SE 2A04 Fall 2000

05. Specification

Instructor: W. M. Farmer

Revised: 27 November 2000

Software Specifications

- A **software specification** describes what a software product is required to do
- Serve three purposes:
  - Blueprint for developing the product
  - Document that describes the product
  - Basis for verifying the correctness of the product
- There is usually a collection of various kinds of specifications associated with a software product

Ways of Classifying Specifications

- Requirements specification vs.
  Product description
- External (blackbox) vs.
  Internal (whitebox)
- Informal (written in a natural language like English) vs.
  Formal (written in a logic like first-order predicate logic using mathematical and logical concepts and notation)
- State is relevant vs.
  State is not relevant

General Recommendations

- Be precise
  - Use unambiguous language
  - Use mathematical and logical concepts and notation
- Be accurate
  - Do not leave anything out that is relevant
  - Make sure the specification is faithful to reality
- Be consistent
  - Make sure assumptions are free of contradictions
  - Standardize structure, vocabulary, and notation
- Be as clear as possible
  - Choose the right level of abstraction
  - Use both informal and formal language
  - When appropriate, put information in tabular form

Software Specification Models

- For procedures in which state is not relevant
  - Function description
  - Relation description
- For procedures in which state is relevant
  - Pre- and postconditions specification
  - State transition specification
- For higher-order data
  - Abstract data type specification
- For all entities
  - Fundamental model: axiomatic theory

Partiality in 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**

Function Description

1. Intended domain D: possible set of inputs
2. Intended range R: possible set of outputs
3. Function $F : D \to R$ of inputs to outputs expressed, for example, as
   - A function abstraction $(\lambda\, x \in D \,.\, s(x))$
   - A table

Notes:

- Cartesian product is used for a fixed multiple number of inputs or outputs
- Set of finite sequences is used for a variable multiple number of inputs or outputs
- F may be partial or total

Relation Description

1. Intended domain A: possible set of inputs
2. Intended range B: possible set of outputs
3. Relation $R \subseteq A \times B$ between inputs and outputs expressed, for example, as
   - A set abstraction $\{(a, b) \in A \times B : \varphi(a, b)\}$
   - A table

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.

State Transition Specification

- A **state transition specification** consists of:
  1. A set of **state variables**
  2. A set of **transition functions**
     - Modify the values of the state variables
     - Possibly produce an output value
- The specification represents a finite state machine:
  - A state is any set of state variable values
  - An input or output is any input or output of a transition function
  - The next state and output functions are formed from the union of the transition functions
- State transition specifications are especially good for specifying modules in which state is relevant

Abstract Data Type Specification

- An **abstract data type** is a set of compound data and associated operators
- Components of an abstract data type specification:
  1. **New type** of data
  2. **Base types** from which the new type is constructed
  3. **Constructors** for building members of the new type from members of the base types and new type
  4. **Selectors** for choosing subcomponents of a member of the new type
- In object-oriented programming, abstract data types are specified by classes
- A member of an abstract data type can be implemented by a module or object

The Fundamental Model: Axiomatic Theory

- **Axiomatic theory** = formal language + set of axioms
- **Language**: vocabulary for objects and their properties
  - Has a precise semantics
  - Can be used to describe multiple situations
- **Axioms**: assumptions about the objects and properties
  - Specify a class of models
  - Basis for proving theorems
- Benefits:
  - Conceptual clarity: inessential details are omitted
  - Generality: theorems hold in all models

Special Cases of the Fundamental Model

Special cases of the Fundamental Model are obtained by choosing particular background logics:

- Propositional logic
- Equational logic (for algebraic specifications)
- First-order predicate logic
  - Many-sorted first-order logic
  - Partial first-order logic
- Set theory (over first-order predicate logic)
- Simple type theory (higher-order predicate logic)
  - PVS logic
  - IMPS logic

A Theory of Stacks: Language

Underlying logic: Partial many-typed first-order logic

Types: Element, Stack

Constants: bottom : Stack

Function symbols:

push : Element × Stack → Stack
top : Stack → Element
pop : Stack → Stack

A Theory of Stacks: Axioms

1. **Bottom is not a push stack.**
   $\forall e : \text{Element}, s : \text{Stack} \,.\, \text{bottom} \neq \text{push}(e, s)$

2. **Push is one-to-one.**
   $\forall e_1, e_2 : \text{Element}, s_1, s_2 : \text{Stack} \,.$
   $\quad \text{push}(e_1, s_1) = \text{push}(e_2, s_2) \Rightarrow (e_1 = e_2 \wedge s_1 = s_2)$

3. **Induction axiom schema for stacks.**
   $(P(\text{bottom}) \wedge$
   $\quad \forall s : \text{Stack} \,.\, P(s) \Rightarrow \forall e : \text{Element} \,.\, P(\text{push}(e, s)))$
   $\quad \Rightarrow \forall s : \text{Stack} \,.\, P(s)$
   for each predicate P

A Theory of Stacks: Axioms (cont.)

4. **Top applied to a push stack.**
   $\forall e : \text{Element}, s : \text{Stack} \,.\, \text{top}(\text{push}(e, s)) = e$

5. **Pop applied to a push stack.**
   $\forall e : \text{Element}, s : \text{Stack} \,.\, \text{pop}(\text{push}(e, s)) = s$

6. **Bottom has no top.**
   $\text{top}(\text{bottom})\uparrow$

7. **Bottom has no pop.**
   $\text{pop}(\text{bottom})\uparrow$

Theory Interpretations

- A **translation** Φ from a theory T to a theory T' is a pair of functions F and G such that:
  - F maps the objects of T to objects of T'
  - G maps the expressions of T to expressions of T'
- Φ is an **interpretation** if it maps the theorems of T to theorems of T'
- **Interpretations are information conduits!**
- For specifications T and T' expressed as theories, T' satisfies T if there is an interpretation of T in T'