# SE 2A04 Fall 2002

# 03 Software Specification and Description

Instructor: W. M. Farmer

Revised: 08 November 2002

# Descriptions of Engineering Products

- A **description** of a product is a **model** of the product
  - Should include only certain key aspects of the product
  - Should be easier to understand than the product itself

- Mathematics is used to make descriptions precise

- A variety of descriptions, instead of a single description, is used to efficiently describe the different aspects of a product
  - There is never a complete description of a product

# Specifications

- A **specification** describes the attributes **required** of a product

- A product **satisfies** a specification if it possesses the attributes described by the specification

- A specification serves three purposes:
  - Blueprint for developing the product
  - Basis for verifying the correctness of the product
  - High-level description of the product

# Actual Descriptions

- An **(actual) description** describes the **actual** attributes of a product

- A **constructive description** describes how the product is constructed from other products
  - A program's code is a constructive description

- A **behavioral description** describes how the product works
  - Blackbox: describes the external (visible) behavior
  - Whitebox: describes the internal (invisible) behavior

# Specification vs. Description

- Both specifications and descriptions describe attributes, but they are different in intent
  - The same descriptive item may be interpreted as either a specification or a description
- Specifications are often interpreted as abstract descriptions
- Descriptions are often interpreted as concrete specifications

# Refinement

- Let $S$ and $S'$ be specifications

- $S'$ is a **refinement** of $S$ if every product that satisfies $S'$ also satisfies $S$

- The **refinement method** is a powerful design method in which a specification $S_0$ is to incrementally refined to a specification $S_n$ of a product that is readily implementable

# Procedure Specification Methods

1. Input/output specification

2. Before/after specification

   - Input/output specification is a special case

3. Trace specification

4. Pre- and postcondition specification

   Note: Specifications methods 1–3 view procedures as certain kinds of functions

# Review of Functions

- $f : A \rightarrow B$ means $f$ is a function that maps members of $A$ to members of $B$

- $f$ can be viewed as a **set of ordered pairs**:

  $$\{(x, y) : A \times B \mid y = f(x)\} \subseteq A \times B$$

- $f$ may not be defined for all members of $A$
  - The **domain** of $f$ is the set $\text{dom}(f) = \{x : A \mid f(x)\downarrow\}$
  - $f$ is **total** if $\text{dom}(f) = A$
  - $f$ is **partial** if $\text{dom}(f) \subseteq A$
  - $f$ is **strictly partial** if $\text{dom}(f) \subset A$

- The function can be specified in various ways:
  - **Definitional specification**: $f = E$
  - **Relational specification**: $(R, D)$
  - **Axiomatic specification**: $A(f)$

# Partiality in Software Specifications

Specifications can be partial in two ways:

1. A specification may **not fully specify** an object or operation
   - What is not specified is considered to be implicitly specified as “don’t care” and can thus be freely implemented

2. A specification may state that the application of an operation in certain states or on certain inputs is **undefined** or **illegal**
   - An undefined application is implemented by an **exception**

# Input/output Specifications

- Let $I$ be a set of possible inputs, and $O$ be a set of possible outputs
- A procedure without side-effects can be viewed as a **function** $f : I \rightarrow O$ **that maps inputs to outputs**

# Definitional Specification

- A **definition** specifies a unique object

- So a definition of a function specifies a unique function:
  - Syntax: $f = E$ where $E$ is an expression
  - Semantics: $f$ is the unique function denoted by $E$

- Example 1: Integer square function $f : \mathbf{Z} \rightarrow \mathbf{Z}$

  $f = \lambda\, x : \mathbf{Z} \,.\, x * x$ (or $f(x) = x * x$)

- Example 2: Integer square root function $g : \mathbf{Z} \rightarrow \mathbf{Z}$

  $g = \lambda\, x : \mathbf{Z} \,.\, \mathrm{I}\, y : \mathbf{Z} \,.\, 0 \leq y \wedge y * y = x$

  Notice that $g$ is strictly partial

# Relational Specification

- A **relational specification** is a pair $(R, D)$ where:
  1. $R \subseteq I \times O$
  2. $D \subseteq \mathsf{dom}(R) = \{x : I \mid \exists\, y : O \;.\; R(x, y)\} \subseteq I$

- $f : I \to O$ **satisfies** $(R, D)$ if:
  1. $\forall\, x : I \;.\; x \in \mathsf{dom}(f) \Rightarrow R(x, f(x))$
  2. $D \subseteq \mathsf{dom}(f)$

- Example 1: Integer square function $f : \mathbf{Z} \to \mathbf{Z}$

  $R = \{(x, y) \in \mathbf{Z} \times \mathbf{Z} \mid y = x * x\}$

  $D = \mathbf{Z}$

- Example 2: Integer square root function $g : \mathbf{Z} \to \mathbf{Z}$

  $R = \{(x, y) \in \mathbf{Z} \times \mathbf{Z} \mid y * y = x\}$

  $D = \{x \in \mathbf{Z} \mid \exists y : \mathbf{Z} \;.\; y * y = x\} \subseteq \{x : \mathbf{Z} \mid 0 \leq x\}$

# Axiomatic Specification

- An **axiomatic specification** is a formula $A(f)$:
  - $A(f)$ is an **axiom** for the behavior of $f$

- $g : I \to O$ **satisfies** $A(f)$ if $A(g)$ is true

- Example 1: Integer square function $f : \mathbf{Z} \to \mathbf{Z}$

  $$A(f) \Leftrightarrow \forall\, x : \mathbf{Z} \;.\; f(x) = x * x$$

- Example 2: Integer square root function $g : \mathbf{Z} \to \mathbf{Z}$

  $$\begin{aligned} A(f) \Leftrightarrow \forall\, x : \mathbf{Z} \;.\; \text{if}(&\exists y : \mathbf{Z} \;.\; y * y = x, \\ &f(x) * f(x) = x, \\ &f(x) \uparrow) \end{aligned}$$

# What is a State?

- A state of a machine is an abstract entity that can only be defined indirectly

- A **description of a state** of a machine is a description of all the information needed to predict the machine's future response to input from the external environment

- Physical machines have an infinite number of states, but they can usually be viewed as if they had a finite number of states
  - Aspects of a state which are irrelevant to the behavior of the machine (e.g., temperature and location) can be ignored
  - **Transition states** between **stable states** can also be ignored

- Digital computers are design to behave as if they were finite state machines

# State Machines

- A **state machine** $M$ consists of the following components:
  1. A fixed set $S$ of **states** including an **initial state**
  2. A fixed set $I$ of **inputs**
  3. A fixed set $O$ of **outputs**
  4. An **output** relation out $\subseteq I \times S \times O$
  5. A **next state** relation ns $\subseteq I \times S \times S$

- $M$ is a **finite state machine** if $S$ is finite

- $M$ is **deterministic** if the relations are functions, i.e., out : $I \times S \to O$ and ns : $I \times S \to S$

# Computing Machines

- A computing machine can viewed as a finite state machine:
  - The machine can only be in one of finitely many **stable states**
  - An **execution** takes the machine through a **sequence of states**

- A program, module, or procedure can be viewed as a small computing machine, i.e., a finite state machine
  - A state of the machine is the set of variables (data structures) that the program, module, or procedure can modify

# Before/After Specifications

- Let $I$ be a set of possible inputs, $O$ be a set of possible outputs, and $S$ be a set of possible states

- A procedure (possibly with side-effects) can be viewed as a **function** $f : I \times S \to O \times S$ **that maps inputs and before-states to outputs and after-states**

- The function $f$ can be represented as a pair $(f_1, f_2)$ of functions where:

  $$f_1 : I \times S \to O$$
  $$f_2 : I \times S \to S$$

- An input/output function is a special case of a before/after function where the after-state is always the same as the before-state

# Before/After Specification Format

Components of a before/after procedure specification:

1. The **name** and **type** of the procedure

2. The **exceptions** that the procedure can raise
   - Represented as predicates

3. **State constants** with value conditions

4. **State variables** with initial values

5. **Behavior rules** (preferably given in a tabular format):
   - Output rules
   - State transition rules
   - Exception rules

# Example 1:
# Counted Integer Square Function

1. counted-int-square : $\mathbf{Z} \rightarrow \mathbf{Z}$

2. Exceptions: none required

3. State constants: none

4. State variables: $c : \mathbf{Z}$ [initially $c = 0$]

5. Behavior rules:

| **Input** $x : \mathbf{Z}$ | **Output** $y : \mathbf{Z}$ | **State Transition** | **Exception** |
|---|---|---|---|
| $x \in \mathbf{Z}$ | $y = x * x$ | $c' = c + 1$ | |

# Example 2: Counted Integer Square Root Function

1. counted-int-sqrt : $\mathbf{Z} \rightarrow \mathbf{Z}$

2. Exceptions: sqrt-complex, sqrt-irrational

3. State constants: none

4. State variables: $c : \mathbf{Z}$ [initially $c = 0$]

5. Behavior rules:

| **Input** $x : \mathbf{Z}$ | **Output** $y : \mathbf{Z}$ | **State Transition** | **Exception** |
|---|---|---|---|
| $x < 0$ | | $c' = c + 1$ | sqrt-complex |
| $0 \leq x \wedge$ $\neg \exists y : \mathbf{Z} \,.\, y * y = x$ | | $c' = c + 1$ | sqrt-irrational |
| $0 \leq x \wedge$ $\exists y : \mathbf{Z} \,.\, y * y = x$ | $0 \leq y \wedge$ $y * y = x$ | $c' = c + 1$ | |

# Trace Specifications

- Let $I$ be a set of possible inputs, $O$ be a set of possible outputs, $S$ be a set of possible states, and $S^*$ be the set of finite sequences of members of $S$

- A **trace** is an execution history expressed as a sequence of states
  - A finite trace is a member of $S^*$

- A procedure (possibly with side-effects) can be viewed as a **function** $f : I \times S^* \rightarrow O \times S^*$ **that maps inputs and before-traces to outputs and after-traces**

- The function $f$ can be represented as a pair $(f_1, f_2)$ of functions where:

$$f_1 : I \times S^* \rightarrow O$$
$$f_2 : I \times S^* \rightarrow S^*$$

# Pre- and Postconditions Specification

- A state is specified by a tuple $X = (x_1, \ldots, x_n)$ of variables
- A procedure is specified by:
  1. A **precondition** $\varphi(x_1, \ldots, x_n)$ on the initial values of the state variables
  2. A **postcondition** $\psi(x_1, \ldots, x_n; x'_1, \ldots, x'_n)$ on the initial and final values of the state variables
- A procedure **satisfies** the specification if, for all states $X = (x_1, \ldots, x_n)$, whenever

$$\varphi(x_1, \ldots, x_n)$$

holds, the procedure is started in state $X$, and the procedure terminates in state $X' = (x'_1, \ldots, x'_n)$, then

$$\psi(x_1, \ldots, x_n; x'_1, \ldots, x'_n)$$

holds.

# Partial vs. Total Correctness

- A procedure $P$ is **partially correct** with respect to a pre- and postcondition specification $S = (\varphi, \psi)$ if $P$ satisfies $S$

- A procedure $P$ is **totally correct** with respect to a pre- and postcondition specification $S = (\varphi, \psi)$ if both:
  - $P$ satisfies $S$
  - $P$ terminates whenever it is started in a state for which the precondition $\varphi$ holds

# Module Design Documents

- **Module Guide**
- For each module:
  - **Module Interface Specification (MIS)**
  - **Module Internal Design (MID)**

# Module Guide

- The Module Guide lists all the modules of the software product

- The following information is given for each module:

  1. Module name
  2. Module nickname (2 or 3 letters)
  3. Service: Short informal description of what services the module provides
  4. Secret: Short informal description of what secret the module hides
  5. Expected changes: A short description of expected implementation changes

# Components of an Axiomatic Input/Output MIS

1. **Imported modules**

2. **Interface**

   - Types
   - Constant names and types
   - Procedure names and types

3. **Exceptions**

4. **Axioms**

# Example: Axiomatic Input/Output MIS For Stacks ADT (1)

- Imported modules: none required

- Interface:

```
INTERFACE Stacks;
  TYPE Stack;
  CONST Bottom: Stack;
  PROCEDURE Push(i: INTEGER; s: Stack): Stack;
  PROCEDURE Top(s: Stack): INTEGER;
  PROCEDURE Pop(s: Stack): Stack;
END Stacks.
```

- Exceptions: EmptyStack

# Example: MIS for Stacks ADT (2)

- Axioms:

  1. **Bottom is not a Push stack.**

     $\forall\, i : \texttt{INTEGER}, s : \texttt{Stack} \;.\; \texttt{Bottom} \neq \texttt{Push}(i, s)$

  2. **Push is one-to-one.**

     $\forall\, i_1, i_2 : \texttt{INTEGER}, s_1, s_2 : \texttt{Stack} \;.$

     $\quad \texttt{Push}(i_1, s_1) = \texttt{Push}(i_2, s_2) \Rightarrow (i_1 = i_2 \wedge s_1 = s_2)$

  3. **Induction axiom for stacks.**

     $\forall\, P : \texttt{Stack} \longrightarrow \texttt{BOOLEAN} \;.$

     $\quad [P(\texttt{Bottom}) \wedge$

     $\quad\quad \forall\, s : \texttt{Stack} \;.\; P(s) \Rightarrow \forall\, i : \texttt{INTEGER} \;.\; P(\texttt{Push}(i, s))]$

     $\quad \Rightarrow \forall\, s : \texttt{Stack} \;.\; P(s)$

# Example: MIS for Stacks ADT (3)

4. **Top applied to a Push stack.**
   $\forall\, i : \texttt{INTEGER}, s : \texttt{Stack} \,.\, \texttt{Top}(\texttt{Push}(i, s)) = i$

5. **Pop applied to a Push stack.**
   $\forall\, i : \texttt{INTEGER}, s : \texttt{Stack} \,.\, \texttt{Pop}(\texttt{Push}(i, s)) = s$

6. **Bottom has no top.**
   $\texttt{Top}(\texttt{Bottom})\!\uparrow$
   [EmptyStack exception]

7. **Bottom has no pop.**
   $\texttt{Pop}(\texttt{Bottom})\!\uparrow$
   [EmptyStack exception]

Note: This MIS has the form of an **axiomatic theory** $(L, \Gamma)$ where

– $L$ is the **language** defined by the interface of the MID
– $\Gamma$ is the set of axioms of the MID

# Example: Stacks ADT Module (1)

```
(*

Title: Stacks ADT

Interface:

INTERFACE Stacks;
  TYPE Stack;
  CONST Bottom: Stack;
  PROCEDURE Push(i: INTEGER; s: Stack): Stack;
  PROCEDURE Top(s: Stack): INTEGER;
  PROCEDURE Pop(s: Stack): Stack;
END Stacks.

*)

MODULE Stacks;

IMPORT Out;
```

# Example: Stacks ADT Module (2)

```
(* Types *)

TYPE

  Stack* = POINTER TO StackRec;

  StackRec =
    RECORD
      item: INTEGER;
      rest: Stack;
    END;

(* Constants *)

VAR Bottom-: Stack;   (* represents the empty stack *)
```

# Example: Stacks ADT Module (3)

```
(* Exceptions: *)

PROCEDURE EmptyStackException();
BEGIN
  Out.String("Stacks.EmptyStackException: The stack is empty.");
  HALT(1)  (* Abort program *)
END EmptyStackException;

(* Local procedures *)

PROCEDURE Empty(s: Stack): BOOLEAN;
BEGIN
  RETURN s = Bottom
END Empty;
```

# Example: Stacks ADT Module (4)

```
(* Interface procedures *)

PROCEDURE Push*(i: INTEGER; s: Stack): Stack;
  VAR t: Stack;
BEGIN
  NEW(t);
  t^.item := i;
  t^.rest := s;
  RETURN t
END Push;

PROCEDURE Top*(s: Stack): INTEGER;
BEGIN
  IF ~Empty(s) THEN
    RETURN s^.item
  ELSE
    EmptyStackException()
  END
END Top;
```

# Example: Stacks ADT Module (5)

```
PROCEDURE Pop*(s: Stack): Stack;
BEGIN
  IF ~Empty(s) THEN
    RETURN s^.rest
  ELSE
    EmptyStackException()
  END
END Pop;

BEGIN
  Bottom := NIL   (* initializes Bottom *)
END Stacks.
```

# Components of a Before/After MIS

1. **Imported modules**

2. **Interface**
   - Types
   - Constant names and types
   - Procedure names and types

3. **Exceptions**

4. **State constants** with value conditions

5. **State variables** with initial values

6. **Behavior rules**
   - Output rules
   - State transition rules
   - Exception rules

# Example: Before/After MIS For Stack Data Structure (1)

- Imported modules: none required

- Interface:

```
INTERFACE Stack;
  PROCEDURE Reset();
  PROCEDURE MaxHeight(): INTEGER;
  PROCEDURE Height(): INTEGER;
  PROCEDURE Empty(): BOOLEAN;
  PROCEDURE Full(): BOOLEAN;
  PROCEDURE Push(i: INTEGER);
  PROCEDURE Pop();
  PROCEDURE Top(): INTEGER;
END Stack.
```

# Example: MIS for Stack (2)

- State constants:

  max : INTEGER  [0 $\leq$ max]

- State variables:

  $s$ : lists[INTEGER]  [initially $s$ = nil]

- Exceptions:

  EmptyStack

  FullStack

- Behavior rules:

  Reset

| Input | Output | Transition | Exception |
|---|---|---|---|
| | | $s'$ = nil | |

# Example: MIS for Stack (3)

`MaxHeight`

| Input | Output | Transition | Exception |
|---|---|---|---|
| | max | | |

`Height`

| Input | Output | Transition | Exception |
|---|---|---|---|
| | $\|s\|$ | | |

`Empty`

| Input | Output | State | Exception |
|---|---|---|---|
| | $\|s\| = 0$ | | |

`Full`

| Input | Output | State | Exception |
|---|---|---|---|
| | $\|s\| = \texttt{max}$ | | |

# Example: MIS for Stack (4)

`Push`

| **Input** | **Output** | **Transition** | **Exception** |
|---|---|---|---|
| $i$ : `INTEGER` | | $s' = \text{cons}(i, s)$ | `Full()` $\Rightarrow$ FullStack |

`Pop`

| **Input** | **Output** | **Transition** | **Exception** |
|---|---|---|---|
| | | $s' = \text{tl}(s)$ | `Empty()` $\Rightarrow$ EmptyStack |

`Top`

| **Input** | **Output** | **Transition** | **Exception** |
|---|---|---|---|
| | $\text{hd}(s)$ | | `Empty()` $\Rightarrow$ EmptyStack |

# Module Structure

- Simple structure:
  - All module interfaces are accessible to all module implementations
  - All modules are indivisible units
- Access structure: Module interfaces are only available to certain module implementations
- Submodule structure: Modules may be decomposed into submodules
  - Example: Modules may contain local modules
- Definitional extension structure:
  - Modules with state are only accessible to their definitional extensions
  - Definitional extensions do not have state and are widely accessible

# References


1. D. Parnas, "On the criteria to be used in decomposing systems into modules", in: D. Hoffman and D. Weiss, *Software Fundamentals*, Addison Wesley, 2001.

2. D. Parnas, P. Clements, and D. Weiss, "The modular structure of complex systems", in: D. Hoffman and D. Weiss, *Software Fundamentals*, Addison Wesley, 2001.