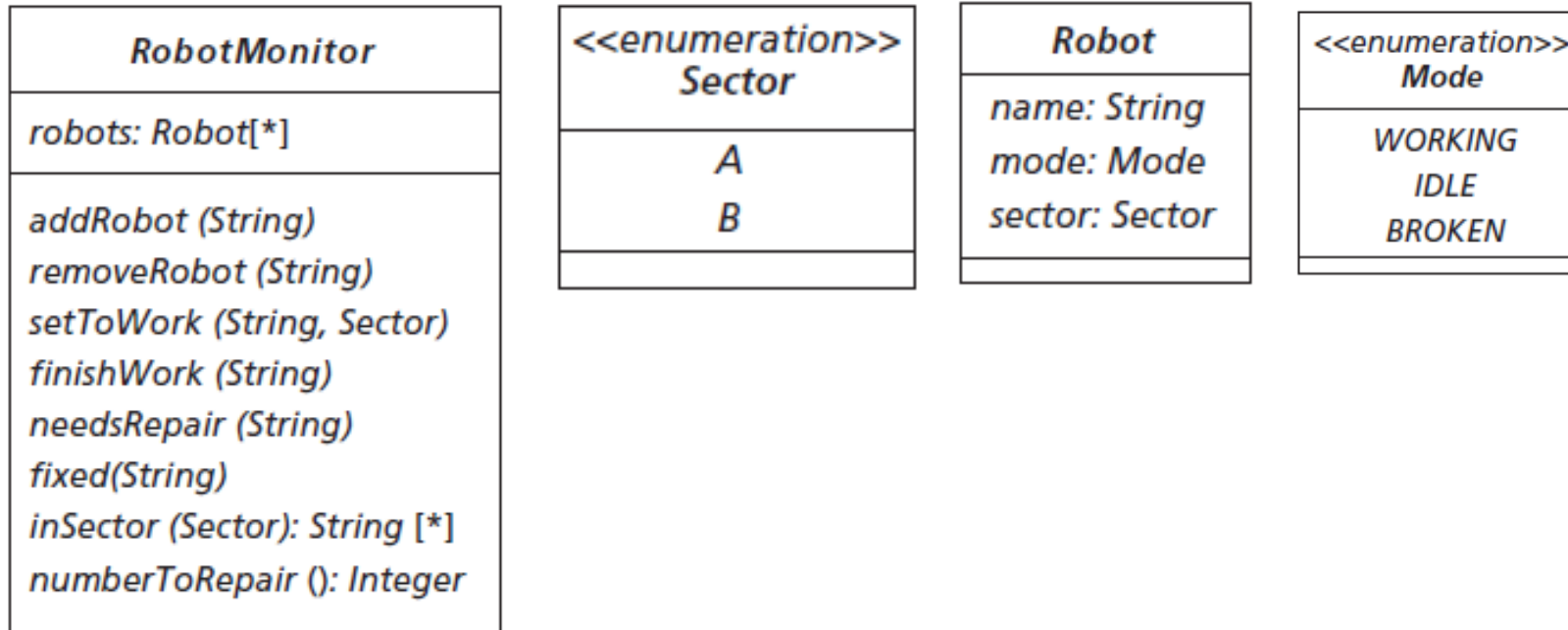
addRobot: accepts the name of a new robot and records the fact that this robot has been added to the collection. Its mode is set to idle and it is therefore not allocated a sector to work in.

removeRobot: accepts the name of a robot and records the removal of this robot from the system.

setToWork: accepts the name of a robot, that must currently be idle, and records the fact that it has been set to work in a given sector.

- **finishWork:** accepts the name of a robot, and records the fact that this robot has been removed from the sector and that its mode has been set to idle.
- **needsRepair:** as above but records its mode as broken.
- **fixed:** accepts the name of a broken robot and records that its mode has been set to idle.
- **inSector:** accepts a given sector and returns the names of those robots in that sector.
- **numberToRepair:** returns the number of broken robots.

a High-Security Building



VDM-SL Specification

types

String = *Char**

Mode = <WORKING> | <IDLE> | <BROKEN>

Sector = <A> |

Robot :: *name*: *String*

mode: *Mode*

sector: [*Sector*]

inv *mk-Robot*(-, *m*, *s*) $\triangleq m = \text{WORKING} \Leftrightarrow s \neq \text{nil}$

state *RobotMonitor* of

robots : *String* $\xrightarrow{m}$ *Robot*

inv *mk-RobotMonitor*(*r*) $\triangleq \forall n \in \text{dom } r \bullet n = r(n).\text{name}$

init *mk-RobotMonitor*(*r*) $\triangleq r = \{\mapsto\}$

end

VDM-SL Specification

```
addRobot (nameIn: String)  
ext wr robots: String  $\xrightarrow{m}$  Robot  
pre nameIn  $\notin \text{dom } \textit{robots}$   
post robots =  $\overline{\textit{robots}} \cup \{ \textit{nameIn} \mapsto \textit{mk-Robot}(\textit{nameIn}, \langle \textit{IDLE} \rangle, \textit{nil}) \}$   
  
removeRobot (nameIn: String)  
ext wr robots: String  $\xrightarrow{m}$  Robot  
pre nameIn  $\in \textbf{dom } \textit{robots} \wedge \textit{robots}(\textit{nameIn}).\textit{mode} \neq \langle \textit{WORKING} \rangle$   
post robots =  $\{ \textit{nameIn} \} \triangleleft \overline{\textit{robots}}$   
  
setToWork (nameIn: String, sectorIn: Sector)  
ext wr robots: String  $\xrightarrow{m}$  Robot  
pre nameIn  $\in \textbf{dom } \textit{robots} \wedge \textit{robots}(\textit{nameIn}).\textit{mode} = \langle \textit{IDLE} \rangle$   
post robots =  $\overline{\textit{robots}} \uparrow \{ \textit{nameIn} \mapsto \textbf{mk-Robot}(\textit{nameIn}, \langle \textit{WORKING} \rangle, \textit{sectorIn} ) \}$ 
```

VDM-SL Specification

```
finishWork (nameIn: String)  
   $\xrightarrow{m}$   
ext wr robots: String  $\xrightarrow{m}$  Robot  
pre   nameIn  $\in \text{dom robots} \wedge \text{robots}(\text{nameIn}).\text{mode} = \text{<WORKING>}$   
post  robots =  $\overline{\text{robots}} \uparrow \{ \text{nameIn} \mapsto \text{mk-Robot}(\text{nameIn}, \text{<IDLE>}, \text{nil}) \}$   
  
needsRepair (nameIn: String)  
   $\xrightarrow{m}$   
ext wr robots: String  $\xrightarrow{m}$  Robot  
pre   nameIn  $\in \text{dom robots}$   
post  robots =  $\overline{\text{robots}} \uparrow \{ \text{nameIn} \mapsto \text{mk-Robot}(\text{nameIn}, \text{<BROKEN>}, \text{nil}) \}$   
  
fixed (nameIn: String)  
   $\xrightarrow{m}$   
ext wr robots: String  $\xrightarrow{m}$  Robot  
pre   nameIn  $\in \text{dom robots} \wedge \text{robots}(\text{nameIn}).\text{mode} = \text{<BROKEN>}$   
post  robots =  $\overline{\text{robots}} \uparrow \{ \text{nameIn} \mapsto \mu(\overline{\text{robots}}(\text{nameIn}), \text{mode} \mapsto \text{<IDLE>}) \}$ 
```

VDM-SL Specification

```
inSector (sectorIn:Sector) result : String-set  
ext rd robots: String  $\xrightarrow{m}$  Robot  
pre    TRUE  
post   result = {r.name | r ∈ rng robots • r.sector = sectorIn}  
  
numberToRepair() number : ℕ  
ext rd robots: String  $\xrightarrow{m}$  Robot  
pre    TRUE  
post   number = card {r | r ∈ rng robots • r.mode = <BROKEN>}
```

Thank You

CS636 Formal Methods

Topics Covered
Well-Formedness

Well-Formedness

- ❑ The property of a model, specification, or expression adhering to the syntactic rules and constraints defined by a formal language or formalism.
- ❑ *When a model or specification is well-formed, it means that it is structurally correct and conforms to the syntax and grammar of the formal language being used.*

Well-Formedness

- ❑ Ensuring well-formedness is essential because formal methods rely on rigorous mathematical techniques for analysis and verification, and any errors or ambiguities in the specification could lead to incorrect results or misinterpretations.
- ❑ Once a model is well-formed, formal analysis techniques can be applied to check properties like consistency, correctness, safety, liveness, and more, thereby enhancing the reliability and correctness of complex systems.

Completeness

- ❑ **Property of a formal model or specification that indicates it includes all the necessary and relevant information to describe the system or problem being modeled.**
- ❑ **A formal model is considered complete when it fully captures all the intended behaviors, states, and constraints of the system in question.**

Completeness

- ❑ **Completeness is essential because it ensures that no critical aspects or essential requirements of the system are omitted from the formal representation.**
- ❑ **If a formal model is not complete, it may lead to incorrect or insufficient conclusions when performing formal analysis or verification.**

Completeness

- ❑ A complete formal model should cover all possible scenarios, edge cases, and interactions between components of the system.
- ❑ It should also encompass all relevant variables, transitions, and invariants that define the system's behavior accurately.

Completeness

- ❑ Achieving completeness in formal models can be challenging, especially for complex systems, as it requires a thorough understanding of the system's requirements and potential behaviors.
- ❑ Additionally, modeling all aspects of a system with complete accuracy can be a time-consuming process.

Completeness

- ❑ Practitioners often strive to strike a balance between the desired level of completeness and the practical limitations, aiming to model the most critical parts of the system accurately to ensure the effectiveness of formal analysis and verification efforts.
- ❑ It's important to note that achieving complete formal models is not always feasible or necessary, but making the model as complete as possible within reasonable bounds is crucial for reliable results in formal methods applications.

Thank You

CS636 Formal Methods

Topics Covered

Proofs

Proof

- ❑ A "proof" refers to a rigorous demonstration or argument that establishes the correctness or validity of a particular claim or statement within a formal system.
- ❑ It involves applying formal rules of logic and reasoning to show that a property, specification, or assertion holds true for a given model or system under consideration.

Proof

- **Proofs play a central role in formal methods for various purposes, including:**
 - **Verification:** Proofs are used to verify that a system meets its desired properties or requirements. For example, in software verification, a proof can demonstrate that a program satisfies a specified safety property, meaning it will never exhibit certain undesirable behaviors.
 - **Correctness:** Proofs are employed to demonstrate the correctness of algorithms, protocols, or systems. By providing a mathematical guarantee of correctness, formal methods can offer strong assurances of system reliability.
 - **Refinement:** Proofs are used in the process of refining abstract models to more concrete designs. These proofs ensure that the implementation correctly refines the higher-level specification.

Proof

- **Proofs play a central role in formal methods for various purposes, including:**
 - **Consistency:** Proofs can be used to check the consistency of a formal model. A consistent model does not contain conflicting statements or logical contradictions.
 - **Completeness:** In formal systems, proofs can establish the completeness of logical systems, showing that all true statements within the system can be proven.

Consistency of a formal methods

- Consistency refers to a property of a formal system or a formal model where it does not contain conflicting statements, logical contradictions, or incompatible assumptions.
- *If a system is inconsistent, it could produce conflicting or unreliable outcomes when subjected to formal analysis, verification, or reasoning.*

Consistency of a formal methods

- ❑ When inconsistency is detected in a formal system or model, it indicates a flaw in the logic or definitions. In such cases, it is crucial to identify and resolve the conflicting statements to restore the system's consistency.
- ❑ Often, inconsistencies are uncovered during formal verification or automated theorem proving, and addressing them is an essential step in ensuring the correctness and reliability of the formal methods application.

Consistency of a formal methods

- ❑ **Consistency ensures that the results obtained through formal methods are reliable, and that the system's behavior is well-defined and free from paradoxes or contradictions.**
- ❑ **It is one of the key aspects that distinguish formal methods from informal methods of reasoning, providing a solid foundation for mathematically rigorous analysis and verification.**

Thank You

CS636 Formal Methods

Topics Covered

Robustness

Robustness

- Ability to provide accurate and reliable results in the face of various challenges, uncertainties, and complexity associated with modeling and verifying systems.
- *A formal method is considered robust if it can handle different scenarios, edge cases, and potential sources of errors while still producing meaningful and trustworthy outcomes.*

Robustness Evaluation Aspects

- ❑ ***Completeness:*** A robust formal method should be able to handle complex systems and specifications while remaining as complete as possible, capturing all essential behaviors and requirements.
- ❑ ***Scalability:*** The method should be scalable, meaning it can handle large and complex systems efficiently, without becoming computationally infeasible or impractical.
- ❑ ***Expressiveness:*** A robust formal method should support a rich set of constructs and language features, allowing modelers to represent a wide range of system behaviors accurately.

Robustness Evaluation Aspects

- ❑ ***Automation:*** *The level of automation provided by the method is crucial for its robustness. Automated tools for model checking, theorem proving, and other analyses reduce the risk of human error and make the verification process more reliable.*
- ❑ ***Soundness and Completeness of Analysis:*** *The method should have sound theoretical foundations, meaning that any conclusion drawn from the analysis is guaranteed to be correct. It should also be as complete as possible, ensuring that it explores all relevant aspects of the model.*
- ❑ ***Handling Abstraction and Refinement:*** *A robust formal method should allow users to work with models at various levels of abstraction and support refinement techniques to iteratively enhance the model's precision.*

Robustness Evaluation Aspects

- ❑ *Handling Uncertainty: Real-world systems often have uncertainties and partial knowledge. A robust formal method should handle these uncertainties gracefully and provide insights into the behavior of the system under different conditions.*
- ❑ *Tool Support: The availability and maturity of tools supporting the formal method are crucial for its robustness. Well-maintained and actively developed tools can improve the usability and effectiveness of the method.*
- ❑ *Applicability to Different Domains: A truly robust formal method should be applicable to a wide range of domains, from hardware and software systems to safety-critical and real-time systems.*

Thank You