04-Notation

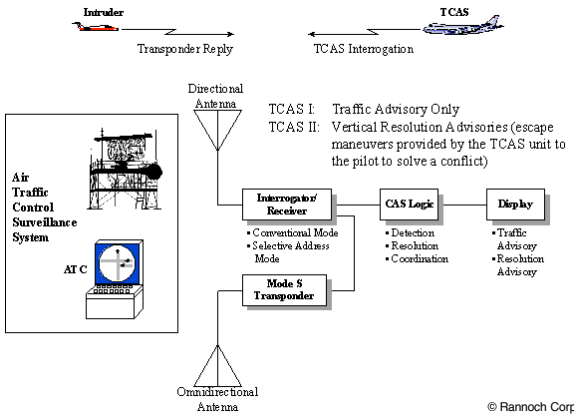
- Readings:
  - **Fundamentals of Software Engineering**, Ch. 5
  - [[rW98] Roel Wieringa, "A survey of structured and object-oriented software specification methods and techniques," **ACM Computing Surveys** December 1998

How to write it down?

- natural language
- structured natural language
- pictorial notation
  - Box-and-Arrow Charts
  - Graphs
    - Flowgraphs
    - Parse Trees
    - Call graphs
    - Dataflow graphs
  - Charts, Diagrams
- data models
- formal language(s)
  - state-oriented
  - function-oriented
  - object-oriented

Which of these are best adapted to providing which types of answers to which types of stakeholders?

TCAS Overview



Levison "Intentional" Spec

High-Level Functional Requirements

[1.18]
TCAS shall provide collision avoidance protection for any two aircraft closing horizontally at any rate up to 1200 knots and vertically up to 10,000 feet per minute.

Assumption: This requirement is derived from the assumption that commercial aircraft can operate up to 600 knots and 5000 fpm during vertical climb or controlled descent (and therefore two planes can close horizontally up to 1200 knots and vertically up to 10,000 fpm).

[1.19]
TCAS shall handle encounters involving multiple aircraft in areas with large numbers of aircraft within a selected range (without saturation of the operating frequencies).

[1.19.1]
TCAS shall operate in en-route and terminal areas with traffic densities up to 0.3 aircraft per square nautical miles (nmi) (i.e., 24 aircraft within 5 nmi) (↓2.13, ↓Page 202).

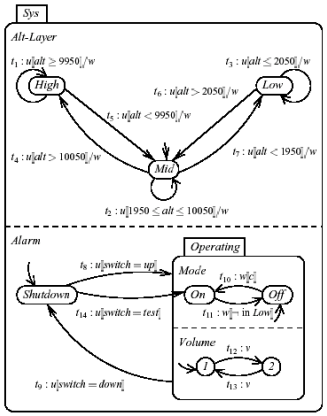
Assumption: Traffic density may increase to this level by 1990, and this will be the maximum density over the next 20 years.

[1.19.2]
TCAS shall operate out to 14 nautical miles (↓Page 188).

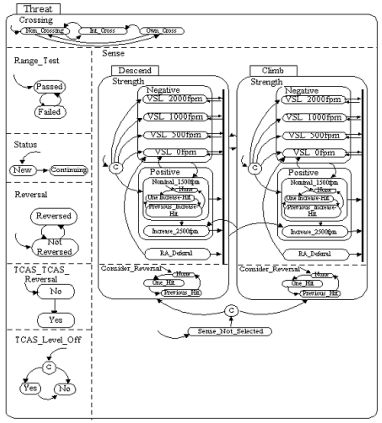
TCAS Truth Tables for Threat_Range_Test

AND	Other_Tracked_Range_Rate$_{t-540}$ > 10 ft/s (RDTHR)	T	T	F	F		
	Other_Tracked_Range$_{t-539}$ > DMOD	F	.	.	.		
	Modified_Tau_Capped$_{t-528}$ < TRTHR	.	.	T	T		
	Other_Tracked_Range$_{t-539}$ ≤ 12.0 nmi (RMAX)	.	.	T	T		
	Other_Tracked_Range_Rate$_{t-540}$ * Other_Tracked_Range$_{t-539}$ > H1	F	.	.	.		
	Nuisance_Alarm_Filter	F	.	F	F		
	Filter_Status$_{s-300}$ **in state** Dont_Filter_RA	T	T	T	T		
	Intruder_Status$_{s-194}$ **in state** Threat	.	T	T	.		
	Range_Track_Firmness$_{s-277}$ **in one of** {3, 4, 5, 6, 7, 8}	.	.	.	T		
	Range_Trackers$_{s-287}$ **in state** Not_Initialized	.	T	.	.		

RMSL



TCAS Statechart



SMV program

```
MODULE main
VAR
        u: boolean;
        v: boolean;
        w: boolean;
        switch: {up, down, testg};
        alt: 0..20000;
        prev-alt: 0..20000;
        Alt-Layer: {High, Mid, Lowg;
...
DEFINE
        stable := !(u|v|w);
        in-Sys := 1;
        in-Alt-Layer := in-Sys;
        in-High := in-Alt-Layer & Alt-Layer = High;
        in-Mid := in-Alt-Layer & Alt-Layer = Mid;
        in-Low := in-Alt-Layer & Alt-Layer = Low;
        in-Alarm := in-Sys;
        in-Shutdown := in-Alarm & Alarm = Shutdown;
...

ASSIGN
init(Alt-Layer) := Mid;
next(Alt-Layer) :=
        case
                t1h4 : High;
                t2t5t6: Mid;
                t3t7 : Low;
                1 : Alt-Layer;
        esac;
init(Alarm) := Shutdown;
next(Alarm) :=
case
        t8t14: Operating;
        t9 : Shutdown;
        1 : Alarm;
esac;
...
```

Best laid plans …

- FAA
  - "CAASD personnel have conducted safety studies to evaluate the performance of each successive version of the TCAS logic .."
  - "In a 1997 report on version 7, CAASD's Dr. Michael McLaughlin examined the reduced risk of collision in aircraft equipped with TCAS II versus the risk in aircraft without TCAS … and concluded that
    - "TCAS should reduce NMAC probability by at least 90 to 98 percent," depending on whether one or both aircraft in an encounter are equipped with TCAS."

… go oft astray

- The investigation into the chain of events behind mid-air collision over southern Germany has increasingly focused on the Swiss air traffic control agency Skyguide. Intially Skyguide blamed the Russian crew of one of the two aircraft for ignoring warnings to dive. But since then new important information has come to light: The pilot of the Russian Tu-154 was given conflicting instructions by air traffic control and his onboard computer The Russian pilot was given only 44 seconds warning A warning system at the control centre was switched off for maintenance Only one controller was on duty at the time The centre's radar system does not meet EU standards … BBC

Natural Language

- Write in "plain English"
  - All stakeholders understand natural language (?)
  - Possible to augment with defined terms
  - Use of punctuation for clarification
  - Text/word processing systems help automate/maintain/alter
- Examples of Natural Language artifacts:
  - User manuals
  - Requirements specifications
  - Test Plans
  - Development status reports

Natural language

- Inherently ambiguous and also complex
- From one of Michael Jackson's books:
  - In an airport at the foot of an escalator are two signs
    - "Shoes must be worn."
    - "Dogs must be carried."

What does this mean?

- In logic it 's clear
  - $\forall x\ (OnEscalator(x) \Rightarrow \exists y(PairOfShoes(y) \wedge IsWearing(x,y))$
  - $\forall x\ ((OnEscalator(x) \wedge IsDog(x)) \Rightarrow IsCarried(x)$
- Or is it?
  - Do dogs have to wear shoes?
    - Is this a question of the types of x and y?
  - What are "shoes"? What are "dogs"? What does it mean to "wear shoes"?
  - Why do the formalizations say "dogs are carried" and "shoes are worn" while the signs say "must be"?

Mood

- The formalizations are in the ***indicative*** mood: statements of fact
- The signs are in the ***optative*** mood: statements of desire
- This kind of "mood mixing" increases confusion

in·dic·a·tive

1:of, relating to, or constituting a verb form or set of verb forms that represents the denoted act or state as an objective fact

op·ta·tive

1 a :of, relating to, or constituting a verbal mood that is expressive of wish or desire

Meaning of terms

- "dog" (noun)
  - OED has 15 definitions
  - 11K words in the full definition
- "shoe" (noun)
  - Webster's has six definitions including
    - covering for the human foot
    - a device that retards, stops, or controls the motion of an object
    - a device (as a clip or track) on a camera that permits attachment of accessory items
    - a dealing box designed to hold several decks of playing cards

Optative vs. indicative mood

- Indicative: describes how things in the world are regardless of the behavior of the system
  - "Each seat is located in one and only one theater."
- Optative: describes what you want the system to achieve
  - "Better seats should be allocated before worse seats at the same price."
- Principle of uniform mood
  - Indicative and optative properties should be entirely separated in a document
  - Reduces confusion of both the authors and the readers
  - Increases chances of finding problems
  - If the software works right, both sets of properties will hold as facts

Mood mixing: example

- The lift never goes from the n^{th} to the $n+2^{nd}$ floor without passing the $n+1^{st}$ floor.
- The lift never passes a floor for which the floor selection light inside the lift is illuminated without stopping at that floor.
- If the motor polarity is set to up and the motor switch setting is changed from off to on, the lift starts to rise within 250 msecs.
- If the upwards arrow indicator at a floor is not illuminated when the lift stops at the floor, it will not leave in the upwards direction.
- The doors are never open at a floor unless the lift is stationary at that floor.
- When the lift arrives at a floor, the lift-present sensor at the floor is set to on.
- If an up call button at a floor is pressed when the corresponding light is off, the light comes on and remains on until the call is serviced by the lift stopping at that floor and leaving in the upwards direction.

Natural Language

- Advantages
  - Easy to train users
  - Clarity is possible (but may be difficult)
  - Completeness is possible (but by no mean assured)
  - Easily modified
  - It is the “least common denominator”
- Disadvantages
  - Determining consistency between natural language artifacts and anything else is hard/subjective
  - Ambiguity in natural language is easy and often intentional
  - Clear natural language expression is very difficult
  - The longer the text, the more information, the more the risk of inconsistency, the harder it is to determine
  - No way of knowing when a specification is "complete"

Natural Language Summary

- Cannot reason definitively about natural language
- Cannot be sure that natural language artifacts are consistent with other artifacts
- Assurances to stakeholders are shaky

How to write it down?

- natural language
- **structured natural language**
- pictorial notation
  - Charts, Diagrams, Box-and-Arrow Charts
  - Graphs
    - Flowgraphs
    - Parse Trees
    - Call graphs
    - Dataflow graphs
- formal language(s)
  - state-oriented
  - function-oriented
  - object-oriented

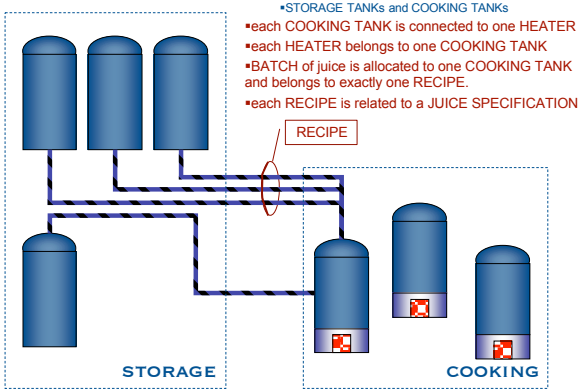
Structured "Natural" Language

- Disciplined Use of Natural Language
- Response to natural language problems of:
  - Imprecision
  - Ambiguity
  - Consistency (especially when due to size)
  - Inability to reason effectively and definitively
- Familiar approaches:
  - Restricted use of reserved terms
  - Structuring (paragraph numbering, outline form, templates, etc.)
- Other, earlier examples of disciplined use of natural language:
  - Legal documents
  - Recipes
  - Help systems

Declarative vs. Imperative

- Declarative specification [indicative, descriptive]
  - Pre and post condition pairs, where
    - a precondition is a condition on the input and system state at the start of executing the function and the postcondition is a condition on the output and the system state after the execution of the function.
  - Implementation independent, but under specifies
- Imperative specification [optative?, prescriptive, operational]
  - describe the activities to be performed to get from the input and initial system state to the output and resulting system state.
  - Leads to executable specification, but over specifies by giving an implementation

Juice Plant Example



- two kinds of TANKs
  - STORAGE TANKs and COOKING TANKs
- each COOKING TANK is connected to one HEATER
- each HEATER belongs to one COOKING TANK
- BATCH of juice is allocated to one COOKING TANK and belongs to exactly one RECIPE.
- each RECIPE is related to a JUICE SPECIFICATION

RECIPE

Declarative

Event flow input: start heating
Data flow input:batch ID
Data store input:ALLOCATION OF BATCH TO COOKING TANK
HEATER OF COOKING TANK
RECIPE OF BATCH
Event flow output: start controlling
Data store output:TEMPERATURE RAMP DATA

Precondition 1:
batch ID occurs exactly once in ALLOCATION OF BATCH TO COOKING TANK
and allocation of batch is cooking tank ID
and recipe for batch ID occurs in RECIPE OF BATCH with end time and end temperature
Postcondition 1:
new(ramp ID) + batch ID + cooking tank ID + heater ID + end time + end temperature
exists in TEMPERATURE RAMP DATA

Precondition 2:
batch ID does not occur exactly once in ALLOCATION OF BATCH TO COOKING TANK
Postcondition 2: error

Precondition 3:
recipe for batch ID does not occur in RECIPE OF BATCH temperature
Postcondition 3: error

But what about state of other data?

Figure 17. A declarative specification.

ACM Computing Surveys, Vol. 30, No. 4, December 1998

Imperative

```
Event flow input: start heating
Data flow input:batch ID
Data store input:ALLOCATION OF BATCH TO COOKING TANK
HEATER OF COOKING TANK
RECIPE OF BATCH
Event flow output: start controlling
Data store output:TEMPERATURE RAMP DATA
If batch ID occurs exactly once in ALLOCATION OF BATCH TO COOKING TANK
then get cooking tank id from ALLOCATION OF BATCH TO COOKING TANK;
     get heater ID from HEATER OF COOKING TANK;
     if recipe for batch ID occurs in RECIPE OF BATCH
     then get end time and end temperature from RECIPE OF BATCH;
          create ramp ID;
          update TEMPERATURE RAMP DATA
          with ramp ID + batch ID + cooking tank ID + heater ID + end time + end temperature
     else error;
else error
```

Implementation specific?

Figure 18. An imperative specification.

PSL (Problem Statement Language)

```
DESCRIPTION:
        this process performs those actions needed to interpret
        time records to produce a pay statement for each hourly
        employee;
KEYWORDS:       independent;
ATTRIBUTES ARE:
        complexity-level
                high;
GENERATES       pay-statement, error-listing;
RECEIVES:       time-card;
SUBPARTS ARE:   hourly-paycheck-validation, hourly-emp-update,
                h-report-entry-generates, hourly-paycheck-production;
PART OF:        payroll-processing;
DERIVES:        pay-statement;
USING:          time-card, hourly-employee-record;
DERIVES:        hourly-employee-report;
USING:          time-card, hourly-employee-record;
DERIVES:        error-listing;
USING:          time-card, hourly-employee-record;
PROCEDURE:      read record, add up hours, multiply by pay rate…..
HAPPENS:        number-of-payments TIMES-PER pay-period;
TRIGGERED BY:   hourly-emp-processing-event;
TERMINATION-CAUSES:  new-employee-processing-event;
SECURITY IS:    company-only;
```


PSL (Problem Statement Language)

PROCEDURE:

1. compute gross pay from time card data
2. compute tax from gross pay
3. subtract tax from gross pay to obtain net pay
4. update hourly employee record
5. update department record accordingly
6. generate paycheck

Note: if status code indicates that employee did not work this pay period, no processing will be done for this employee

Discipline Mechanisms in PSL

- Use of keywords (defined elsewhere in specification)
  - fosters precision, clarity
  - helps support consistency determination: some
  - keyword fields have defined relations to others (eg. Input-to and output-from)
- Use of templates
  - facilitates determination of completeness
  - fosters clarity
  - facilitates consistency checking
- Use of structure:
  - HIERARCHY:
    - standard practice for dealing with size, complexity
    - exploits innate human capacity for abstraction
  - DATA FLOW:
  - CONTROL FLOW:

Structured Natural Language

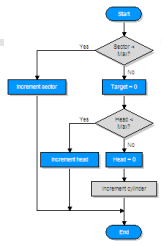
- big step in the right direction
  - improvement over unstructured natural language
- possible to determine some kinds of consistency thru:
  - mechanisms for reducing ambiguity
  - mechanisms for fostering completeness
  - structuring mechanisms for dealing with complexity
- **but**
  - stilted form reduces clarity: less suitable for some key stakeholder groups
  - some residual reliance on natural language means ambiguity remains
  - size is still a problem: PSL specs (for example) can be huge: consistency determination is long/error prone

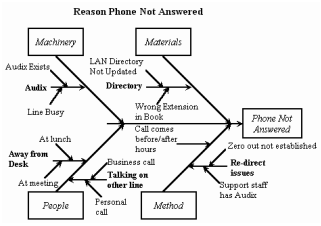
How to write it down?

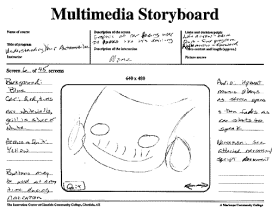
- natural language
- structured natural language
- **pictorial notation**
  - Charts, Diagrams, Box-and-Arrow Charts
  - Graphs
    - Flowgraphs
    - Parse Trees
    - Call graphs
    - Dataflow graphs
- formal language(s)
  - state-oriented
  - function-oriented
  - object-oriented

Various charts

- Flowcharts
- Storyboards
- Cause and Effect Diagram
- Pareto Chart
- Histogram



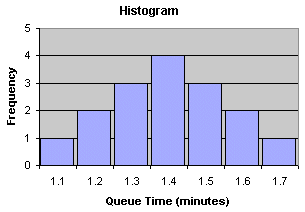


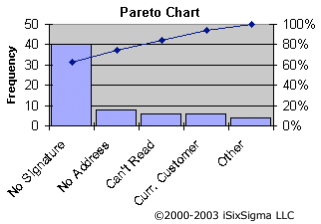


Various charts

- Flowcharts
- Storyboards
- Cause and Effect Diagram
- Pareto Chart
- Histogram … and more

80/20 Rule
- 80% of process defects arise from 20% of the process issues.
- 80% of delays in schedule arise from 20% of the possible causes of the delays.
- 80% of customer complaints arise from 20% of your products or services.





Pictorial and Diagrammatic Approaches

- Diagrams composed of visual elements
  - rigorously defined (definable?) semantics
  - used as modeling devices
  - depict key structural aspects of system
- Benefits
  - greatly improve clarity
  - greatly improve clarity consistency
  - facilitate completeness of notation
  - reduce ambiguity
- but
  - reduce modifiability, perhaps significantly
  - restrictions in semantics impede completeness
  - more on these issues later.....

How to write it down?

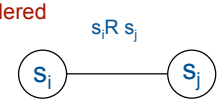
- natural language
- structured natural language
- pictorial notation
  - Charts, Diagrams
  - **Graphs**
    - Flowgraphs
    - Parse Trees
    - Call graphs
    - Dataflow graphs
- formal language(s)
  - state-oriented
  - function-oriented
  - object-oriented

Graphs

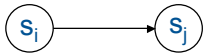
- A graph, G = (N, E), is an ordered pair consisting of a node set, N, and an edge set, E = $\{(n_i, n_j)\}$
  - If the pairs in E are ordered, then G is called a directed graph, and is depicted with arrowheads on its edges
  - If not, the graph is called an undirected graph
- Relations:
  - A ***relation***, R, over a set, S = $\{s_i\}$ is a set of ordered n-tuples R = $\{r_j\}$, where $r_i = (s_{i,1}, s_{i,2}, \ldots, s_{i,n})$
  - A ***binary relation*** is a relation where all the tuples are 2-tuples
    If (s_i, s_j) is an element of R, then we often write $s_i R s_j$
- Another view of relations:
  - The relation, R, over the set S can be defined as:
    $R = \{ (s_i, .., s_j) \mid PRED(s_i, ..., s_j) = True$, for some predicate, PRED$\}$
  - If the tuples are ordered, the relation is called an ***ordered relation***
  - If the tuples, $<t_{i,1}, t_{i,2}, ... t_{i,n}>$ are unordered, the relation is an ***unordered relation***

Relations & graphs

- Binary relations ($s_i R s_j$) can be represented as a graph
  - unordered



  - ordered

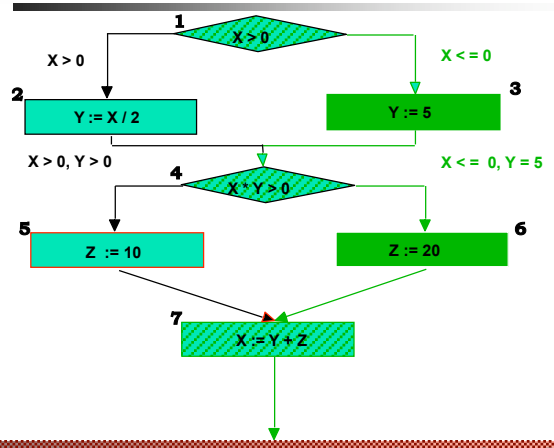


- General relations can be represented as multigraphs, hypergraphs

Flowgraphs & relations

- Examples
  - Let I = {all integers}, define Q= { (x,y,z) | x, y, z are integers and y = x**2 , z = x**3 }
  - Let S = {all states of the U.S., S_i}, define B = { (S_i, S_j) | S_i and S_j share a border}
  - Let L = {all statements L_i in a program, P}, define ImmFol = {(L_i, L_j) | the execution of L_j may immediately follow the execution of L_i for some execution of P}
- Flowgraphs
  - Let S = {all statements s_i in a program, P} and ImmFol
  - Then FG = (S, ImmFol) is called the **flowgraph** of P
    - FG is an ordered graph
    - Every execution sequence (ie. the sequence in which the statements of P are executed for a given execution of P) corresponds to a path in FG.
    - However, the converse is not true. A path through FG may not correspond to an execution sequence for P
    - A loop in P appears as a cycle in FG

Example with an infeasible path



Some Properties of Relations

- Some familiar properties of ordered binary relations, R, over the set S={s_k}:
  - Symmetry: s_i R s_j ==> s_j R s_i for all pairs, s_i and s_j in S
  - Reflexivity: s R s, for all s in S
  - Transitivity: s_i R s_j and s_j R s_k ==> s_i R s_k, for all s_i, s_j and s_k in S
  - A relation that is symmetric, reflexive and transitive is called an equivalence relation
  - If R = {(s_i, s_j)} is transitive, then C={(s_a, s_b)| there exists a sequence, i1, i2, ..., in, such that s_a=s_{i1} R s_{i2}, s_{i2} R s_{i3}, ... s_{in-1} R s_{in} = s_b } is called the transitive closure of R
  - Antisymmetry: s_i R s_j ==> ~(s_j R s_i) for all pairs, s_i and s_j in S
  - Irreflexivity: s ~R s for all s in S

Examples

- If S={all subroutines written in Fortran} s_1 R s_2 if and only if s_1 calls s_2, then R is an **irreflexive relation**
- Let PS ={c_e, all the statements in a program that consists of a set of modules, M={m_t} },
  INMOD = { (c_e, c_f) | c_e and c_f appear in the same module m_t }
  INMOD is an **equivalence relation**
- **Is the relation ImmFol transitive?**
- **What about**
  Fol = { (L1, L2) | the execution of L2 may follow the execution of L1 for some execution of P} ?

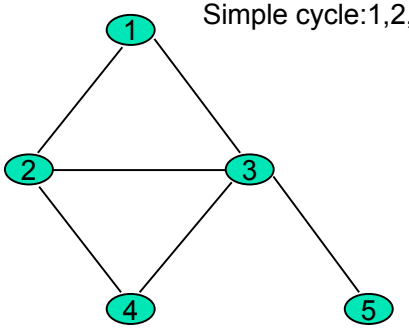
Paths

- A ***path,*** P, through an ***ordered graph*** G=(N ,E) is a sequence of edges, ($<n_{i,1}, n_{j,1}>$, $<n_{i,2}, n_{j,2}>$, ... , $<n_{i,t}, n_{j,t}>$) such that $n_{j,k-1} = n_{i,k}$ for all $2 \le k \le n$
- A **path**, UP, through an ***unordered graph*** UG=(N,U) is a sequence of edges, ($<n_{j,1}, n_{j,1}>$, $<n_{i,2}, n_{j,2}>$, ... , $<n_{i,t}, n_{j,t}>$) such that all of the $<n_{i,z}, n_{j,z}>$ can be ordered to assure that $n_{j,z-1} = n_{i,z}$ for all $2 \le k \le n$
  - In either case, $n_{i,1}$ is called the start node and $n_{j,t}$ is called the end node.
  - The length of a path is the number of edges in the path
- A **graph** G is **connected** if and only if, for every pair of nodes, n_1, n_2, there is path from one of them to the other with G considered to be an unordered graph.
- A ***path***, P, through a ***directed graph*** G = (N, E) is a sequence of edges, ($(n_{i,1}, n_{j,1})$, $(n_{i,2}, n_{j,2})$, ... , $(n_{i,t}, n_{j,t})$) such that $n_{j,k-1} = n_{i,k}$ for all $2 \le k \le t$
  - $n_{i,1}$ is called the start node and $n_{j,t}$ is called the end node
  - the length of a path is the number of edges in the path
  - paths are also frequently represented by a sequence of nodes ($n_{i,1}$, $n_{i,2}$, $n_{i,3}$, …, $n_{i,t}$)
- Cycles
  - a cycle in a graph G is a path whose start node and end node are the same
  - a simple cycle in a graph G is a cycle such that all of its nodes are different (except for the start and end nodes)
  - if a graph G has no path through it that is a cycle, then the graph is called acyclic

Examples

Cycle:1,3,2,4,3,1

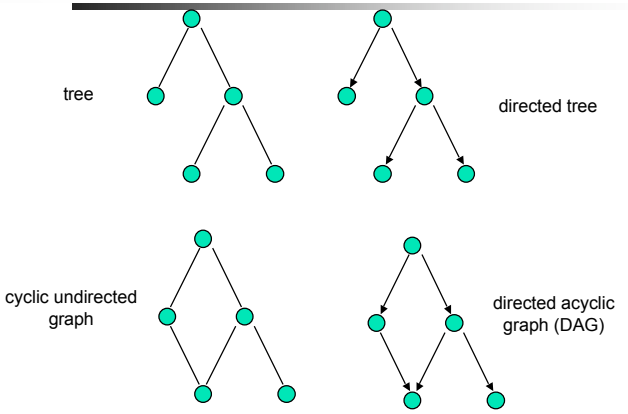
Simple cycle:1,2,3,1



Trees

- A **cycle** in a graph G is a path whose start node and end node are the same
- A **simple cycle** in a graph G is a cycle such that all of its nodes are different (except for the start and end nodes)
- If a graph G is connected and has no path through it that is a cycle, then the graph is called **acyclic**.
- An acyclic unordered graph is called a **tree**
  - If the unordered version of an ordered graph is acyclic, the graph is called a **directed tree**
  - A collection of trees is called a **forest**
- If the unordered version of an ordered graph has cycles, but the ordered graph itself has no cycles, then the graph is called a **Directed Acyclic Graph (DAG)**

Examples

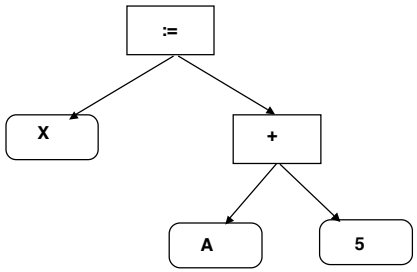


Abstract Syntax Tree (AST)

- a common form for representing expressions
  - executable statements are expressions
  - programs are expressions, where the operator is execute and the operands are the statements
- 2 kinds of nodes: operator and operands
  - operator applied to N operands
- An abstract syntax graph G = (N_1, N_2, E) where N_1 are nodes that represent operators in the language, N_2 are nodes that represent identifiers or literals , and E represents is "applied to"

Abstract Syntax Tree

X:= A + 5;



Abstract Syntax Trees

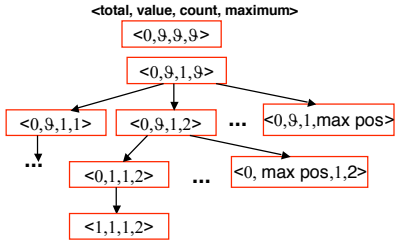
- have many advantages
  - provide a visual display of the body of an object
    - body of an assignment, addition, while, etc.
  - supports incremental modification
    - incremental syntactic or semantic analysis
  - basis for structural editing
    - user is provided with a template and fills in the slots
    - can assure syntactic consistency
    - need to control granularity of consistency checking
      - e.g., keystroke, semi-colon, user-request
  - used to create other graph models

computation tree

- models all the possible executions of a system
- at each node, shows the state (value) of each variable
- effectively infinite number of paths
- some paths may be effectively infinite

example computation tree

```
total, value, count, maximum : pos int;
total := 0;
count := 1;
read maximum;
while (count <= maximum) do
        read value;
        total := total + value;
        count: = count + 1;
endwhile;
print total;
```



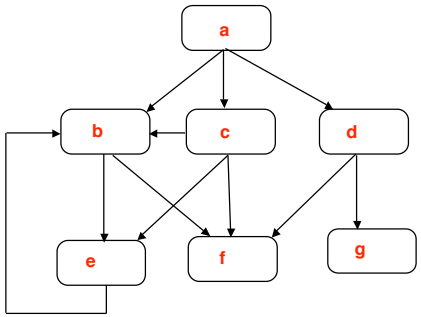
Computation Trees

- have advantages
  - represent the space that we want to reason about
  - for anything interesting they are too large to create or reason about
  - other models of executable behavior are providing **abstractions** of the computation tree model
    - abstract values
    - abstract flow of control
    - specialize abstraction depending on focus of analysis

Callgraphs

- Let PROC = {procedures S_i comprising a program P} and CALLS = {(S_i, S_j) | S_j is directly invoked from S_i during some execution of P}, then CG = (PROC, CALLS) is called the **Call Graph** of P
- CG is
  - a directed graph
  - does not represent the order entities are invoked
  - does not represent the number of times an entity is invoked
  - a cycle in g indicates that the nodes along the cycle syntactically participate in a recursive calling chain
  - if P is written in a language that does not allow recursion, then CG will be acyclic
  - provides a framework for inter-component analysis

Call Graph Example



Control Flow Graph (CFG)

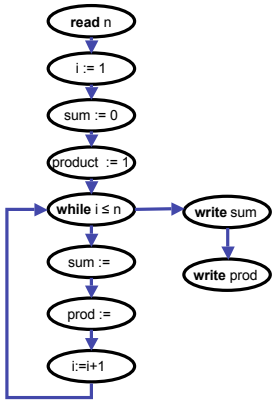
- represents the flow of executable behavior
- G = (N, E, S, T) where
  - the nodes N represent executable instructions (statement or statement fragments);
  - the edges E represent the potential transfer of control;
  - S is a designated start node;
  - T is a designated final node
  - E = { (ni, nj) | syntactically, the execution of nj follows the execution of ni}

Control Flow Graph (CFG)

- nodes may correspond to single statements, parts of statements, or several statements
- execution of a node means that the instructions associated with a node are executed in order from the first instruction to the last
- nodes are 1-in, 1-out

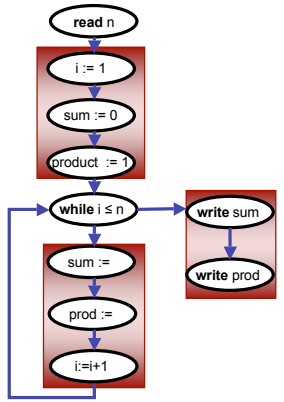
Control Flow Graph Model

```
read n;
i := 1;
sum := 0;
product  := 1;
while i ≤ n do
   sum := sum +1;
   product := product * i;
   i:= i+1;
endwhile;
write sum;
write product;
```



Reducing the CFG

- basic blocks are nodes that contain sequential execution
- Can reduce the number of nodes in the CFG, but may add more complications to the analysis



Benefits of CFG

- probably the most commonly used representation
  - numerous variants
- basis for inter-component analysis
  - collections of CFGs
- basis for various transformations
  - compiler optimizations
  - S/W analysis
- basis for automated analysis
  - graphical representations of interesting programs are too complex for direct human understanding

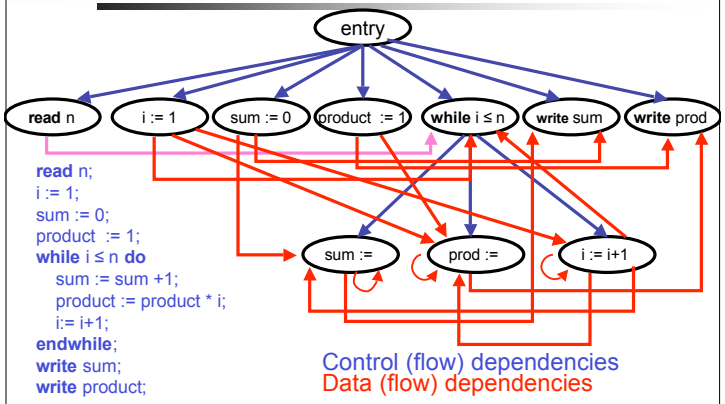
Some dataflow relations

- DataFlow(i, j) if node i creates data that node j uses
- Input(n) if n is a node that supplies initial input data
- Output(n) if n is a node that transmits data to end users
- EdgeAnnotation(e, text) if the string text describes the data that flows along edge e
- NodeAnnotation(n, text) if the string text describes the functioning of node n
- Questions this helps answer:
  - Why create this data? Who uses this data? What results does the end user see? What does the end user have to input?
  - Questions this can't answer: What is the exact sequence of events? How does a node do its job?

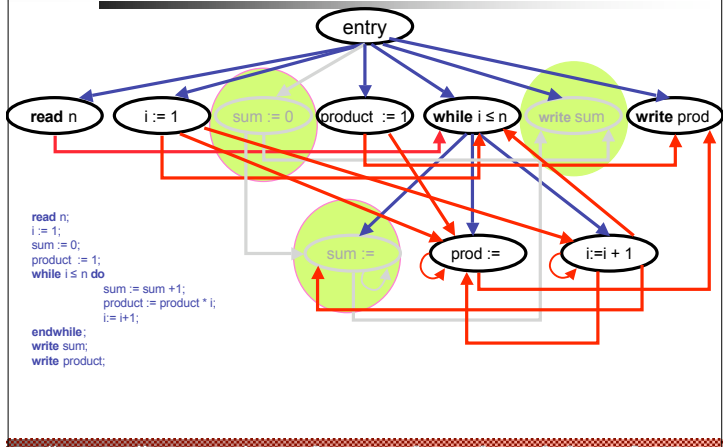
Program Dependence Graphs

- G is a directed graph, G = (V, E)
  - edges in E are of several types, representing control and data dependencies
  - vertices in V represent assignment statements and predicates and other special nodes
- Program Slice - Concept introduced by Mark Weiser in 1979
  - Argued it was a mental abstraction that programmers used when debugging
  - Program slice S is a reduced, executable program obtained from P by removing statements from P, such that S replicates part of the behavior of P
  - A slice includes all statements and predicates that might affect V at point p.
- How can we use the Program Dependency Graph to create slices?
  - A slice corresponds to all nodes that are reachable from a selected node (forward slice)

Example



Program Dependence Graph



Dataflow graphs & slices

- Uses
  - Data flow coverage criteria for selecting test cases
    - coverage criteria exercise subsets of control and data dependencies in the hope of exposing faults
  - debugging:
    - which statements could have caused an observed failure?
  - maintenance:
    - which statements will be affected by a change?
    - which statements could affect this statement?
  - dependence analysis
    - program dependencies provide a theory for restricting/focusing attention
- Problems
  - in practice, a program slice is often too big to be useful
  - infeasible paths lead to imprecision
  - complex data structures lead to imprecision