Haroon Rashid

Registration No 16549

Semester 6th

Final Assignment: Software Verification and Validation

Submitted To: Sir Zain Shaukat

Q1. MCQS:

Answer:

1. When should company stop the testing of a particular software?

a. After system testing done
b. It depends on the risks for the system being tested
c. After smoke testing done
d. None of the above

Answer: b

2. White-Box Testing is also known as ________ .

a. Structural testing
b. Code-Based Testing
c. Clear box testing
d. All of the above

Answer: d

3. ___________ refers to a different set of tasks ensures that the software that has been built is traceable to Customer Requirements.

a. Verification
b. Requirement engineering
c. Validation
d. None of the above

Answer: c

4. ________ verifies that all elements mesh properly and overall system functions/performance is achieved.

a. Integration testing
b. Validation testing
c. Unit testing
d. System Testing

Answer: d

5. What do you verify in White Box Testing?
- *Published on 03 Aug 15*

a. Testing of each statement, object and function on an individual basis.
b. Expected output.
c. The flow of specific inputs through the code.
d. All of the above

Answer: d

6. __________ refers to the set of tasks that ensures the software correctly implements a specific function.
- *Published on 03 Aug 15*

a. Verification
b. Validation
c. Modularity
d. None of the above.

Answer: a

7. Who performs the Acceptance Testing?
- *Published on 03 Aug 15*

a. Software Developer
b. End users
c. Testing team
d. Systems engineers

Answer: b

8. Which of the following is not a part of Performance Testing?
- *Published on 30 Jul 15*

a. Measuring Transaction Rate.
b. Measuring Response Time.
c. Measuring the LOC.
d. None of the above.

Answer: c

9. Which of the following can be found using Static Testing Techniques?
- *Published on 29 Jul 15*

a. Defect
b. Failure
c. Both A & B

Answer: a

10. Testing of individual components by the developers are comes under which type of testing?

- Published on 29 Jul 15

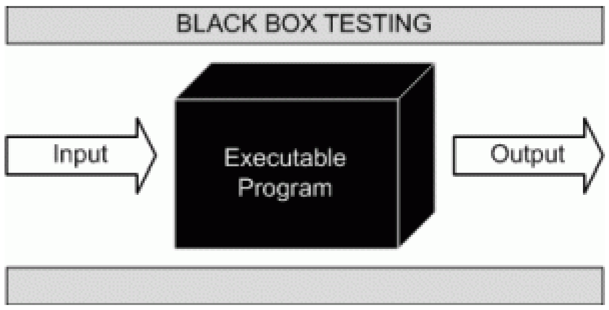
a. Integration testing
b. Validation testing
c. Unit testing
d. None of the above

Answer: c

Q2. Explain Black Box testing and White Box testing in detail.

Answer:

Black Box Testing: BLACK BOX TESTING, also known as Behavioral Testing, is a software testing method in which the internal structure/design/implementation of the item being tested is not known to the tester. These tests can be functional or non-functional, though usually functional.



This method is named so because the software program, in the eyes of the tester, is like a black box; inside which one cannot see. This method attempts to find errors in the following categories:

- Incorrect or missing functions
- Interface errors
- Errors in data structures or external database access
- Behavior or performance errors
- Initialization and termination errors

- **black box testing:** Testing, either functional or non-functional, without reference to the internal structure of the component or system.

- **black box test design technique:** Procedure to derive and/or select test cases based on an analysis of the specification, either functional or non-functional, of a component or system without reference to its internal structure.

Levels Applicable To

Black Box Testing method is applicable to the following levels of software testing:

- Integration Testing
- System Testing
- Acceptance Testing
  The higher the level, and hence the bigger and more complex the box, the more black-box testing method comes into use.

Techniques

Following are some techniques that can be used for designing black box tests.

- **Equivalence Partitioning***:* It is a software test design technique that involves dividing input values into valid and invalid partitions and selecting representative values from each partition as test data.
- **Boundary Value Analysis:** It is a software test design technique that involves the determination of boundaries for input values and selecting values that are at the boundaries and just inside/ outside of the boundaries as test data.
- **Cause-Effect Graphing:** It is a software test design technique that involves identifying the cases (input conditions) and effects (output conditions), producing a Cause-Effect Graph, and generating test cases accordingly.

Advantages

- Tests are done from a user's point of view and will help in exposing discrepancies in the specifications.
- Tester need not know programming languages or how the software has been implemented.
- Tests can be conducted by a body independent from the developers, allowing for an objective perspective and the avoidance of developer-bias.
- Test cases can be designed as soon as the specifications are complete.

Disadvantages

- Only a small number of possible inputs can be tested and many program paths will be left untested.
- Without clear specifications, which is the situation in many projects, test cases will be difficult to design.
- Tests can be redundant if the software designer/developer has already run a test case.
- Ever wondered why a soothsayer closes the eyes when foretelling events? So is almost the case in Black Box Testing.

White Box Testing: WHITE BOX TESTING (also known as Clear Box Testing, Open Box Testing, Glass Box Testing, Transparent Box Testing, Code-Based Testing or Structural Testing) is a software testing method in which the internal structure/design/implementation of the item being tested is known to the tester. The tester chooses inputs to exercise paths through the code and determines the appropriate outputs. Programming know-how and the implementation knowledge is essential. White box testing is testing beyond the user interface and into the nitty-gritty of a system.

This method is named so because the software program, in the eyes of the tester, is like a white/transparent box; inside which one clearly sees.

white-box testing: Testing based on an analysis of the internal structure of the component or system.

white-box test design technique: Procedure to derive and/or select test cases based on an analysis of the internal structure of a component or system.

Example

A tester, usually a developer as well, studies the implementation code of a certain field on a webpage, determines all legal (valid and invalid) AND illegal inputs and verifies the outputs against the expected outcomes, which is also determined by studying the implementation code.

White Box Testing is like the work of a mechanic who examines the engine to see why the car is not moving.

Levels Applicable To

White Box Testing method is applicable to the following levels of software testing:

Unit Testing: For testing paths within a unit.

Integration Testing: For testing paths between units.

System Testing: For testing paths between subsystems.

However, it is mainly applied to Unit Testing.

Advantages

Testing can be commenced at an earlier stage. One need not wait for the GUI to be available.

Testing is more thorough, with the possibility of covering most paths.

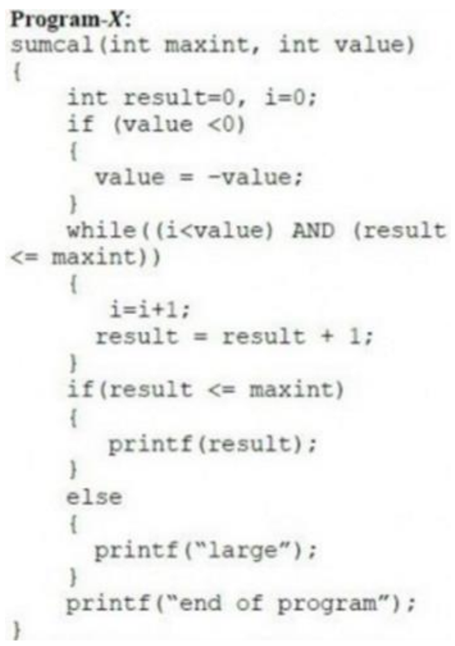
Disadvantages

Since tests can be very complex, highly skilled resources are required, with a thorough knowledge of programming and implementation.

Test script maintenance can be a burden if the implementation changes too frequently.

Since this method of testing is closely tied to the application being tested, tools to cater to every kind of implementation/platform may not be readily available.

Q3. Find the cyclomatic Complexity and draw the Graph of this code.



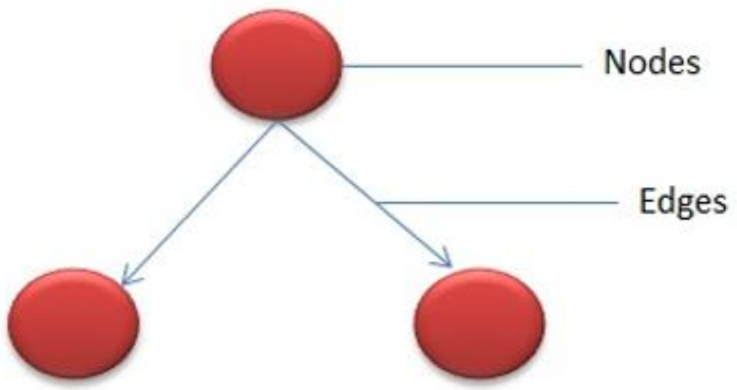
Program-*X*:

```
sumcal(int maxint, int value)
{
    int result=0, i=0;
    if (value <0)
    {
      value = -value;
    }
    while((i<value) AND (result
<= maxint))
    {
       i=i+1;
      result = result + 1;
    }
    if(result <= maxint)
    {
       printf(result);
    }
    else
    {
      printf("large");
    }
    printf("end of program");
}
```

Answer: CYCLOMATIC COMPLEXITY is a software metric used to measure the complexity of a program. It is a quantitative measure of independent paths in the source code of the program. Independent path is defined as a path that has at least one edge which has not been traversed before in any other paths. Cyclomatic complexity can be calculated with respect to functions, modules, methods or classes within a program.

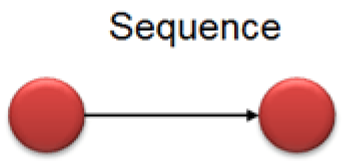
This metric was developed by Thomas J. McCabe in 1976 and it is based on a control flow representation of the program. Control flow depicts a program as a graph which consists of Nodes and Edges.

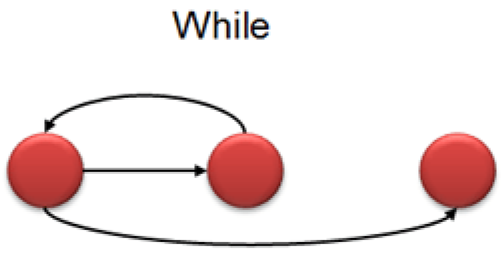
In the graph, Nodes represent processing tasks while edges represent control flow between the nodes.

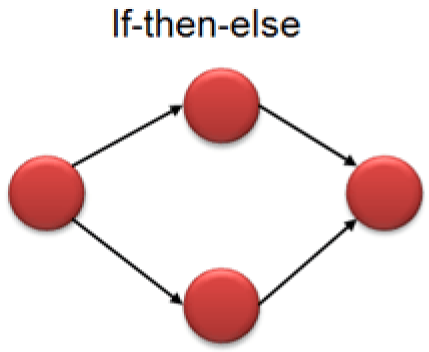


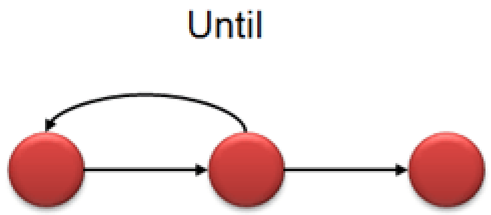
Flow graph notation for a program:

Flow Graph notation for a program defines several nodes connected through the edges. Below are Flow diagrams for statements like if-else, While, until and normal sequence of flow.



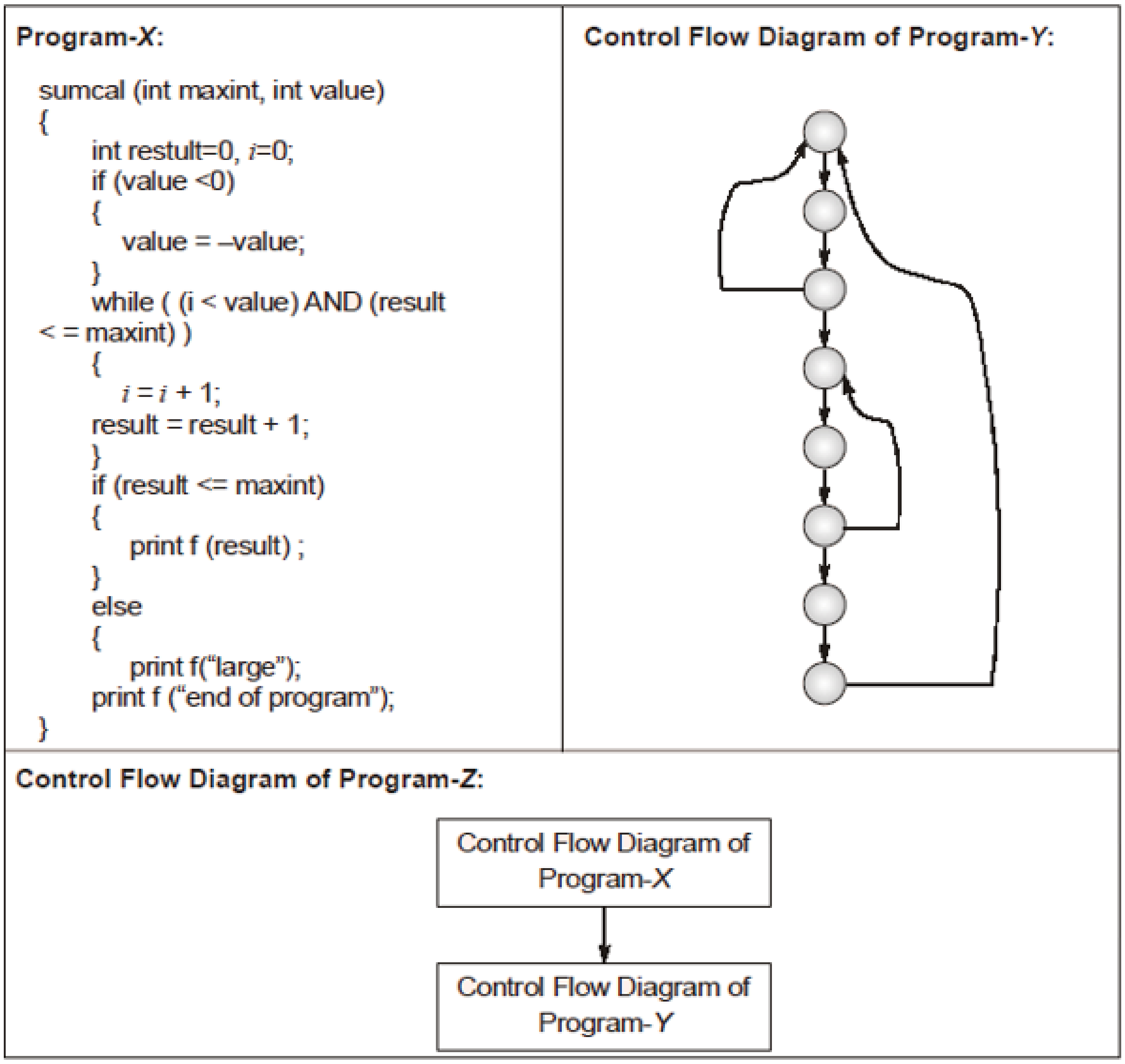






Cyclomatic Complexity and the Graph of the code:

CFG is below


Control Flow Diagram of Program-Y:
Control Flow Diagram of Program-Z:
Control Flow Diagram of Program-X
Control Flow Diagram of Program-Y

Program-*X*:

```
sumcal (int maxint, int value)
{
      int restult=0, i=0;
      if (value <0)
      {
         value = –value;
      }
      while ( (i < value) AND (result
< = maxint) )
      {
         i = i + 1;
      result = result + 1;
      }
      if (result <= maxint)
      {
          print f (result) ;
      }
      else
      {
          print f("large");
      print f ("end of program");
}
```

Cyclomatic Complexity of the above code is as follow:

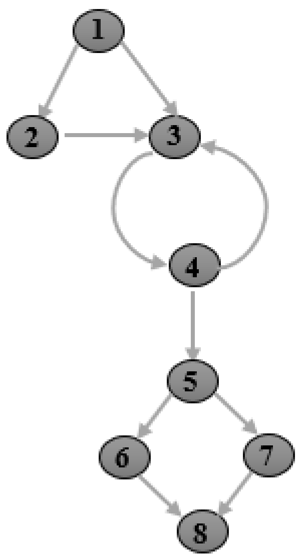
The cyclomatic complexity is the number of loops/conditions +1

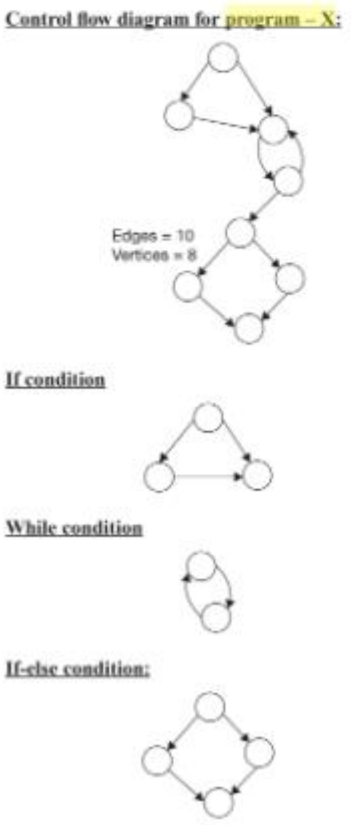
So in the above code there are total 3 conditions in which 2 "if conditions" and " 1 while condition".

There for the V(G) = P+1 = 3+1 = 4

So the cyclomatic complexity for the above code is 4.

Graph: The control flow graph for the above code is bellow:





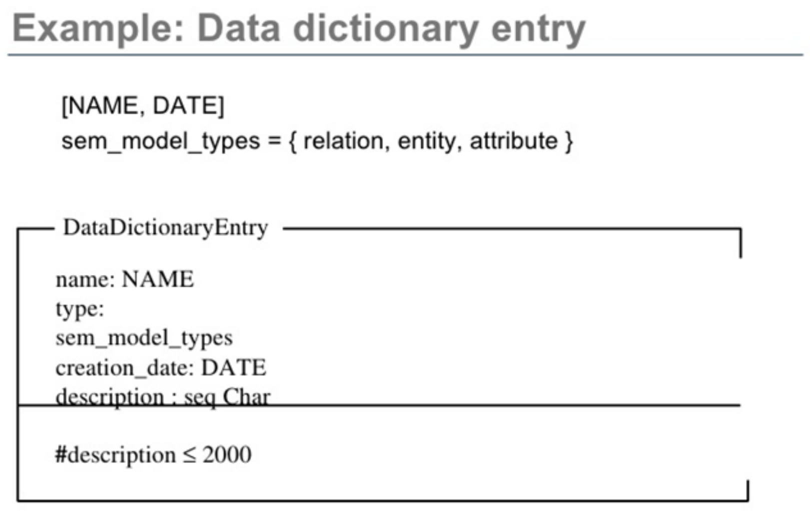
Control flow diagram for program – X:

If condition

While condition

If-else condition:

Q4. What is Z specification and why its is used for, also give some example this code written in Z specification.



Example: Data dictionary entry

[NAME, DATE]
sem_model_types = { relation, entity, attribute }

```
DataDictionaryEntry
name: NAME
type:
sem_model_types
creation_date: DATE
description : seq Char
-----------------------------
#description ≤ 2000
```

Answer:

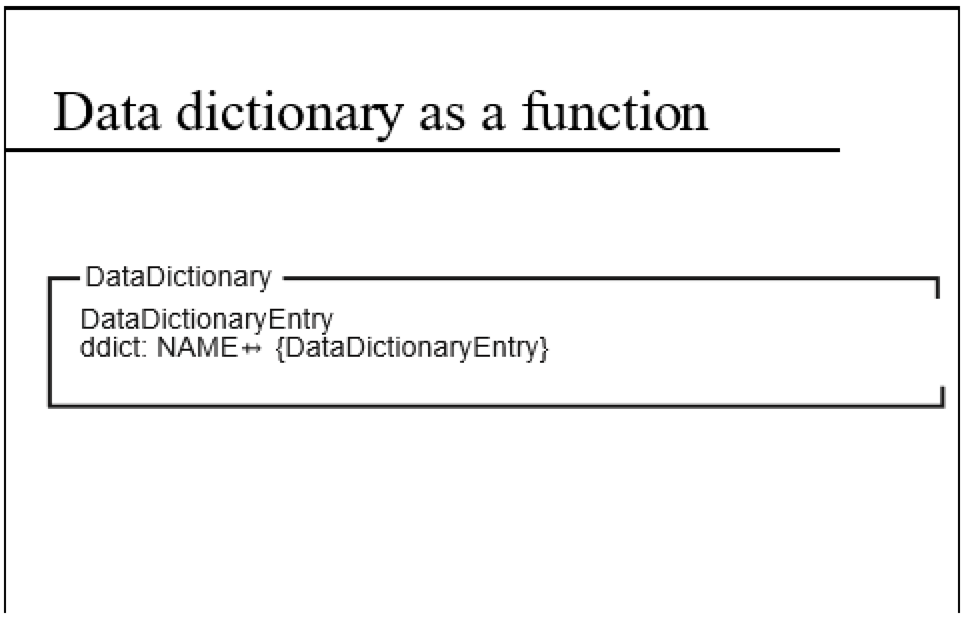
Definition Z specification: Z is a model oriented formal specification language based on Zermelo-Fränkel axiomatic set theory and first order predicate logic. It is a mathematical specification language, with the help of which natural language requirements can be converted into mathematical form.

Introduction:

With the ever-increasing complexity of computer systems, reliable and effective, design and development of high-quality systems that satisfy their requirements is extremely important. In the mission and safety critical system failure can cause cost overrun, loss of lives or even severe economic consequences can arise. So, in such situations, it is necessary that errors are uncovered before software is put into operation. These challenges call for acceptance of proper engineering methods and tools and have motivated the use of formal methods in software engineering. There are varieties of formal specification languages available to fulfill this goal and one way to achieve this goal is by using Z formal specification language. Z is model oriented formal method based on set theory and first order predicate calculus. Description of Z Formal Specification Language: The Z language is a model oriented, formal specification language that was proposed by Jean-Raymond Abrail, Steve Schuman and Betrand Meyer in 1977 and it was later further developed at the programming research group at Oxford University. It is based on Zermelo Fränkel axiomatic set theory and first order predicate logic. The Z notation is a strongly typed, mathematical, specification language. It has robust commercially available tool support for checking Z texts for syntax and type errors in much the same way that a compiler checks code in an executable programming language. It cannot be executed, interpreted or compiled into a running program. It allows specification to be decomposed into small pieces called schemas. The schema is the main feature that distinguishes Z from other formal notations. In Z, both static and dynamic aspects of a system can be described using schemas. The Z specification describes the data model, system state and operations of the system. Z specification is useful for those who find the requirements, those who implement programs to meet those requirements, those who test the consequences, and those who write instruction manuals for the system. Z also helps in refinement towards an implementation by mathematically relating the abstract and concrete states. Z is being used by a wide variety of companies for many different applications. In the Z notation there are two languages: Mathematical Language The mathematical language is used to describe various aspects of a design: objects and the relationships between them by using propositional logic, predicate logic, sets, relation and functions. Schema Language The schema language is used to structure and compose descriptions: collecting pieces of information, encapsulating them, and naming them for reuse.

Why it uses for:

• A Z specification forces the software developer to completely analyze the problem domain. (e.g. identify the state space and pre and post conditions for all operations).
• A Z specification forces all major design decisions to be made prior to coding the implementation. Coding should not commence until you are certain about what you should be coding.
• A Z specification is a valuable tool for generating test data, and the conformance testing of completed systems.
• A Z specification allows formal exploration of properties of system.
• The flexibility to model a specification which can directly lead to the code.
• A large class of structural models can be described in Z without higher – order features, and can thus be analyzed efficiently.
• Independent Conditions can be easily added later.



Data dictionary as a function

```
┌ DataDictionary ──────────────────────
│ DataDictionaryEntry
│ ddict: NAME ↔ {DataDictionaryEntry}
└──────────────────────────────────────
```

Data dictionary - initial state

Init-DataDictionary

DataDictionary'

---

ddict' = $\varnothing$

Add and lookup operations

Add_OK

Δ DataDictionary
name?: NAME
entry?: DataDictionaryEntry

---

name? $\notin$ dom ddict
ddict' = ddict $\cup$ {name? $\mapsto$ entry?}

Lookup_OK

Ξ DataDictionary
name?: NAME
entry!: DataDictionaryEntry

---

name? $\in$ dom ddict
entry! = ddict (name?)

Add and lookup operations

```
Add_Error
Ξ DataDictionary
name?: NAME
error!: seq char
--------------------------------
name? ∈ dom ddict
error! = “Name already in dictionary”
```

```
Lookup_Error
Ξ DataDictionary
name?: NAME
error!: seq char
--------------------------------
name? ∉ dom ddict
error! = “Name not in dictionary”
```

Function over-riding operator

- ReplaceEntry uses the function overriding operator (written ⊕). This adds a new entry or replaces an existing entry.
  - phone = { Ian → 3390, Ray → 3392, Steve → 3427}
  - The domain of phone is {Ian, Ray, Steve} and the range is {3390, 3392, 3427}.
  - newphone = {Steve → 3386, Ron → 3427}
  - phone ⊕ newphone = { Ian → 3390, Ray → 3392, Steve → 3386, Ron → 3427}

Replace operation

Replace_OK

Δ DataDictionary
name?: NAME
entry?: DataDictionaryEntry

---

name? $\in$ dom ddict
ddict' $\oplus$ {name? $\mapsto$ entry?}

Delete entry

Delete_OK

Δ DataDictionary
name?: NAME

---

name? $\in$ dom ddict
ddict' = {name?} $⩤$ ddict

Data dictionary extract operation

- The Extract operation extracts from the data dictionary all those entries whose type is the same as the type input to the operation
- The extracted list is presented in alphabetical order
- A sequence is used to specify the ordered output of Extract

The Extract operation

Extract

DataDictionary
rep!: seq {DataDictionaryEntry}
in_type?: Sem_model_types

$\forall n : \text{dom ddict} \bullet \text{ddict}(n).\text{type} = \text{in\_type?} \Rightarrow \text{ddict}(n) \in \text{rng rep!}$

$\forall i : 1 \leq i \leq \#\text{rep!} \bullet \text{rep!}(i).\text{type} = \text{in\_type?}$

$\forall i : 1 \leq i \leq \#\text{rep!} \bullet \text{rep!}(i) \in \text{rng ddict}$

$\forall i, j : \text{dom rep!} \bullet (i < j) \Rightarrow \text{rep.name}(i) <_{\text{NAME}} \text{rep.name}(j)$

Data dictionary specification

```
┌ The_Data_Dictionary ──────────────────────────
│
│ DataDictionary
│ Init-DataDictionary
│ Add
│ Lookup
│ Delete
│ Replace
│ Extract
└───────────────────────────────────────────────
```