Chapter - 4

TRANSACTION FLOW TESTING

Transaction flow Graphs

- Transaction flows are introduced as a representation of a system's processing.
- The methods that were applied to control flow graphs are then used for functional testing.
- Transaction flows and transaction flow testing are to the independent system tester what control flows and path testing are to the programmer.

Transaction flow Graphs-continued.,

- The transaction flow graph is to create a behavioral model of the program that leads to functional testing.
- The transaction flowgraph is a model of the structure of the system's behavior (functionality).

Transaction Flows

- A transaction is a unit of work seen from a system user's point of view.
- A transaction consists of a sequence of operations, some of which are performed by a system, persons or devices that are outside of the system.
- Transaction begin with Birth-that is they are created as a result of some external act.
- At the conclusion of the transaction's processing, the transaction is no longer in the system.

Example of a Transaction

- A transaction for an online information retrieval system might consist of the following steps or tasks:
  - Accept input (tentative birth)
  - Validate input (birth)
  - Transmit acknowledgement to requester
  - Do input processing
  - Search file
  - Request directions from user
  - Accept input
  - Validate input
  - Process request
  - Update file
  - Transmit output
  - Record transaction in log and clean up (death)

Complications in transaction flow graphs

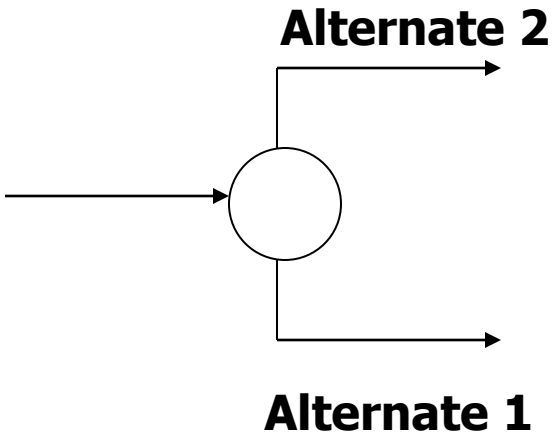
- In simple cases, the transactions have a unique identity from the time they're created to the time they're completed.
- In many systems the transactions can give birth to others, and transactions can also merge.

Births

- There are three different possible interpretations of the decision symbol, or nodes with two or more out links.
  - Decision
  - Biosis
  - Mitosis

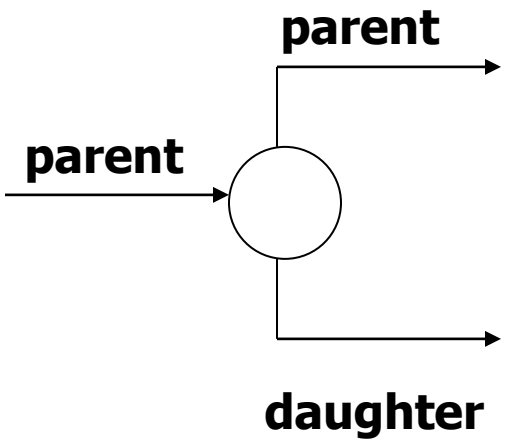
Decision

- Here the transaction will take one alternative or the other alternative but not both



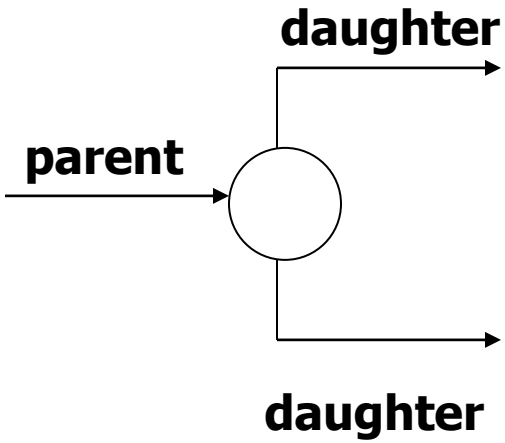
Biosis

- Here the incoming transaction gives birth to a new transaction, and both transaction continue on their separate paths, and the parent retains it identity.



Mitosis

- Here the parent transaction is destroyed and two new transactions are created.

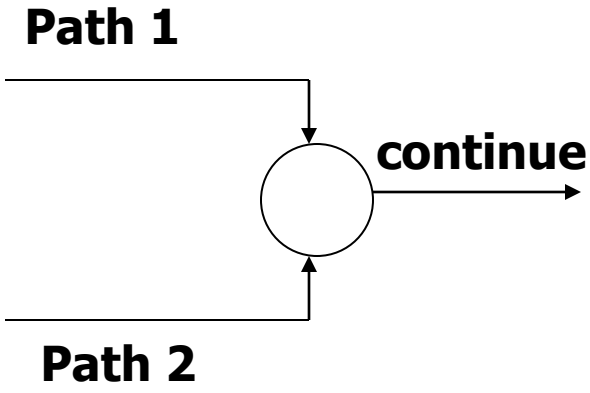


Mergers

- Transaction flow junction points are potentially as troublesome as transaction flow splits.
- There are three types of junctions:
  - Ordinary Junction
  - Absorption
  - conjugation

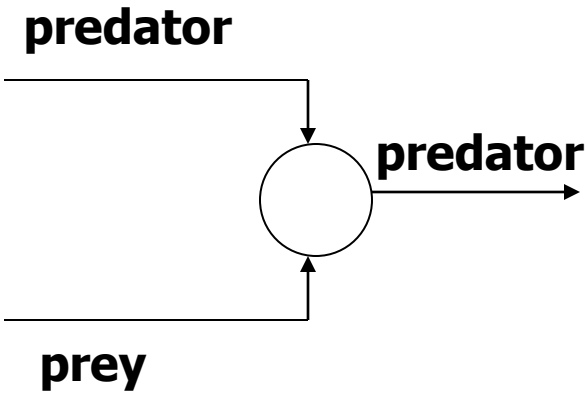
Ordinary junction

- An ordinary junction which is similar to the junction in a control flow graph. A transaction can arrive either on one link or the other.



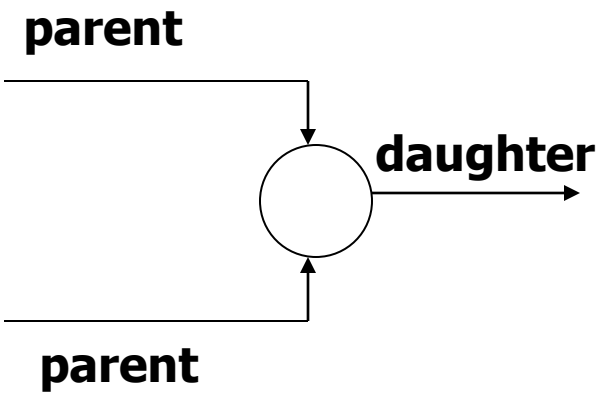
Absorption

- In absorption case, the predator transaction absorbs prey transaction. The prey gone but the predator retains its identity.



Conjugation

- In conjugation case, the two parent transactions merge to form a new daughter. In keeping with the biological flavor this case is called as conjugation.



Transaction flow testing techniques

- Get the transaction flows:

- Complicated systems that process a lot of different, complicated transactions should have explicit representations of the transactions flows, or the equivalent.
- Transaction flows are like control flow graphs, and consequently we should expect to have them in increasing levels of detail.
- The system's design documentation should contain an overview section that details the main transaction flows.
- Detailed transaction flows are a mandatory pre requisite to the rational design of a system's functional test.

Inspections, reviews, walkthroughs

- Transaction flows are natural agenda for system reviews or inspections.
- 1. In conducting the walkthroughs, you should:
  - A. discuss enough transaction types to account for 98%-99% of the transaction the system is expected to process.
  - B. discuss paths through flows in functional rather than technical terms.
  - Ask the designers to relate every flow to the specification and to show how that transaction, directly or indirectly, follows from the requirements.

Inspections, reviews, walkthroughs-continued.,

- 2. make transaction flow testing the corner stone of system functional testing just as path testing is the corner stone of unit testing.
- 3. select additional flow paths for loops, extreme values, and domain boundaries.
- 4. design more test cases to validate all births and deaths.
- 5. publish and distribute the selected test paths through the transaction flows as early as possible so that they will exert the maximum beneficial effect on the project.

Path Selection

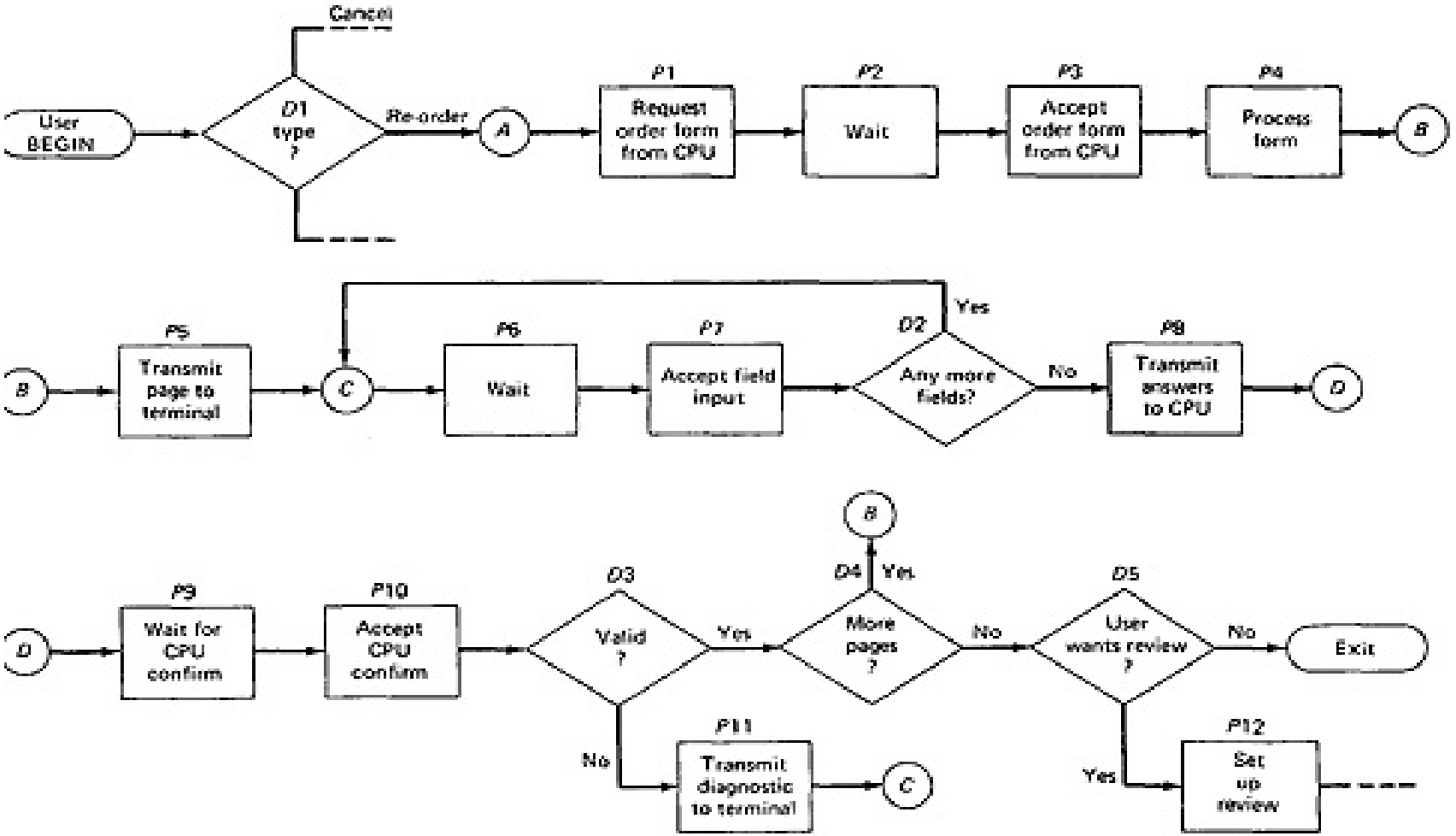
- Select a set of covering paths (c1+c2) using the analogous criteria you used for structural path testing.
- Select a covering set of paths based on functionally sensible transactions as you would for control flow graphs.
- Try to find the most tortuous, longest, strangest path from the entry to the exit of the transaction flow.

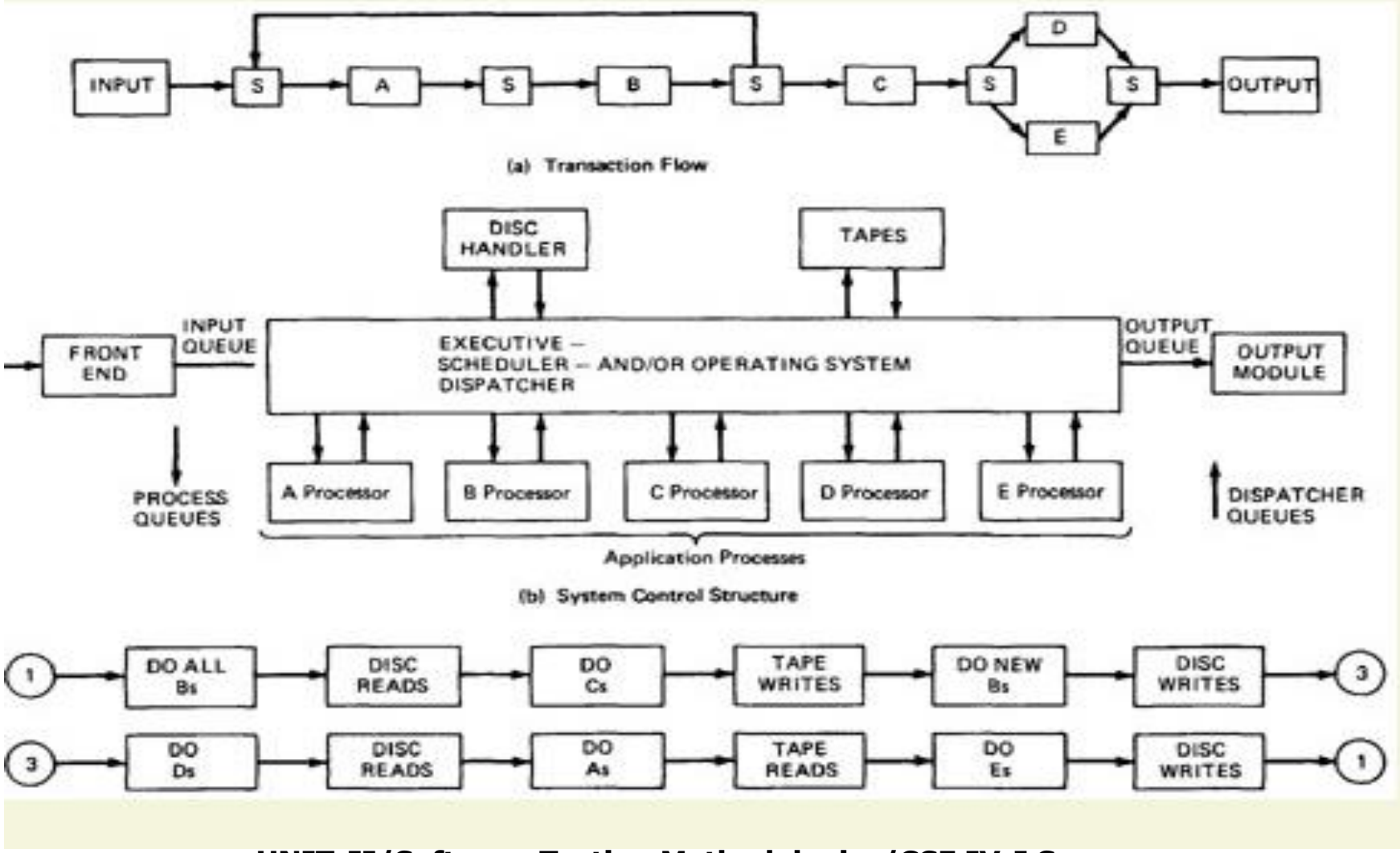
Sensitization

- Most of the normal paths are very easy to sensitize-80% - 95% transaction flow coverage (c1+c2) is usually easy to achieve.
- The remaining small percentage is often very difficult.
- Sensitization is the act of defining the transaction. If there are sensitization problems on the easy paths, then bet on either a bug in transaction flows or a design bug.

Instrumentation

- Instrumentation plays a bigger role in transaction flow testing than in unit path testing.
- The information of the path taken for a given transaction must be kept with that transaction and can be recorded by a central transaction dispatcher or by the individual processing modules.
- In some systems such traces are provided by the operating systems or a running log.


Cancel
User BEGIN
D1 type ?
Re-order
A
P1
Request order form from CPU
P2
Wait
P3
Accept order form from CPU
P4
Process form
B
B
P5
Transmit page to terminal
C
P6
Wait
P7
Accept field input
D2
Any more fields?
Yes
No
P8
Transmit answers to CPU
D
D
P9
Wait for CPU confirm
P10
Accept CPU confirm
D3
Valid ?
Yes
No
B
D4
More pages ?
Yes
No
D5
User wants review ?
No
Exit
Yes
P11
Transmit diagnostic to terminal
C
P12
Set up review


INPUT
S
A
S
B
S
C
S
D
E
S
OUTPUT
(a) Transaction Flow
DISC HANDLER
TAPES
FRONT END
INPUT QUEUE
EXECUTIVE – SCHEDULER – AND/OR OPERATING SYSTEM DISPATCHER
OUTPUT QUEUE
OUTPUT MODULE
PROCESS QUEUES
A Processor
B Processor
C Processor
D Processor
E Processor
DISPATCHER QUEUES
Application Processes
(b) System Control Structure
1
DO ALL Bs
DISC READS
DO Cs
TAPE WRITES
DO NEW Bs
DISC WRITES
3
3
DO Ds
DISC READS
DO As
TAPE READS
DO Es
DISC WRITES
1

Chapter - 5

DATA FLOW TESTING

Data Flow Testing Basics

- Data flow testing is the name given to a family of test strategies based on selecting paths through the program's control flow in order to explore sequences of events related to the status of data objects.
- For example, pick enough paths to assure that every data object has been initialized prior to use or that all defined objects have been used for something.

Motivation

- " it is our belief that, just as one would not feel confident about a program without executing every statement in it as part of some test, one should not feel confident about a program without having seen the effect of using the value produced by each and every computation."

Data flow machines

- There are two types of data flow machines with different architectures.
- Von Neumann machines
- Multi-instruction, multi-data machines (MIMD)

Von Neumann machine Architecture

- Most computers today are von neumann machines.
- This architecture features interchangeable storage of instructions and data in the same memory units.
- The Von Neumann machine Architecture executes one instruction at a time in the following, micro instruction sequence:
  - Fetch instruction from memory
  - Interpret instruction
  - Fetch operands
  - Process or Execute
  - Store result
  - Increment program counter
  - GOTO 1

Multi-instruction, multi-data machines (MIMD) Architecture

- These machines can fetch several instructions and objects in parallel.
- They can also do arithmetic and logical operations simultaneously on different data objects.
- The decision of how to sequence them depends on the compiler.

Data Flow Graphs

- The data flow graph is a graph consisting of nodes and directed links.
- We will use a control graph to show what happens to data objects of interest at that moment.
- Our objective is to expose deviations between the data flows we have and the data flows we want.

Data Object State and Usage

- Data objects can be created, killed and used.
- They can be used in two distinct ways:
  - In a calculation
  - As a part of a control flow predicate
- The following symbols denote these possibilities.
  - d- defined, created, initialized, etc.,
  - k- killed, undefined, released.
  - u- used for some thing.
    - c- used in calculations
    - p- used in a predicate

Defined

- an object is defined explicitly when it appears in a data declaration.
- Or implicitly when it appears on the left hand side of the assignment.
- It is also to be used to mean that a file has been opened.
- A dynamically allocated object has been allocated.
- Something is pushed on to the stack.
- A record written.

Killed or Undefined

- An object is killed on undefined when it is released or otherwise made unavailable.
- When its contents are no longer known.
- Release of dynamically allocated objects back to the availability pool.
- Return of records
- The old top of the stack after it is popped.
- An assignment statement can kill and redefine immediately.

Usage

- A variable is used for computation (c) when it appears on the right hand side of an assignment statement.
- It is used in a Predicate (p) when it appears directly in a predicate.

Data Flow Anomalies

- An anomaly is denoted by a two-character sequence of actions.
- For example, ku means that the object is killed and then used, where as dd means that the object is defined twice without an intervening usage.
- What is an anomaly is depend on the application.

Data Flow Anomalies-continued.,

- There are nine possible two-letter combinations for d, k and u. some are bugs, some are suspicious, and some are okay.
- dd- probably harmless but suspicious. Why define the object twice without an intervening usage.
- dk- probably a bug. Why define the object without using it.
- du- the normal case. The object is defined and then used.
- kd- normal situation. An object is killed and then redefined.
- kk- harmless but probably buggy. Did you want to be sure it was really killed?
- ku- a bug. the object does not exist.
- ud- usually not a bug because the language permits reassignment at almost any time.
- uk- normal situation.
- uu- normal situation.

Data Flow Anomalies-continued.,

- In addition to the two letter situations there are six single letter situations.
- We will use a leading dash to mean that nothing of interest (d,k,u) occurs prior to the action noted along the entry-exit path of interest.
- A trailing dash to mean that nothing happens after the point of interest to the exit.

Data Flow Anomalies-continued.,

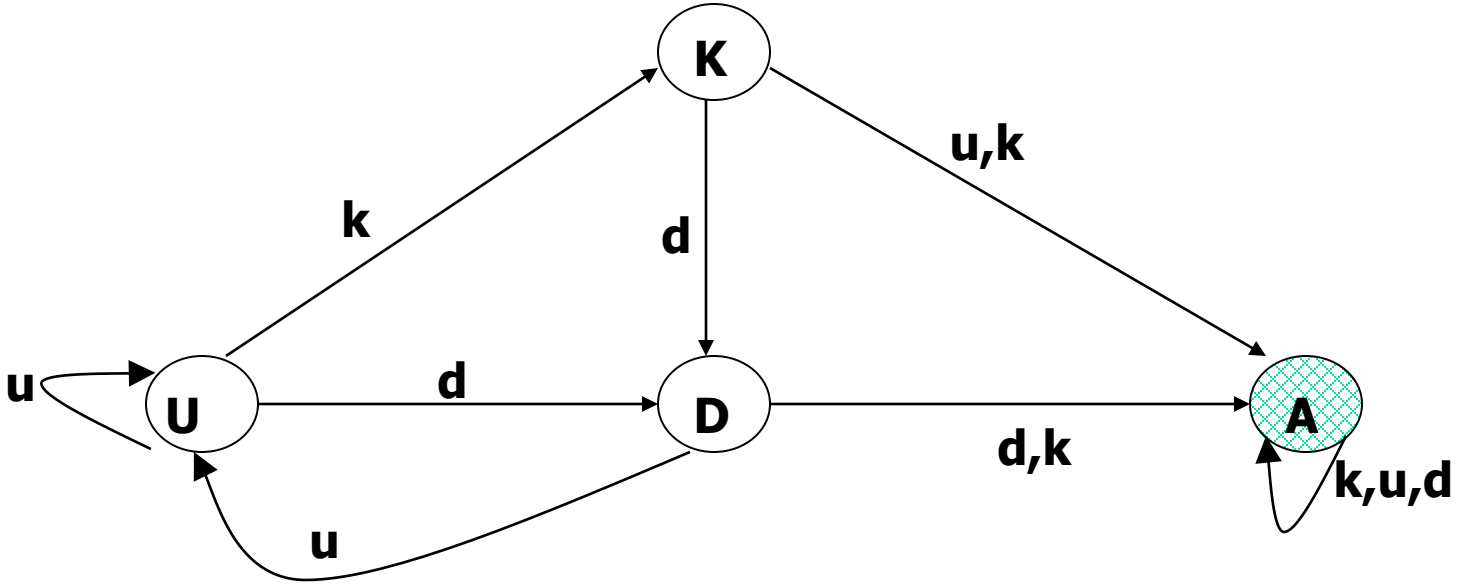
- -k: possibly anomalous because from the entrance to this point on the path, the variable had not been defined. We are killing a variable that does not exist.
- -d: okay. This is just the first definition along this path.
- -u: possibly anomalous. Not anomalous if the variable is global and has been previously defined.
- k-: not anomalous. The last thing done on this path was to kill the variable.
- d-: possibly anomalous. The variable was defined and not used on this path. But this could be a global definition.
- u-: not anomalous. The variable was used but not killed on this path. Although this sequence is not anomalous, it signals a frequent kind of bug. If d and k mean dynamic storage allocation and return respectively, this could be an instance in which a dynamically allocated object was not returned to the pool after use.

Data-Flow Anomaly State Graph

- Data flow anomaly model prescribes that an object can be in one of four distinct states:
  - K- undefined, previously killed, does not exist
  - D- defined but not yet used for anything
  - U- has been used for computation or in predicate
  - A- anomalous
- These capital letters (K,D,U,A) denote the state of the variable and should not be confused with the program action, denoted by lower case letters.

Unforgiving Data-Flow Anomaly State Graph

- Unforgiving model, in which once a variable becomes anomalous it can never return to a state of grace.



Static Vs Dynamic Anomaly Detection

- Static analysis is analysis done on source code without actually executing it.
  - For example: source code syntax error detection is the static analysis result.
- Dynamic analysis is done on the fly as the program is being executed and is based on intermediate values that result from the program's execution.
  - For example: a division by zero warning is the dynamic result.

Static Vs Dynamic Anomaly Detection-continued.,

- If a problem, such as a data flow anomaly, can be detected by static analysis methods, then it does not belongs in testing- it belongs in the language processor.
- There is actually a lot more static analysis for data flow analysis for data flow anomalies going on in current language processors.
- For example, language processors which force variable declarations can detect (–u) and (ku) anomalies.
- But still there are many things for which current notions of static analysis are **inadequate.**

Why isn't static analysis enough?

- there are many things for which current notions of static analysis are **inadequate they are:**
  - Dead Variables.
  - Arrays.
  - Records and pointers.
  - Dynamic subroutine or function name in a call.
  - False anomalies.
  - Recoverable anomalies and alternate state graphs.
  - Concurrency, interrupts, system issues.
- Although static analysis methods have limits, they are worth using and a continuing trend in language processor design has been better static analysis methods for data flow anomaly detection.

The Data Flow Model

- The data flow model is based on the program's control flow graph.
- Here we annotate each link with symbols or sequences of symbols that denote the sequence of data operations on that link with respect to the variable of interest. Such annotations are called **link weights.**
- the control flow graph structure is same for every variable: it is the weights that change.

Components of the Model

- Here are the modeling rules:

- To every statement there is a node, whose name is unique. Every node has at least one outlink and at least one inlink except for exit nodes and entry nodes.
- Exit nodes are dummy nodes placed at the outgoing arrowheads of exit statements to complete the graph. The entry nodes are dummy nodes placed at entry statements for the same reason.
- The outlink of simple statements are weighted by the proper sequence of data flow actions for that statement.
- Predicate nodes are weighted with the p- use on every outlink, appropriate to that outlink.

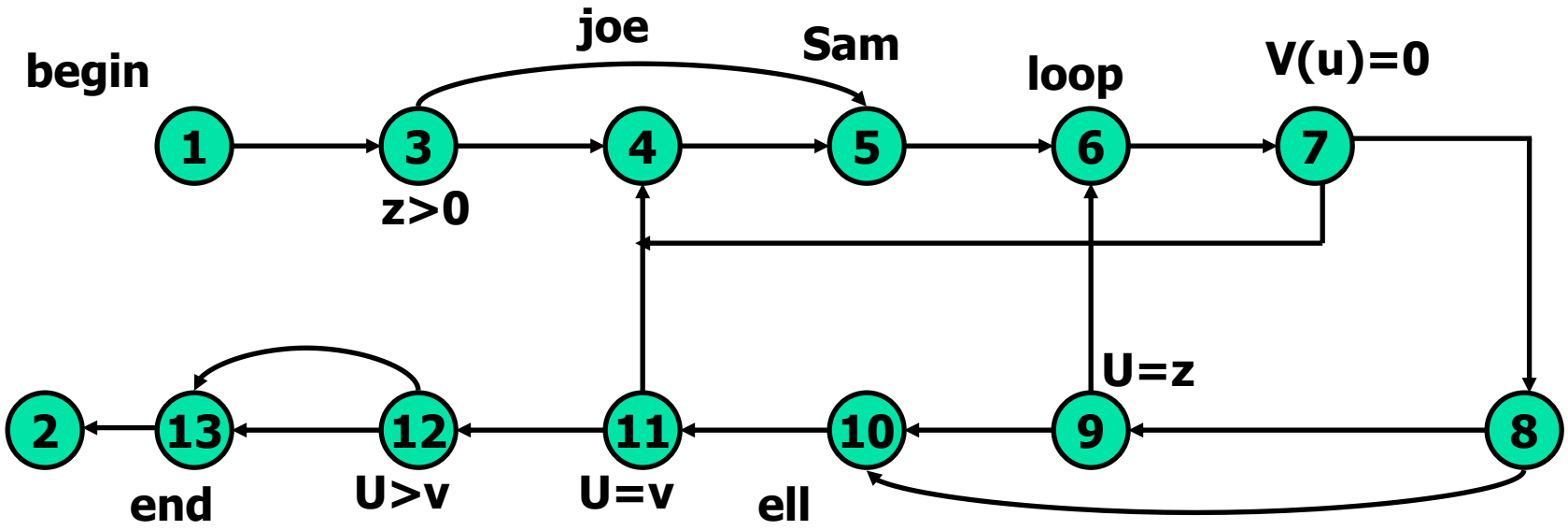
Components of the Model- continued.,

- Every sequence of simple statements can be replaced by a pair of nodes that has, as weights on the link between them, the concatenation of link weights.
- If there are several data flow actions on a given link for a given variable, then the weight of the link is denoted by the sequence of actions on that link for that variable.
- Conversely, a link with several data flow actions on it can be replaced by a succession of equivalent links, each of which has at most one data flow action for any variable.

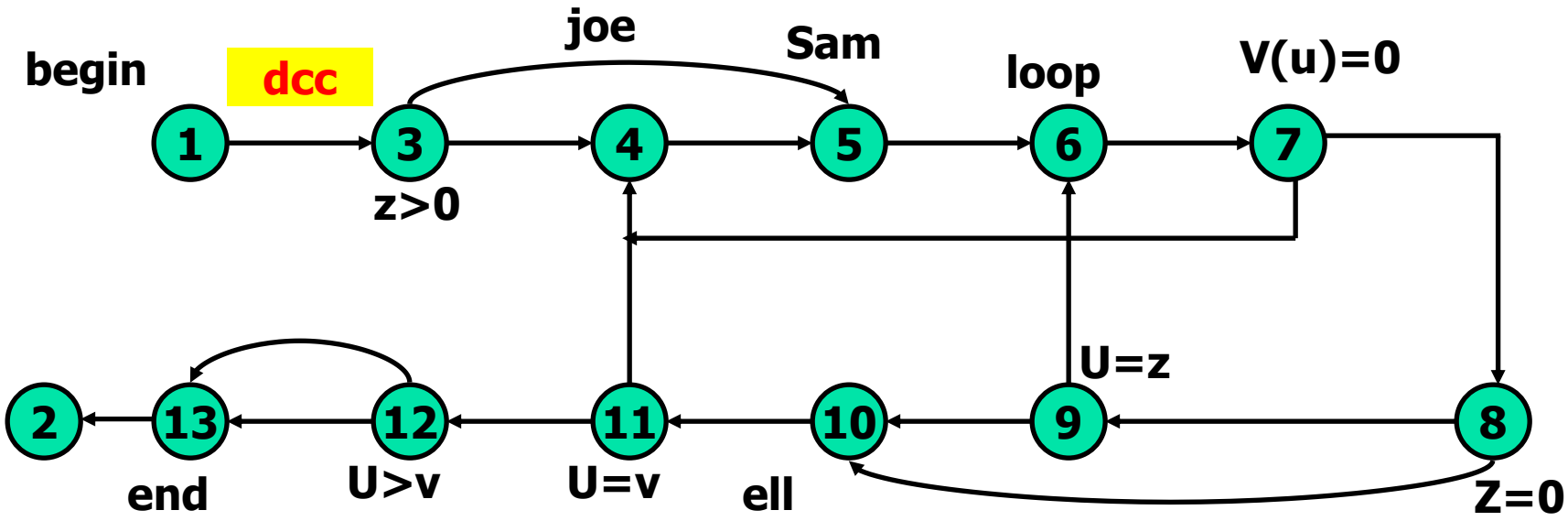
Example-Data Flow Graph

```
        CODE (PDL)
        INPUT X,Y
        Z:=X+Y
        V:=X-Y
        IF Z>0 GOTO SAM
JOE:    Z:=Z-1                    ELL:  Z:=Z-1
SAM:    Z:=Z+V                          IF Z=0 GOTO ELL
        FOR U=0 TO Z                    U:=U+1
        V(U),U(V) :=(Z+V)*U             NEXT U
        IF V(U)=0 GOTO JOE              V(U-1) :=V(U+1)+U(V-1)
                                        V(U+U(V)) :=U+V
                                        IF U=V GOTO JOE
                                        IF U>V THEN U:=Z
                                        Z:=U
                                        END
```

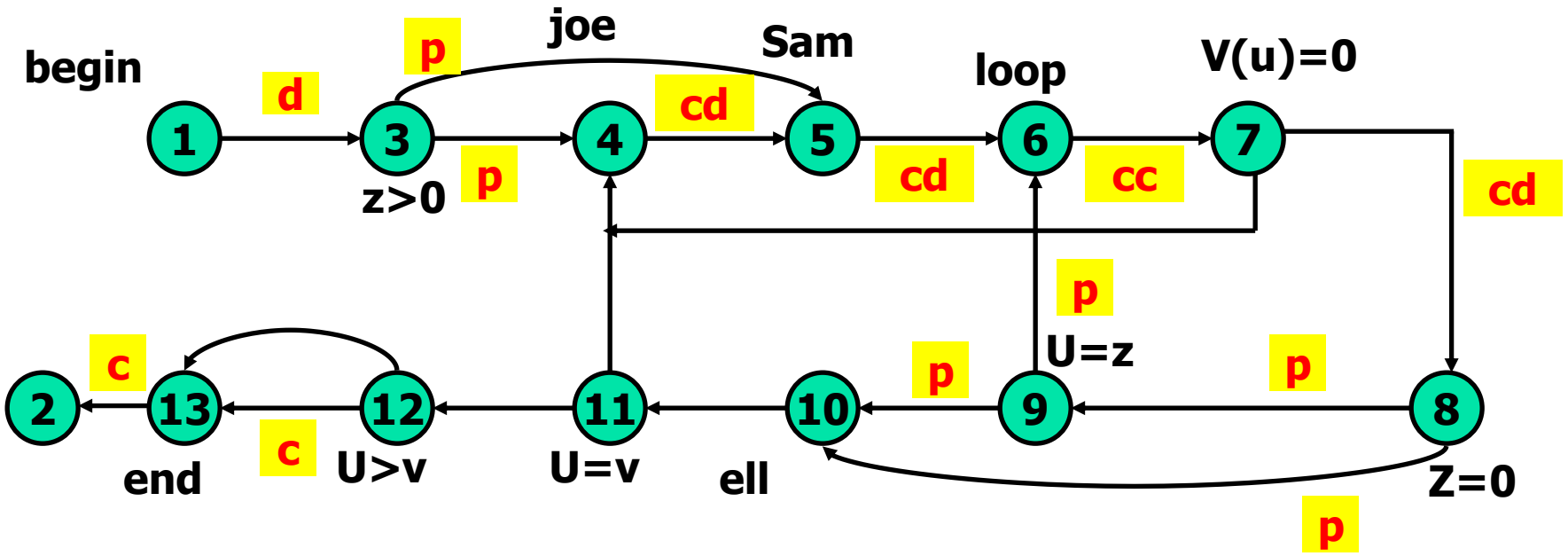
Un annotated Control Flow Graph



Control Flow Graph Annotated for X and Y Data Flows



Control Flow Graph Annotated for Z Data Flow



Data Flow Testing Strategies

- Data Flow Testing Strategies are structural strategies.
- Data Flow Testing Strategies require data flow link weights.
- Data Flow Testing Strategies are based on selecting test path segments (sub paths) that satisfy some characteristic of data flows for all data objects.
- For example, all sub paths that contain a d.

Definition Clear Path Segment

- Definition Clear Path Segment with respect to a variable X is a connected sequence of links such that X is defined on the first link and not redefined or killed on any sub sequent link of that path segment.
- The fact that there is a definition clear sub path between two nodes does not imply that all sub paths between those nodes are definition clear.

Loop Free Path Segment

- A loop free path segment is a path segment for which every node is visited at most once.

A simple path segment

- A simple path segment is a path segment in which at most one node is visited twice.

du path

- A du path from node i to k is a path segment such that if the last link has a computational use of X, then the path is simple and definition clear; if the penultimate node is j-that is, the path is (i.p,q,……r,s,t,j,k) and link (j,k) has a predicate use- then the path from i to j is both loop free and definition clear.

The data flow testing strategies

- Various types of data flow testing strategies in decreasing order of their effectiveness are:
  - All du paths Strategy
  - All uses Strategy
  - All p-uses/some c-uses Strategy
  - All c-uses/some p-uses Strategy
  - All definitions Strategy
  - All predicates uses, all computational uses Strategy

All du paths Strategy

- The all du path (ADUP) strategy is the strongest data flow testing strategy.
- It requires that every du path from every definition of every variable to every use of that definition be exercised under some test.

All Uses Strategy

- The all uses strategy is that at least one definition clear path from every definition of every variable to every use of that definition be exercised under some test.

All P-uses/some c-uses Strategy

- All p-uses/some c-uses (APU+C) strategy is defined as follows: for every variable and every definition of that variable, include at least one definition free path from the definition to every predicate use; if there are definitions of the variables that are not covered by the above prescription, then add computational use test cases as required to cover every definition.

All C-uses/some p-uses Strategy

- The all c-uses/some p-uses strategy (ACU+P) is to first ensure coverage by computational use cases and if any definition is not covered by the previously selected paths, add such predicate use cases as are needed to assure that every definition is included in some test.

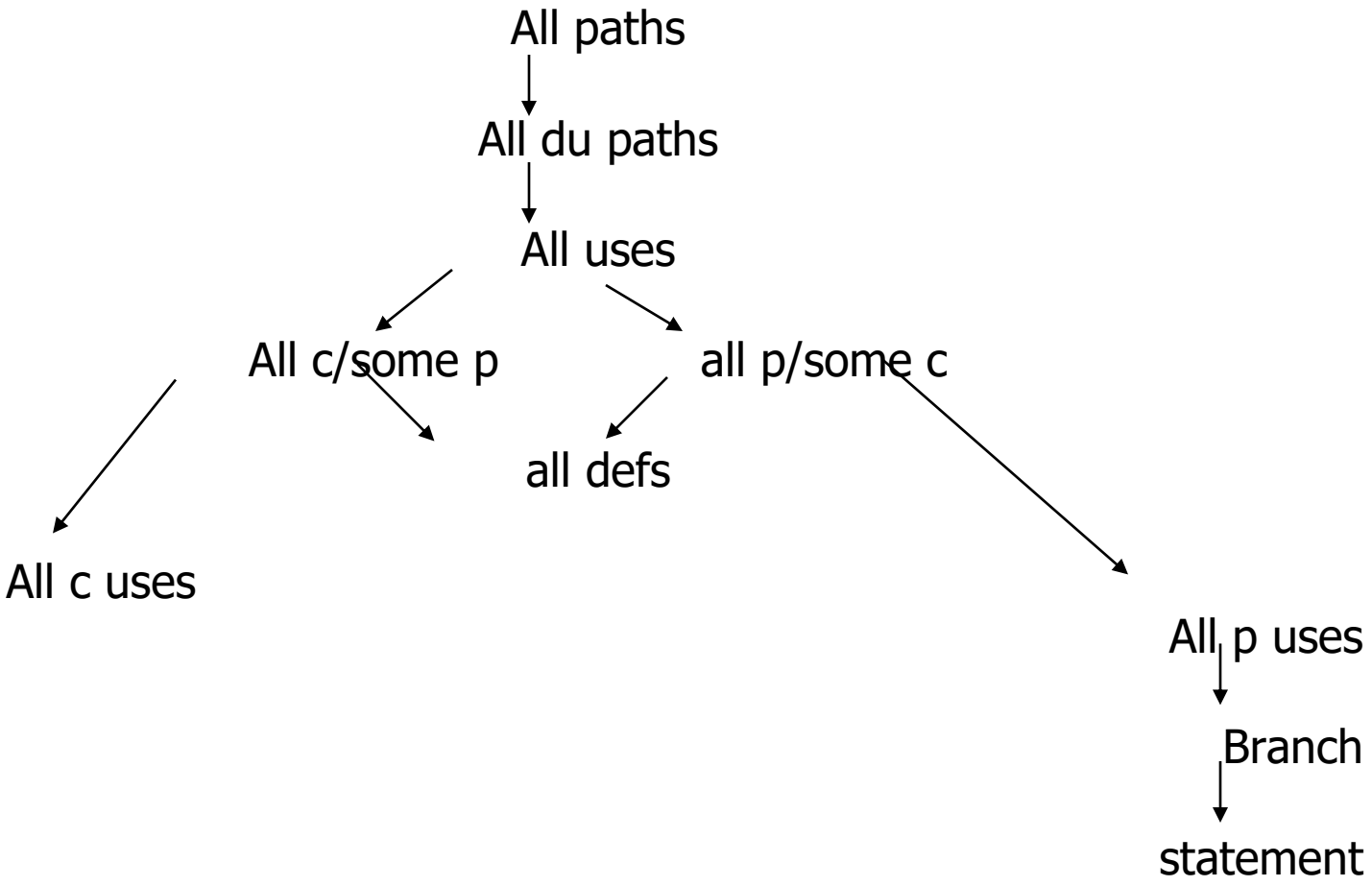
All Definitions Strategy

- The all definitions strategy asks only every definition of every variable be covered by at least one use of that variable, be that use a computational use or a predicate use.

All predicate-uses, All Computational uses Strategies

- The all predicate uses strategy is derived from APU+C strategy by dropping the requirement that we include a c-use for the variable if there are no p-uses for the variable.
- The all computational uses strategy is derived from ACU+P strategy by dropping the requirement that we include a p-use for the variable if there are no c-uses for the variable.

Ordering the Strategies



Slicing and Dicing

- A program slice is a part of a program defined with respect to a given variable X and a statement i: it is the set of all statements that could affect the value of X at statement i- where the influence of a faulty statement could result from an improper computational use or predicate use of some other variable at prior statements. If X is incorrect at statement i, it follows that the bug must be in the program slice for X with respect to i;
- A program dice is a part of a slice in which all statements which are known to be correct have been removed.