SFWRENG / MECHTRON 3K04
Software Development

Today: High-Level SE

October 6th, 2025

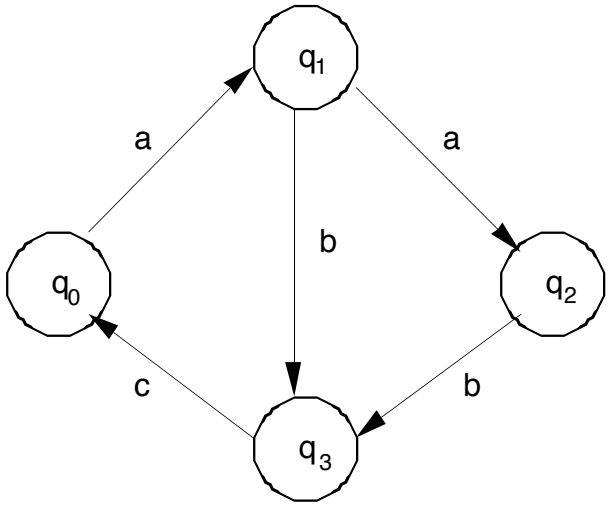
Dr. Angela Zavaleta Bernuy

McMaster University | Engineering
Computing & Software

Brighter World

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.



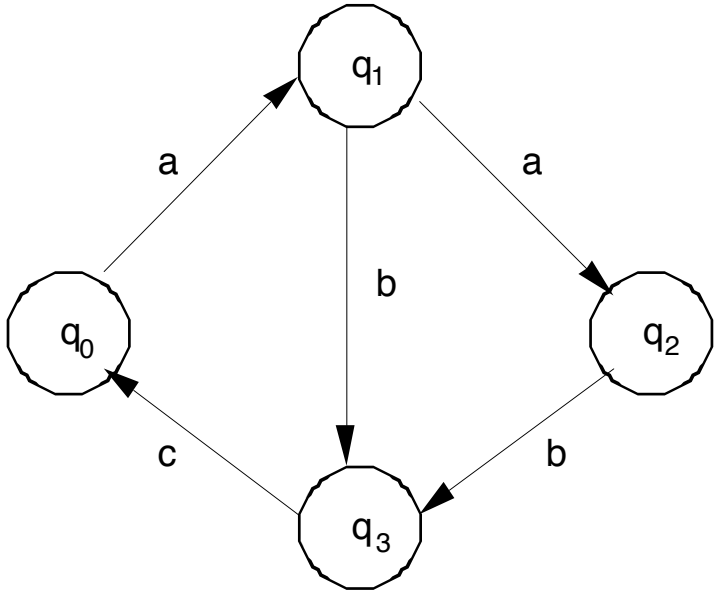
Finite State Machines (FSMs)

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

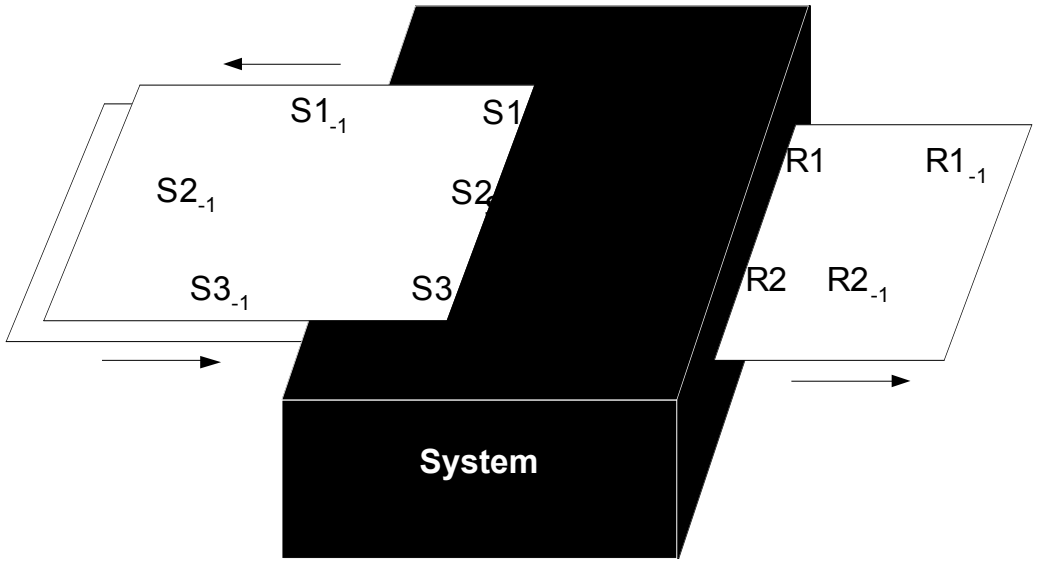


Finite State Machines (FSMs) Limitations

- 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

{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 example1 (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 example2 (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 ≤ i ≤ length **and** 1 ≤ j ≤ length **and** i≠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*

Let’s Review!

Everything up to this point, it is fair game for the Midterm ☺

SFWRENG / MECHTRON 3K04
Software Development

Today: Design

October 8th, 2025

Dr. Angela Zavaleta Bernuy

McMaster University | Engineering
Computing & Software

Brighter World

Design

- Design can be broken into two different branches:
  - Back-end design
  - Front-end design
- Back-end design involves internal design questions
  - What are we building with?
  - How are we putting things together?
- Front-end design involves user facing design questions
  - What will people use our design for?
  - How will they use our design?

Software Architecture and Design

This week, we will cover all these questions:

- What is design?
- How can a system be decomposed into modules?
- What is a module's interface?
- What are the main relationships among modules?
- Prominent software design techniques and information hiding
- Design notations

What is back-end design?

- Provides structure to any artifact
- Decomposes system into parts (*structure*), assigns responsibilities (*behaviour*), ensures that parts fit together to achieve a global goal (*interactions*)
  - Often, but not always the global goal will come from the specifications/requirements for the system
- Design refers to both an activity and the result of the activity
- This emphasizes development focused design

What is back-end design?

- In this context, design is meant for other developers (engineers, business, manufacturers etc.)
- ***Good*** design allows end-users to understand how to use the design for implementation, analysis, and more design
- RDP, MIS, MID are notations that help to document design for easier understanding

Design: Two Meanings

- Design acts as a bridge between *requirements* and the *implementation* of the software
- Design as an activity (the process)
  - Activity that gives a structure to the artifact
  - e.g., a requirements specification document must be *designed*
    - must be given a structure that makes it easy to understand and evolve
  - Requirement: Pacemaker should be robust with respect to change
    - Design should then decide how to build a robust Pacemaker system
- Design as an artifact (the result)
  - Properties of the system
  - Affordances: What can the system do?
  - Signifiers: What does the system tell people it can do?
  - Requirement: Users should be able to tell what pacemaker they are connected to
    - Design should then decide some way to provide feedback to users for determining what pacemaker they are connected to.

Designing Software

Software design is the process of giving structure to the system

Requirements	Software Design Document
Decomposed so domain experts can read and understand it easily	Decomposed so software will be structure for easy maintenance

The Software Design Activity

- Defined as system decomposition into modules
- Produces a Software Design Document (SDD)
  - Describes a system decomposition into modules
- Often *a software architecture* is produced prior to a software design

Software Architecture

- Shows gross structure and organization of the system to be designed
- Its description includes description of…
  - Main components of a system
  - Relationships among those components
  - Rationale for decomposition into its components
  - Constraints that must be respected by any design of the components
- Guides the development of the design