SE 2AA4 Winter 2007

06 Specification

William M. Farmer

Department of Computing and Software
McMaster University

27 March 2007

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.
  - The product is acceptable iff it satisfies specification.
- A specification serves several purposes:
  1. An agreement between client and developer, designer and implementer, etc.
  2. Blueprint for developing the product.
  3. Basis for verifying the correctness of the product.
  4. 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:
  - A specification describes the attributes that the product is required to have.
  - A description describes the attributes that the product actually has.
- 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.

Uses of Specifications

- Statement of the requirements of a product.
  - Requirements specification.
  - Design specification.
  - User's requirements specification.
- Statement of the interface between components.
  - External environment specification.
  - Communication protocols.
  - Module interface specifications.
- Reference point for verification and maintenance.

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 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 $\mathrm{dom}(f) = \{x \in A \mid f(x) \downarrow\}$.
  - f is total if $\mathrm{dom}(f) = A$.
  - f is partial if $\mathrm{dom}(f) \subset A$.
- 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 $f : I \longrightarrow O$ 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 $f : \mathbf{Z} \rightarrow \mathbf{Z}$.

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

- 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 partial.

Relational Specification

- A relational specification is a pair (R, D) where:
  1. $R \subseteq I \times O$.
  2. $D \subseteq \mathrm{dom}(R) = \{x : I \mid \exists\, y : O \,.\, R(x, y)\} \subseteq I$.
- $f : I \longrightarrow O$ satisfies (R, D) if:
  1. $\forall\, x : I \,.\, x \in \mathrm{dom}(f) \Rightarrow R(x, f(x))$.
  2. $D \subseteq \mathrm{dom}(f)$.
- Example 1: Integer square function $f : \mathbf{Z} \longrightarrow \mathbf{Z}$.

  $R = \{(x, y) \in \mathbf{Z} \times \mathbf{Z} \mid y = x * x\}$.
  $D = \mathbf{Z}$.
- Example 2: Integer square root function $g : \mathbf{Z} \longrightarrow \mathbf{Z}$.

  $R = \{(x, y) \in \mathbf{Z} \times \mathbf{Z} \mid y * y = x\}$.
  $D = \{x \in \mathbf{Z} \mid \exists y : \mathbf{Z} \,.\, y * y = x\} \subseteq \{x : \mathbf{Z} \mid 0 \leq x\}$.

Axiomatic Specification

- An axiomatic specification is a formula $A(f)$:
  - $A(f)$ is an axiom that expresses the behavior of f.
- $g : I \longrightarrow O$ satisfies $A(f)$ if $A(g)$ is true.
- Example 1: Integer square function $f : \mathbf{Z} \rightarrow \mathbf{Z}$.

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

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

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

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 designed 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 $\text{out} \subseteq I \times S \times O$.
  5. A next state relation $\text{ns} \subseteq I \times S \times S$.
- M is a finite state machine if S is finite.
- M is deterministic if the relations are functions, i.e., $\text{out} : I \times S \to O$ and $\text{ns} : I \times S \to S$.

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 $f : I \times S \rightarrow O \times S$ 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:

$$f_1 : I \times S \rightarrow O.$$
$$f_2 : I \times S \rightarrow S.$$

- 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 signature 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: $c : \mathbf{Z} \quad$ [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 φ 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. Services: 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 a Before/After MIS

1. Module name.
2. Imported modules.
3. Interface.
   - Types.
   - Constant signatures.
   - Procedure signatures.
   - Exceptions.
4. State constants with value conditions.
5. State variables with initial values.
6. Behavior rules.
   - Output rules.
   - State transition rules.
   - Exception rules.

Example 3: Before/After MIS for a Stack Data Structure (1)

- Module name: `StackDataStructure`.
- Imported modules: `ElementAdt`.
- Interface:

```
procedure top(): Elt;
procedure height(): Int;
procedure push(e: Elt);
procedure pop();
exception EmptyStack;
exception FullStack;
```

- State constants:

  $\mathsf{max} : \texttt{Int} \quad [0 \leq \mathsf{max}]$

- State variables:

  $s : \mathsf{list}[\texttt{Elt}] \quad [\mathsf{initially}\ s = \mathsf{nil}]$

Example 3: MIS for a Stack Data Structure (2)

- Behavior rules:

`Top`

Input	Output	Transition	Exception
	$\mathrm{hd}(s)$		$s = \mathrm{nil} \leadsto$ `EmptyStack`

`Height`

Input	Output	Transition	Exception
	$\|s\|$		

`Push`

Input	Output	Transition	Exception
e : `Elt`		$s' = \mathrm{cons}(e, s)$	$\|s\| = \max \leadsto$ `FullStack`

`Pop`

Input	Output	Transition	Exception
		$s' = \mathrm{tl}(s)$	$s = \mathrm{nil} \leadsto$ `EmptyStack`

Components of an Axiomatic Input/Output MIS

1. Module name.
2. Imported modules.
3. Interface.
   - Types.
   - Constant signatures.
   - Procedure names and types.
   - Exceptions.
4. Axioms.

Note: An axiomatic input/output MIS has the form of an axiomatic theory (L, Γ) where:

- L is the language defined by the interface of the MIS.
- Γ is the set of axioms of the MIS.

Uses of an Axiomatic Input/Output MIS

1. To design an MID that satisfies the MIS.
2. To explore the abstract behavior of the interface components.
3. To blackbox test that an MID satisfies the MIS.
4. To mathematically verify that an MID satisfies the MIS.

Example 4: Axiomatic Input/Output MIS for a Stacks ADT (1)

- Module name: `StackAdt`.
- Imported modules: `ElementAdt`.
- Interface:

```
type        Stack;
const       bottom: Stack;
procedure   push(e: Elt; s: Stack): Stack;
procedure   top(s: Stack): Elt;
procedure   pop(s: Stack): Stack;
exception   EmptyStack;
```

Example 4: MIS for a Stacks ADT (2)

- Axioms:

  1. Bottom is not a push stack.

     $\forall\, e : \texttt{Elt}, s : \texttt{Stack} \;.\; \texttt{bottom} \neq \texttt{push}(e, s)$

  2. Push is one-to-one.

     $\forall\, e_1, e_2 : \texttt{Elt}, s_1, s_2 : \texttt{Stack} \;.$

     $\quad \texttt{push}(e_1, s_1) = \texttt{push}(e_2, s_2) \Rightarrow (e_1 = e_2 \wedge s_1 = s_2)$

  3. Induction axiom for stacks.

     $\forall\, P : \texttt{Stack} \rightarrow \texttt{Bool} \;.$

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

     $\quad \forall\, s : \texttt{Stack} \;.\; [P(s) \Rightarrow \forall\, e : \texttt{Elt} \;.\; P(\texttt{push}(e, s))]]$

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

  4. Top applied to a push stack.

     $\forall\, e : \texttt{Elt}, s : \texttt{Stack} \;.\; \texttt{top}(\texttt{push}(e, s)) = e$

  5. Pop applied to a push stack.

     $\forall\, e : \texttt{Elt}, s : \texttt{Stack} \;.\; \texttt{pop}(\texttt{push}(e, s)) = s$

Example 4: MIS for a Stacks ADT (3)

6. Bottom has no top.
   $\texttt{top(bottom)}\uparrow \quad [\rightsquigarrow \texttt{EmptyStack}]$
7. Bottom has no pop.
   $\texttt{pop(bottom)}\uparrow \quad [\rightsquigarrow \texttt{EmptyStack}]$

Example 5: MIS for a Lists ADT (1)

- Module name: `ListAdt`.
- Imported modules: `ElementAdt`.
- Interface:

```
type        List;
const       nil: List;
procedure   cons(e: Elt; k: List): List;
procedure   member(i: Int, k: List): Elt;
procedure   take(i: Int, k: List): List;
procedure   drop(i: Int, k: List): List;
exception   BadIndex;
```

Example 5: MIS for a Lists ADT (2)

- Axioms:

  1. Nil is not a cons list.
     $\forall\, e : \texttt{Elt}, k : \texttt{List} \;.\; \texttt{nil} \neq \texttt{cons}(e, k)$
  2. Cons is one-to-one.
     $\forall\, e_1, e_2 : \texttt{Elt}, k_1, k_2 : \texttt{List} \;.$
     $\quad \texttt{cons}(e_1, k_1) = \texttt{cons}(e_2, k_2) \Rightarrow (e_1 = e_2 \;\; \& \;\; k_1 = k_2)$
  3. Induction axiom for lists.
     $\forall\, P : \texttt{List} \rightarrow \texttt{Bool} \;.$
     $\quad [P(\texttt{nil}) \;\; \&$
     $\quad \forall\, k : \texttt{List} \;.\; [P(k) \Rightarrow \forall\, e : \texttt{Elt} \;.\; P(\texttt{cons}(e, k))]]$
     $\quad \Rightarrow \forall\, k : \texttt{List} \;.\; P(k)$
  4. Membership with respect to nil.
     $\forall\, i : \texttt{Int} \;.\; \texttt{Member}(i, \texttt{nil}) \uparrow \quad [\rightsquigarrow \texttt{BadIndex}]$

Example 5: MIS for a Lists ADT (3)

5. Membership with respect to cons.

   $\forall\, i : \texttt{Int}, e : \texttt{Elt}, k : \texttt{List}\ .$

   $\quad \texttt{member}(i, \texttt{cons}(e, k)) \simeq$

   $\quad\quad \mathsf{if}(i < 0,$

   $\quad\quad\quad \bot \quad [\leadsto \texttt{BadIndex}],$

   $\quad\quad\quad \mathsf{if}(i = 0, e, \texttt{member}(i - 1, k)))$

6. Membership with respect to take.

   $\forall\, i, j : \texttt{Int}, k : \texttt{List}\ .$

   $\quad \texttt{member}(i, \texttt{take}(j, k)) \simeq$

   $\quad\quad \mathsf{if}(i < 0,$

   $\quad\quad\quad \bot \ [\leadsto \texttt{BadIndex}],$

   $\quad\quad\quad \mathsf{if}(i < j,$

   $\quad\quad\quad\quad \texttt{member}(i, k),$

   $\quad\quad\quad\quad \bot \ [\leadsto \texttt{BadIndex}]))$

Example 5: MIS for a Lists ADT (4)

7. Membership with respect to drop.

$$\forall\, i, j : \texttt{Int}, k : \texttt{List}\ .$$
$$\quad \texttt{member}(i, \texttt{drop}(j, k)) \simeq$$
$$\quad\quad \text{if}(i < 0,$$
$$\quad\quad\quad \bot \;\; [\rightsquigarrow \texttt{BadIndex}],$$
$$\quad\quad\quad \texttt{member}(j + i, k))$$