Object-Oriented Programming (OOP)

- One kind of module, ADT, called class
- A class exports operations to manipulate instance objects
  - Often called methods
  - For example, push(), pop(), empty() for Stack
- Instance objects are accessible via references
- *Class* acts as a kind of *Type*, and *Objects* of the *Class* are really variables of that *Type*
  - Let's create an object, Stack S1
  - Stack is a type and S1 is a variable (object) of type Stack

Object-Oriented Programming (OOP)

- So, if we have Class Stack, with method push(int v), then we can instantiate 2 objects of stacks, by
  - Stack S1, S2; (instantiates two empty stacks) AND
  - S1.push(3); S2.push(6); pushes the number 3 onto stack S1 and pushes 6 onto stack S2.
- These stacks are completely independent – they have their own storage areas
- With OOP, we can instantiate as many independent objects as we need, all based on the same class definition.

Syntactic Changes in OOP

- No need to export opaque types (like type STACK = ?)
  - Class name used to declare objects (e.g., Stack S1)
  - Allows dynamic creation of objects based on defined cardinality
- Object references
  - When you declare objects, they are accessed through references
  - For instance:
    - **Home House, Condo;** declares two objects (**House** and **Condo**).
    - **House.rooms;** means we're accessing the attribute rooms of the object **House**
    - **Condo.rooms.furniture.move();** shows changing: you access an object inside another object (**rooms**), then insde that (**furniture**), and finally call its method **move()**
  - Allows us to easily access properties of a declared class (both attributes and operations)

A Further Relation: Inheritance

- Classes may be organized in a hierarchical structure in OOP
- For instance, let a new class B be the child class of A
  - As soon as B is created, it will acquire the attributes and behaviours from the parent class A
- Class B *specializes* class A
  - B inherits from A
- A *generalizes* B
- A is a **superclass** of B
- B is a **subclass** of A

A Further Relation: Inheritance - Example

```
class EMPLOYEE
exports
        function FIRST_NAME(): string_of_char;
        function LAST_NAME(): string_of_char;
        function AGE(): natural;
        function WHERE(): SITE;
        function SALARY: MONEY;
        procedure HIRE (FIRST_N: string_of_char;
                        LAST_N: string_of_char;
                        INIT_SALARY: MONEY);
        Initializes a new EMPLOYEE, assigning a new identifier.
        procedure FIRE();
        procedure ASSIGN (S: SITE);
        An employee cannot be assigned to a SITE if already assigned to it (i.e., WHERE
        must be different from S). It is the client's responsibility to  ensure this. The effect is to
        delete the employee from those in WHERE, add the employee to those in S, generate
        a new id card with security code to access the site overnight, and update WHERE.
end EMPLOYEE
```

A Further Relation: Inheritance – Example

```
class ADMINISTRATIVE_STAFF inherits EMPLOYEE
exports
        procedure DO_THIS (F: FOLDER);
        This is an additional operation that is specific to
        administrators; other operations may also be added.
end ADMINISTRATIVE_STAFF

class TECHNICAL_STAFF inherits EMPLOYEE
exports
        function GET_SKILL(): SKILL;
        procedure DEF_SKILL (SK: SKILL);
        These are additional operations that are specific to
        technicians; other operations may also be added.
end TECHNICAL_STAFF
```

Inheritance

- Classes may be organized in a hierarchical structure in OOP
- A way of building software incrementally
- A subclass defines a subtype
  - subtype is *substitutable* for parent type
- Polymorphism
  - a variable referring to type A can refer to an object of type B if B is a subclass of A
- Dynamic binding
  - the method invoked through a reference depends on the type of the object associated with the reference at runtime

Let’s go back to Documentation and Notation

Notation and Documentation for the MIS

- The MIS can be documented in natural language or formally, but it is **key** that it is documented so that both *syntax* and *semantics* are specified
- Essentials:
  - Overview of services provided by the module
  - Syntax of the interface (exported types, constants and access programs)
  - Explanations of types and constants
  - For each access program, description of its black-box behaviour, not how it is implemented. Assumptions and limitations must also be described.
    - An excellent way of describing black-box behaviour of the access program is to describe the relation between outputs and inputs of the program.

Classification of Specifications Styles

- Informal, semi-formal, formal
- Operational
  - Behaviour specification in terms of some abstract machine
  - Often specific to a sequence of steps
- Descriptive
  - Behaviour described in terms of properties
  - Often generalized into a more abstract machine
- This is important as people generally have a bias towards one style or another when documenting product development.

Classification of Specifications Styles: Example

- Specification of a geometric figure E:

E can be drawn as follows:

1. Select two points P1 and P2 on a plane
2. Get a string of a certain length and fix its ends to P1 and P2
3. Position a pencil
4. Move the pen clockwise, keeping the string tightly stretched, until you reach the point where you started drawing

This is an **operational** specification

Classification of Specifications Styles: Example

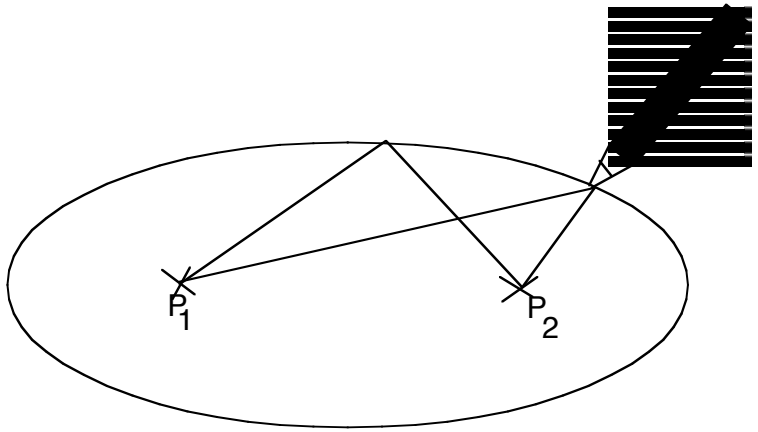
- Specification of a geometric figure E:

Geometric figure E is described by the following equation:

$$ax^2 + by^2 + c = 0$$

Where a, b, and c, are suitable constants

This is a **descriptive** specification



Classification of Specifications Styles: Example 2

This is an **operational** specification

"Let a be an array of n elements. The result of its sorting is an array b of n elements such that the first element of b is the minimum of a (if several elements of a have the same value, any one of them is acceptable); the second element of b is the minimum of the array of n-1 elements obtained from a by removing its minimum element; and so on until all n elements of a have been removed."

This is a **descriptive** specification

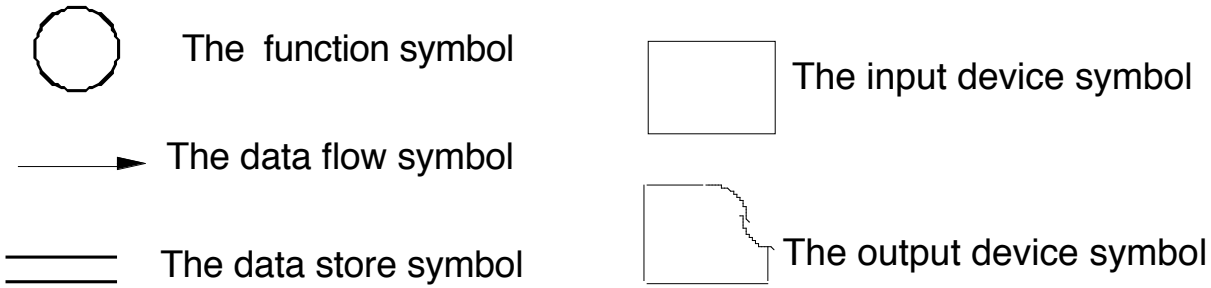
"The result of sorting array a is an array b which is a permutation of a and is sorted."

Data Flow Diagrams (DFDs)

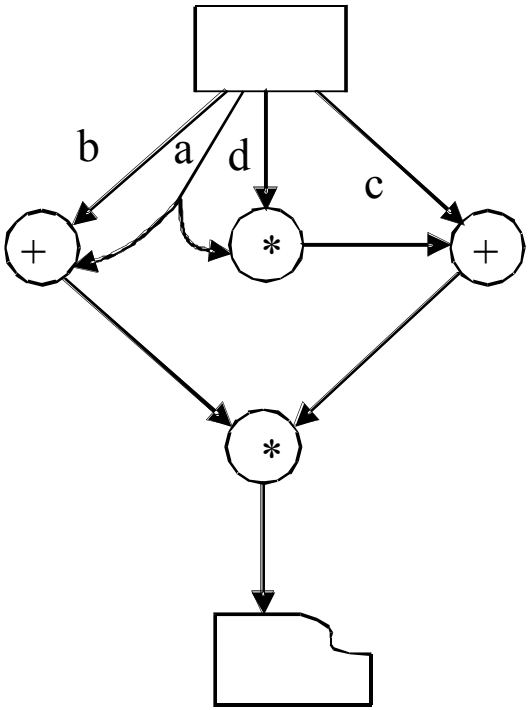
- A semi-formal operational specification
- System viewed as a collection of data manipulated by functions
- Data can be persistent
  - They are stored in *data repositories*
- Data can flow
  - They are represented by *data flows*
- DFDs have a graphical notation

Data Flow Diagrams (DFDs) – Graphical Notation

- *Bubbles* represent functions
- *Arcs* represent data flows
- *Open boxes* represent persistent store
- *Closed boxes* represent I/O interaction

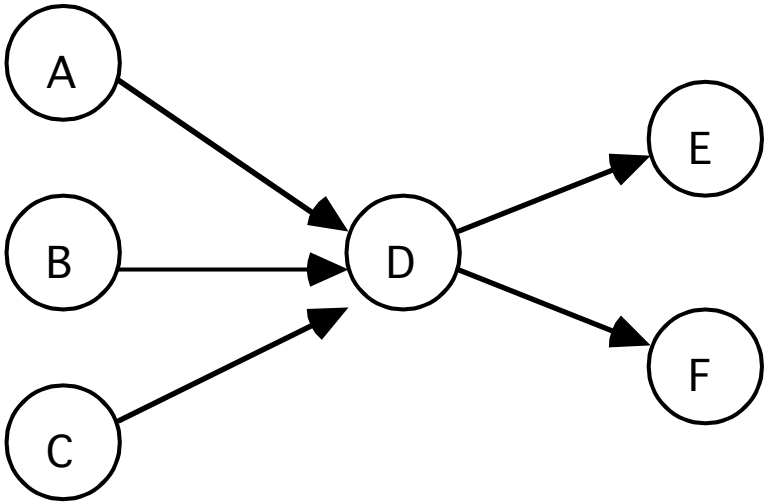


Data Flow Diagrams (DFDs) – Example



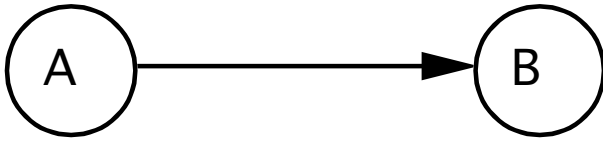
An Evaluation of Data Flow Diagrams (DFDs)

- Easy to read, however,
- There are informal semantics
  - How to define leaf functions?
  - Inherent ambiguities
- Outputs from A, B, C are all needed?
- Outputs for E and F are produced at the same time?



An Evaluation of Data Flow Diagrams (DFDs)

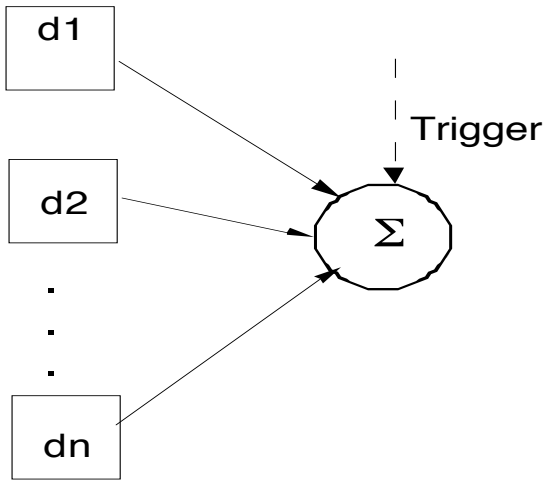
- Control of information is absent



- Possible interpretations:
  - A produces datum, waits until B consumes it
  - B can read the datum many times without consuming it
  - A pipe is inserted between A and B

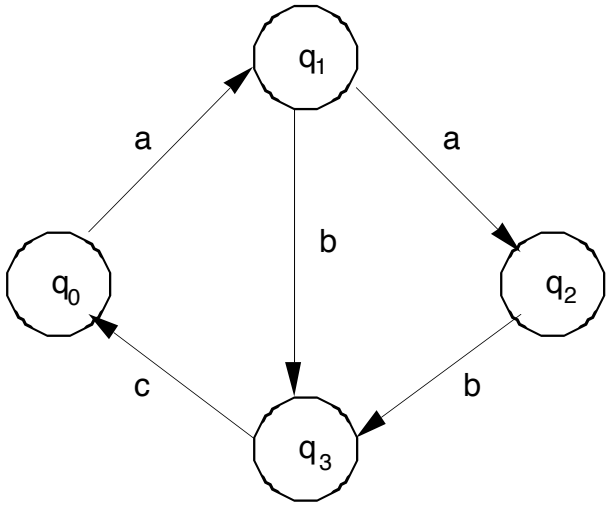
Formalization/Extensions of DFDs

- There have been attempts to formalize DFDs
- There have been attempts to extend DFDs (e.g., for real-time systems)



Finite State Machines (FSMs)

- Can specify control flow aspects, not just data flow
- They are defined as:
  - A finite set of states Q
  - A finite set of inputs I
  - A transition function d: Q x I -> Q telling us how the system moves from one state to another when it sees an input
- The diagram is a State Transition Diagram, showing the possible states and how inputs a, b, c trigger changes.



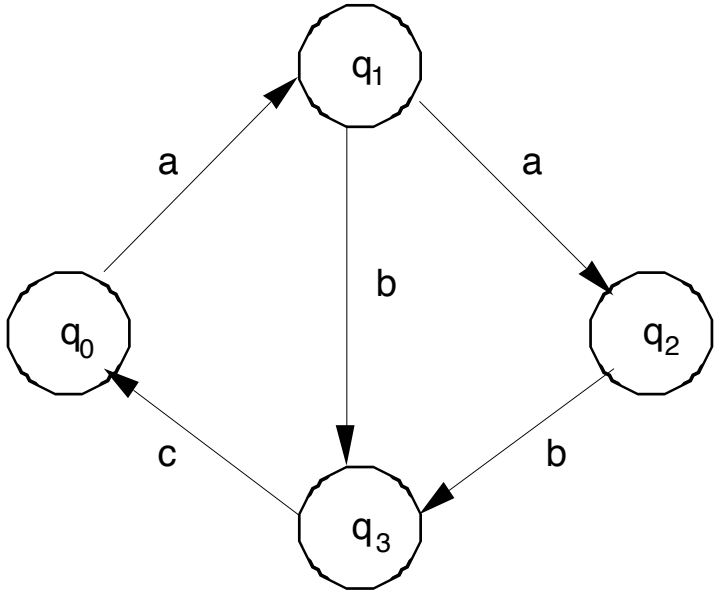
Finite State Machines (FSMs)

State Transition Table

	q_0	q_1	q_2	q_3
a	q_1	q_2	-	-
b	-	q_3	q_3	-
c	-	-	-	q_0

Cannot occur simultaneously

State Transition Diagram

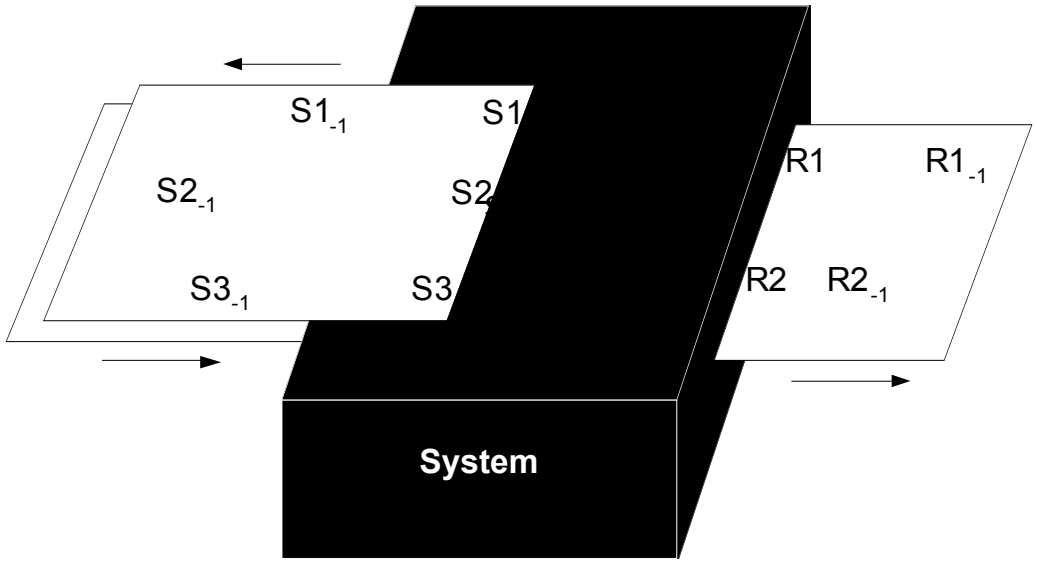


Finite State Machines (FSMs) Limitations

- Finite memory
  - FSM can only remember the current state
- State explosion
  - If you try to compose multiple FSMs, the number of possible combined states grows exponentially
  - Given a number of FSMs with k1, k2, … kn states, their composition is a FSM with k1 * k2 *… * kn. This growth is exponential with the number of FSMs, not linear (we would like it to be k1 + k2 +… + kn)

Mills' Black-Box

Mills, H.D.: Stepwise refinement and verification in box-structured systems. Computer 21 (1988) 23-36



$R = f (S, S_h)$

S is the set of stimuli

S_h the set of stimulus history

R the set of responses

Logic Specifications

Examples of first-order theory (FOT) formulas:

- x > y **and** y > z **implies** x > z
- x = y ≡ y = x
- **for all** x, y, z (x > y **and** y > z **implies** x > z)
- x + 1 < x – 1
- **for all** x (**exists** y (y = x + z))
- x > 3 **or** x < -6

Different Notation

- (x > y) ∧ (y > z) ⟹ x > z
- x = y ≡ y = x
- ∀ (x, y, z) ((x > y) ∧ (y > z) ⟹ x > z)
- x + 1 < x – 1
- ∀ (x) (∃ (y) | (y = x + z))
- (x > 3) ∨ (x < -6)

Note: there are ***lots*** of different logic notations – do not be intimidated just because the notations may look unfamiliar to you

Specifying Complete Programs

- A property, or requirement, for P is specified as a formula of the type

$$\{Pre\ (i_1, i_2, \ldots, i_n)\ \}$$
$$P$$
$$\{Post\ (o_1, o_2, \ldots, o_m, i_1, i_2, \ldots, i_n)\}$$

Pre: precondition, Post: postcondition

Specifying Complete Programs – Example

- Division

$\{\text{exists } z(i_1 = z * i_2)\}$

P

$\{o_1 = i_1 / i_2\}$

Specifying Complete Programs – Example

- Division – a stronger requirement

$\{i_1 > i_2 > 0\}$

P

$\{i_1 = i_2 * o_1$ **and** $o_1 \geq 0$ **and** $o_1 < i_2\}$

Specifying Complete Programs – Example

- Program to compute greatest common divisor

$\{i_1 > 0$ **and** $i_2 > 0\}$

P

$\{($**exists** z_1, z_2 $(i_1 = o * z_1$ **and** $i_2 = o * z_2))$ **and**

not (**exists** h

(**exists** z_1, z_2 $(i_1 = h * z_1$ **and** $i_2 = h * z_2)$ **and** $h > o))\}$

Specifying Procedures

{n > 0} -- n is a constant value
procedure search (table: **in** integer_array; n: **in** integer;
element: **in** integer; found: **out** Boolean);
{found ≡ (**exists** i (1 ≤ i ≤ n **and** table (i) = element))}

Which procedure is this?

{n > 0 }
procedure reverse (a: **in out** integer_array; n: **in** integer);
{**for all** i (1≤ i ≤ n) **implies** (a'(i) = 'a(n - i +1))}

Which procedure is this?

where 'x means value of x on entry to procedure, x' means value of x on exit of procedure

Specifying Classes

- When moving from individual procedures to classes, we need to ensure invariants are always respected.
  - Invariant predicates
  - Pre/Post conditions
- Example of invariant specifying an array, IMPL, used to implement an abstract data type SET
  - **for all** i, j ($1 \leq i \leq$ length **and** $1 \leq j \leq$ length **and** $i \neq j$)
    - **implies** IMPL[i]≠IMPL[j]
    - (no duplicates are stored)

Specifying Classes

- In general, for every operation (op) in the class:

  {Invariant and pre-conditions}
  Program for op
  {Invariant and post-conditions}

- In the Object Oriented world we also have to worry about object initialization
  - i.e. the constructor (cstr), so we also need {true} Program for cstr {Invariant}

Specifying Classes

Consider the earlier example where a class represents the abstract data type SET, and is implemented using the array IMPL of size length. An invariant was given earlier. Now consider what we would specify for an operation DELETE which removes an element x from SET.

{**exists** i (1 ≤ i ≤ length) **and** IMPL[i] = x}

DELETE(x IN integer)

{**for all** i (1 ≤ i ≤ length' **implies** IMPL'[i] ≠ x) **and**

for all i ((1 ≤ i ≤ 'length **and** 'IMPL[i] ≠ x) **implies**

exists j (1 ≤ j ≤ length' **and** IMPL'[j] = 'IMPL[i]))}

Function Tables

- Tabular expressions describe relations through pre and post conditions - ideal for describing behaviour without sequences of operations
- They make it easy to ensure input domain coverage
- They are easy to read and understand (you need just a little practice to write them)
- Coding from tables results in extremely well-structured code
- They facilitate identification of test cases
- Extremely good for real-time/embedded systems
- Disadvantages:
  - We don't know of any way of using tabular expressions to describe typical algorithms.
    - For example, how could we use a table to describe the Gauss Elimination algorithm?
  - As yet, we don't know how to use tables to document the requirements for a GUI, etc.

Side Note: Events vs. Conditions

- Events can be viewed as “pulses” in time - they do not last (retain their values) – we have referred to this as *time-discrete*

- Conditions may retain their values indefinitely – we have referred to this as *time-continuous*