# Critical systems software testing and QA

Bogdan Wiszniewski

Dept. of Intelligent Interactive Systems
rm 423 EA,
ph. 58-347-1089
email: bogwiszn@pg.edu.pl

# Course organization

- Office hours:
  Thursdays, 15:15-17:00 hrs

- Practicals (labs & project):
  Dr. **Adam Łukasz Kaczmarek**,
  rm. EA422, adam.kaczmarek@eti.pg.edu.pl

# Course organization

- Objectives:

  1. Present software development standards in the European space industry and techniques of their implementation.
  2. Learn how to assess and manage critical system software quality in an IT project.
  3. Gain basic hands-on experience in bug tracking and reporting in a software project.

# Course content

1. Development of high-quality IT systems in a systematic way
2. ECSS standards: series „E” and „Q”
3. Software product life cycle vs. testing cycle.
4. Validation, verification and testing (VVT) processes in a software product lifecycle.
5. Planning of VVT processes.
6. Static analysis techniques of software systems.
7. Error, program and environment models in software testing.
8. Black-box (functional) software testing strategies.
9. White-box (structural) software testing strategies

# Literature

- IEEE Software and Systems Engineering Standards, http://standards.ieee.org/findstds/standard/software_and_systems_engineering.html
- Space engineering – Software, ECSS-E-ST-40C, 6 March 2009, European Cooperation for Space Standardization, ESA-ESTEC, http://ecss.nl/standards/ecss-standards-on-line/active-standards
- Space product assurance - Software product assurance, ECSS-Q-ST-80C Rev.1, 15 February 2017, European Cooperation for Space Standardization, ESA-ESTEC, http://ecss.nl/standards/ecss-standards-on-line/active-standards

# Grading

| Z# | Week | % | opis |
|---|---|---|---|
| L1 | 8 | 15 | Definition of tests for selected functionalities |
| L2 | 8 | 15 | Test execution and reporting |
| P | 15 | 30 | Documentation of the testing process according to the ESA standard (ECSS) |
| T | 8/15 | 40 | All weeks (lecture part) |
| | = | 100 | …of the final score |

# Pass/fail criteria

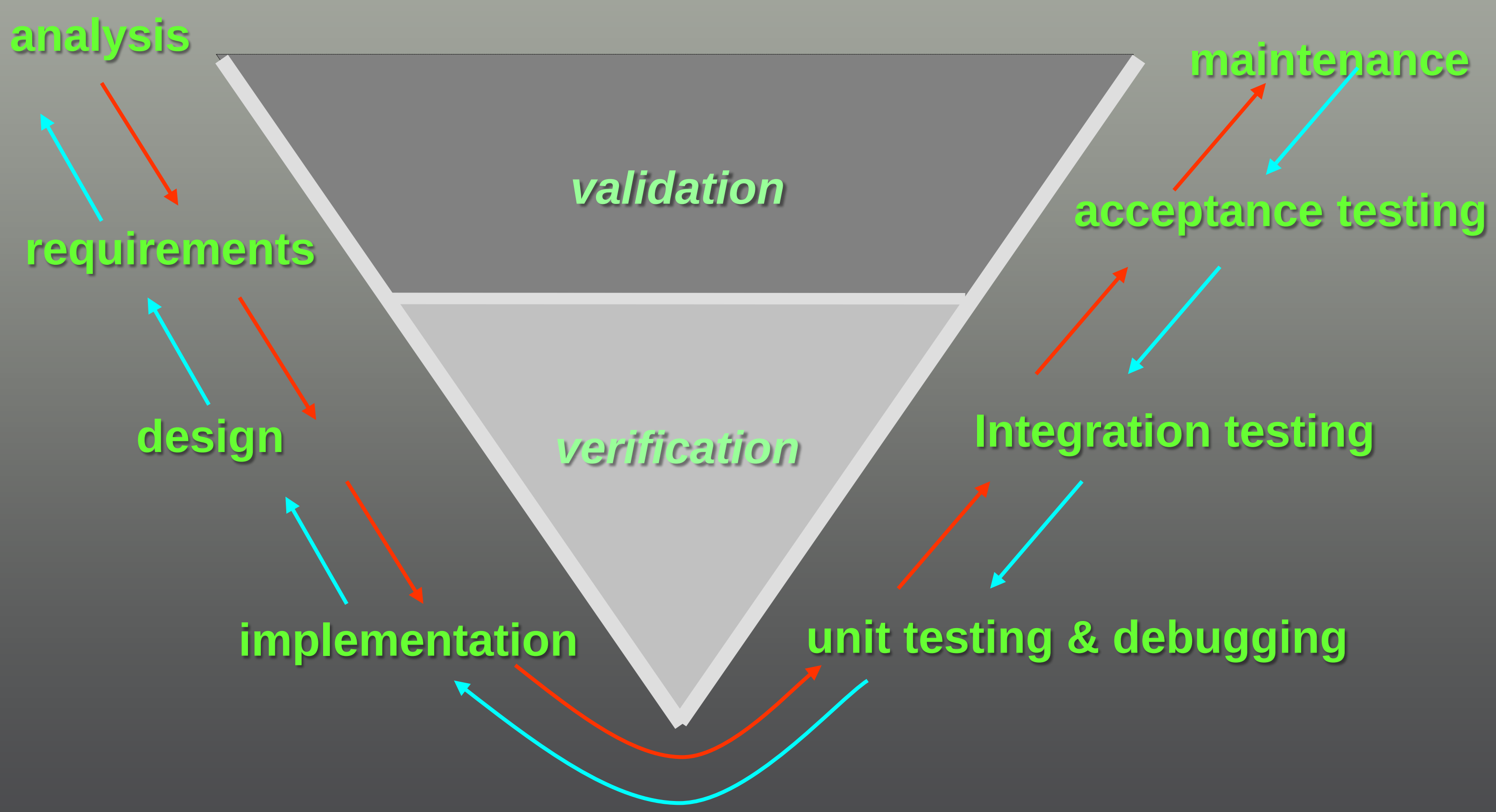
1. Total score of 50% minimum
2. Attending the final test (any non-zero score accepted)
3. All assignments must be submitted in the due time. No late assignments accepted, except of a valid medical excuse.

# Systematic approach

A. Product life-cycle vs. its testing cycle
B. VVT processes
C. Planning of VVT processes
D. Static analysis techniques
E. Error, program and runt-time environment models
F. Black-box (functional) testing
G. White-box (structural) testing

# A. Product life-cycle vs. its testing cycle

# Model “V”


analysis
requirements
design
implementation
validation
verification
maintenance
acceptance testing
Integration testing
unit testing & debugging

# Objectives

- Validation

  Assess whether the system (or its component) meets its requirements specification

→ ***Are we building the right product?***

# Objectives

- Verification

  Assess whether the product of a given phase meets the assumptions made at the beginning of this phase

  → ***Are we building the product right?***

# Objectives

- Testing

  Analysis of the system behavior (or its component) in order to measure (assess) its quality

  → ***How good is (or will be) the system?***

# Example

- Object:

→ ***A library routine for sorting matrices***

# Example

- Object:

  → ***A library routine for sorting matrices***

  – Testing

  → ***Does the object return a sorted matrix?***

# Example

- Object:

  → *A library routine for sorting matrices*

  - Testing

    → *Does the object return a sorted matrix?*

  - Verification

    → *Does the object sort matrices?*

# Example

- Object:

  → *A library routine for sorting matrices*

  - Testing

    → *Does the object return a sorted matrix?*

  - Verification

    → *Does the object sort matrices?*

  - Validation

    → *Can the procedure be included in the existing system library?*

# Systematic approach

# Systematic approach

bugs

# Systematic approach

run-time environment models

bugs

# Systematic approach

run-time environment models

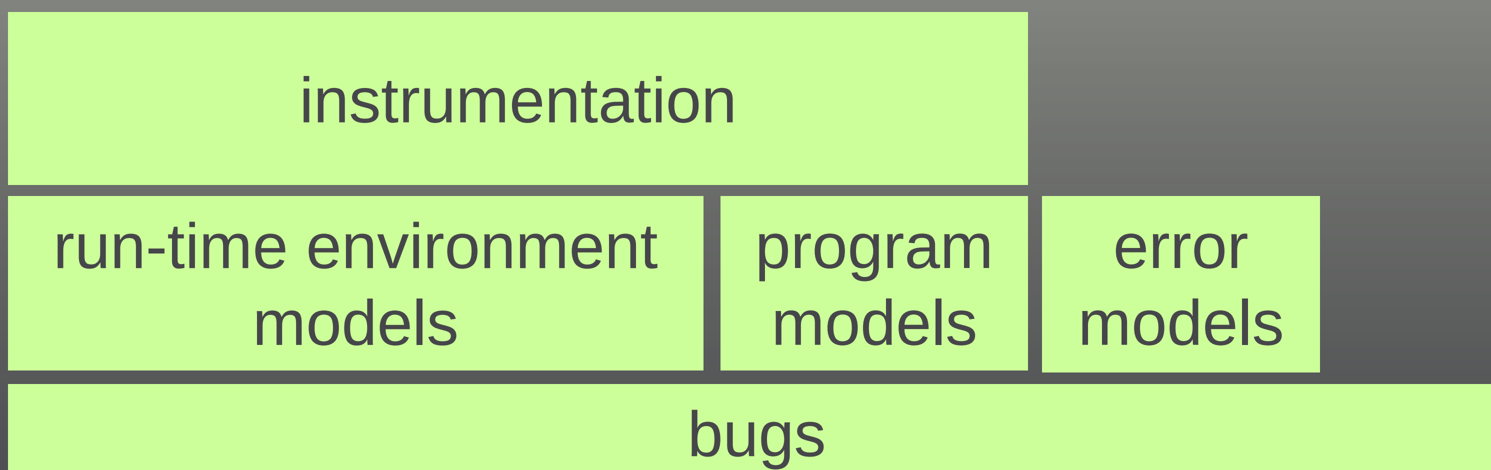
program models

bugs

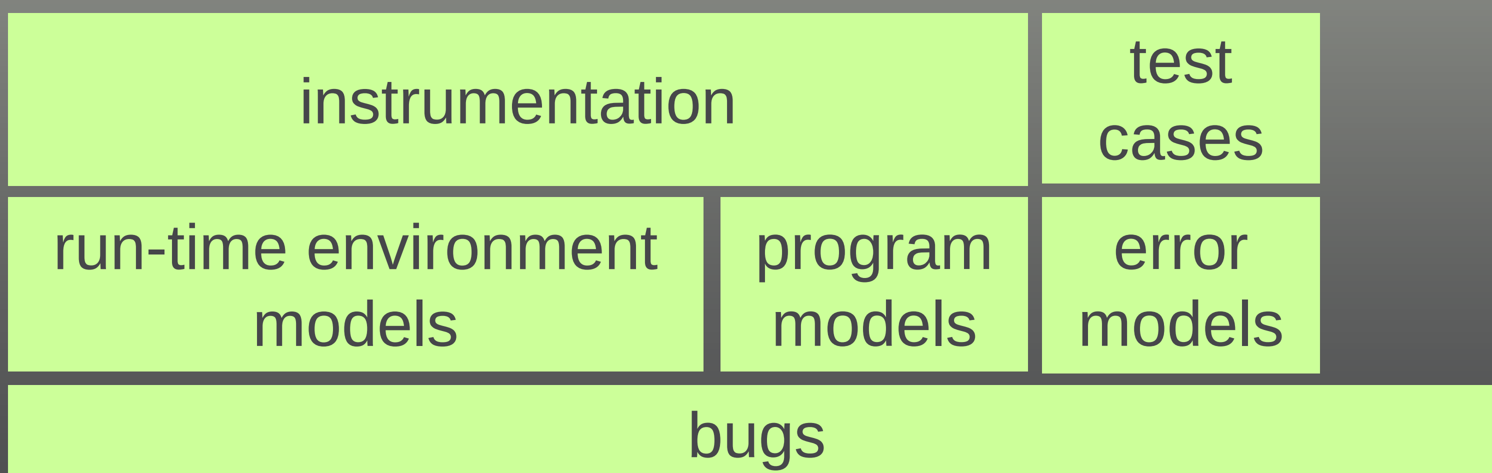
# Systematic approach

| run-time environment models | program models | error models |
| --- | --- | --- |
| bugs | | |

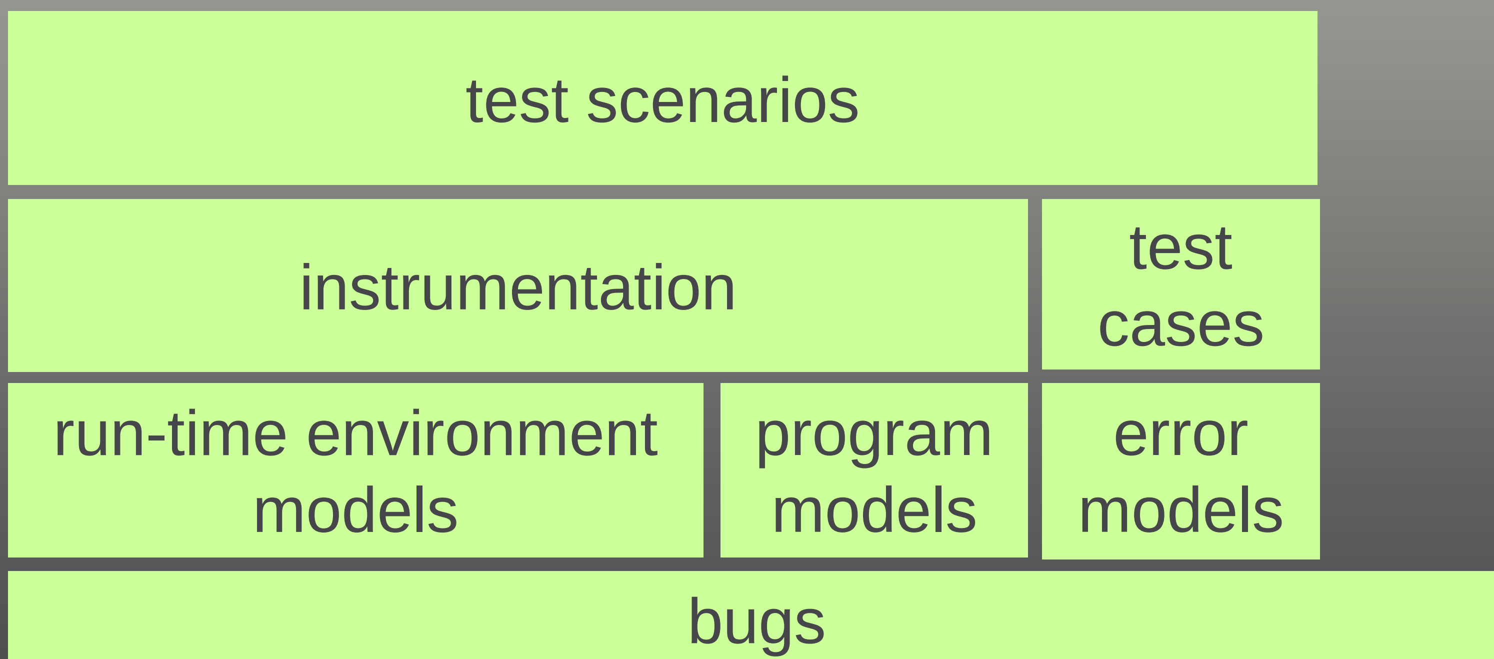
# Systematic approach



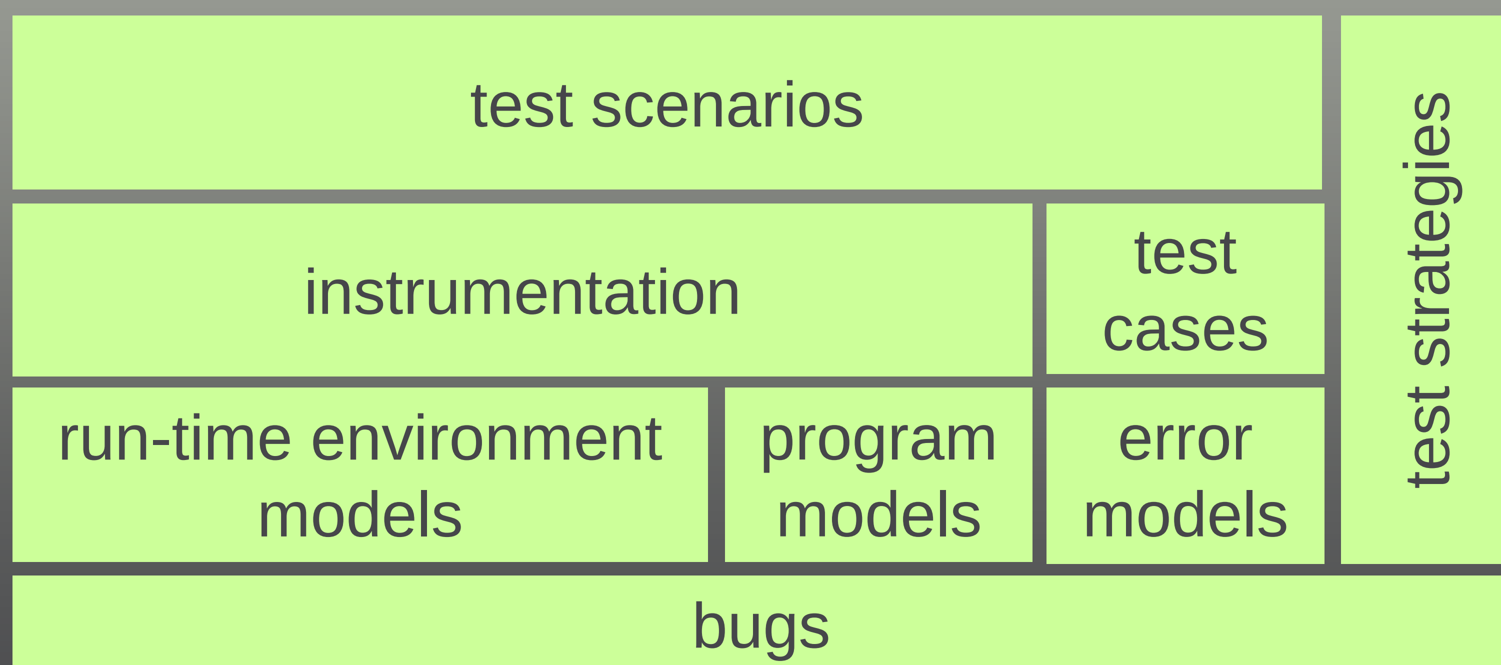
# Systematic approach



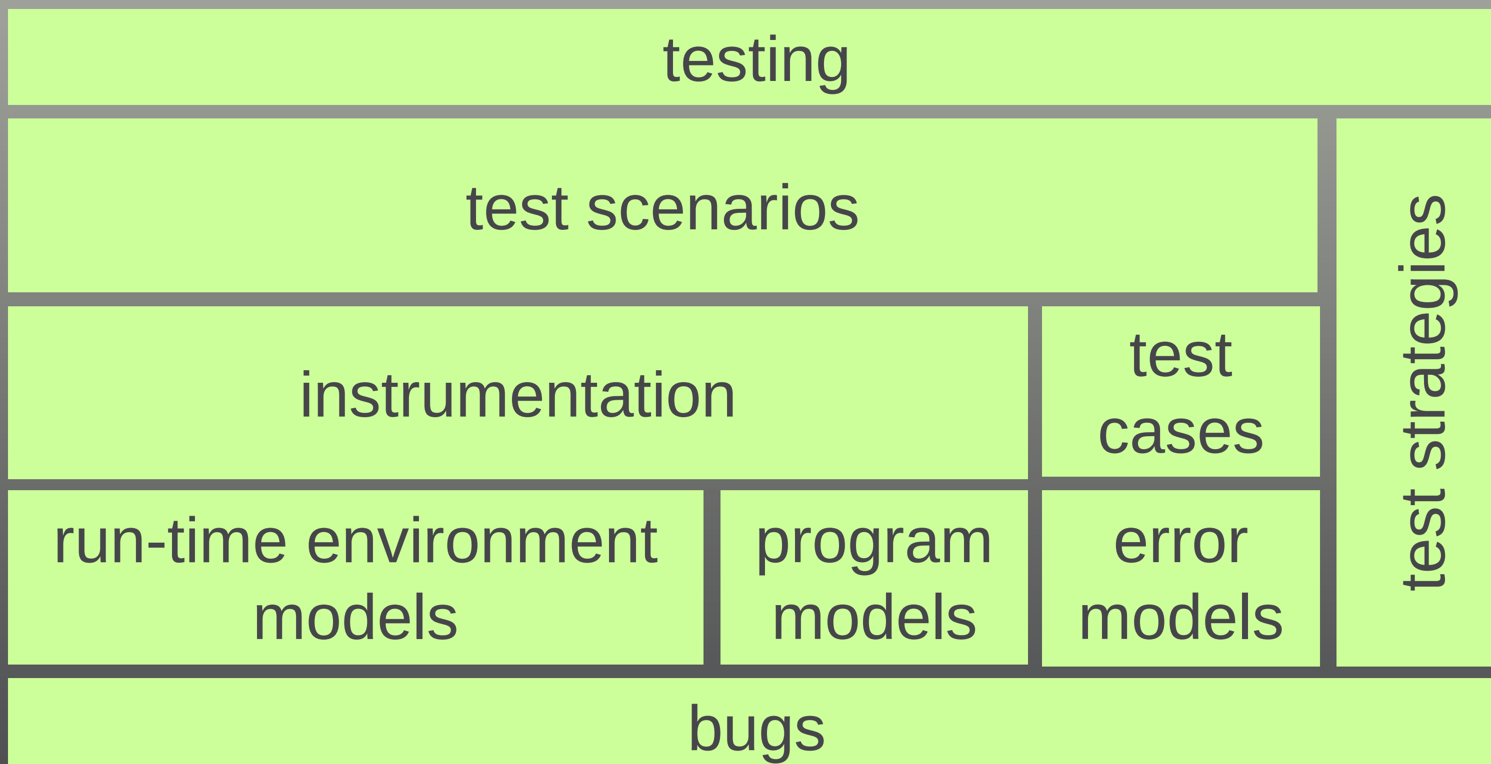
# Systematic approach



# Systematic approach



# Systematic approach



# Dynamic analysis

- Test case

*A single element selected from an enumerable set of program behaviors*

# Dynamic analysis

- Test completion criterion

  *A set of test cases defined based on the program behavior model*

# Dynamic analysis

- Testing strategy

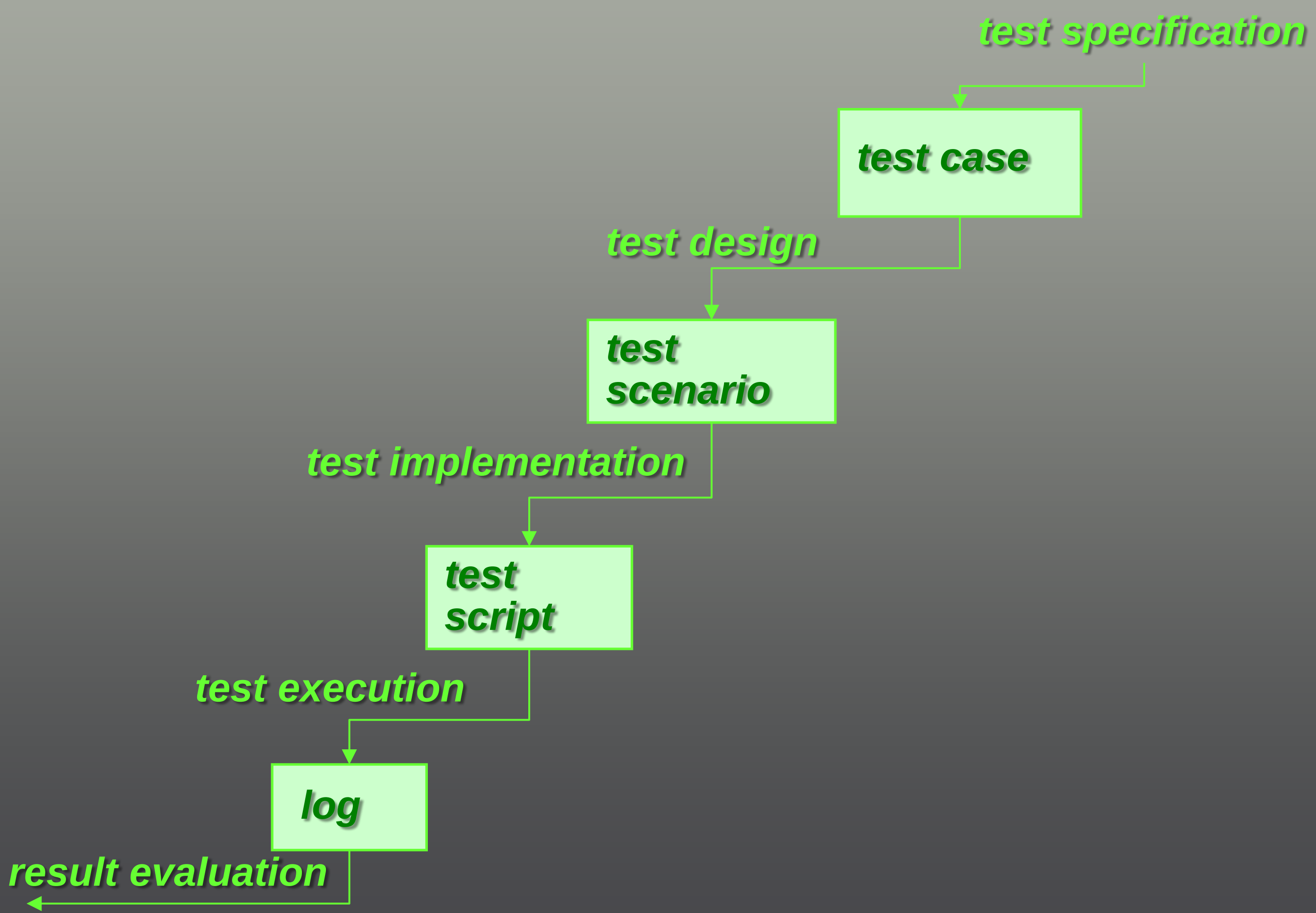
*A set of rules for selecting test cases to a (possibly finite) set according to some adopted criterion*

# Dynamic analysis

- Test scenario

  *Systematic observation of the expected behavior of an IT product conducted in a supervised mode*

# Test case ”life cycle”



# Testing levels

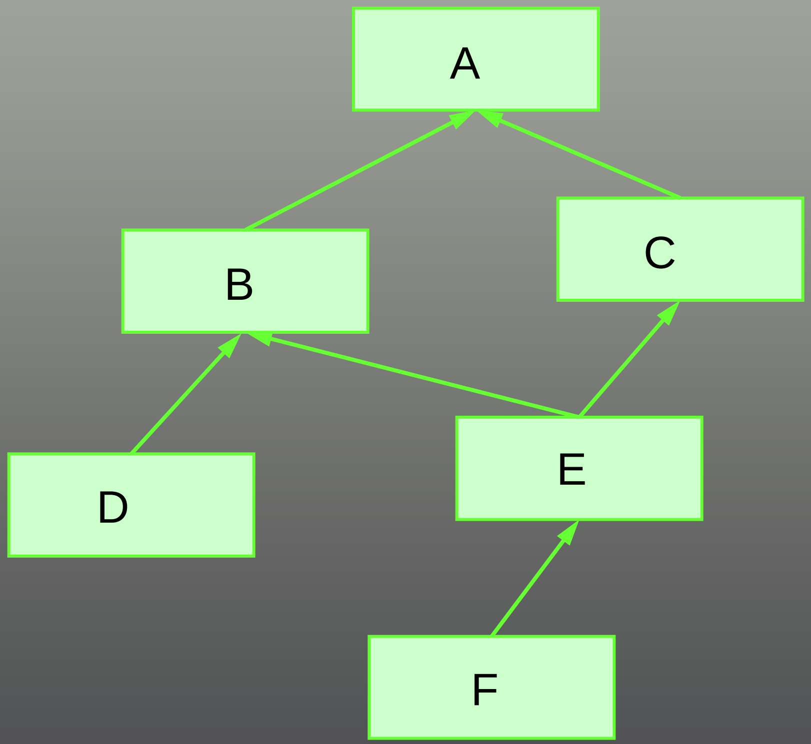
- Unit/module testing
- Integration testing
- System testing
- Acceptance testing
- Alfa/Beta-testing

# Unit testing

- Locating and removing errors
- Test completion
- Regression testing
- Test harness

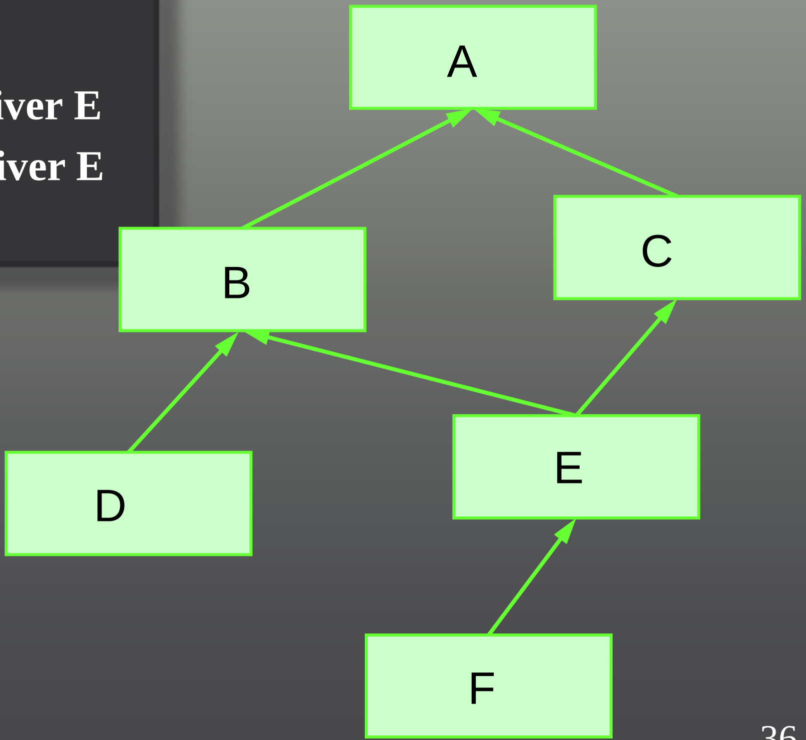
# Integration testing

- Strategies:
  - incremental
  - big-bang



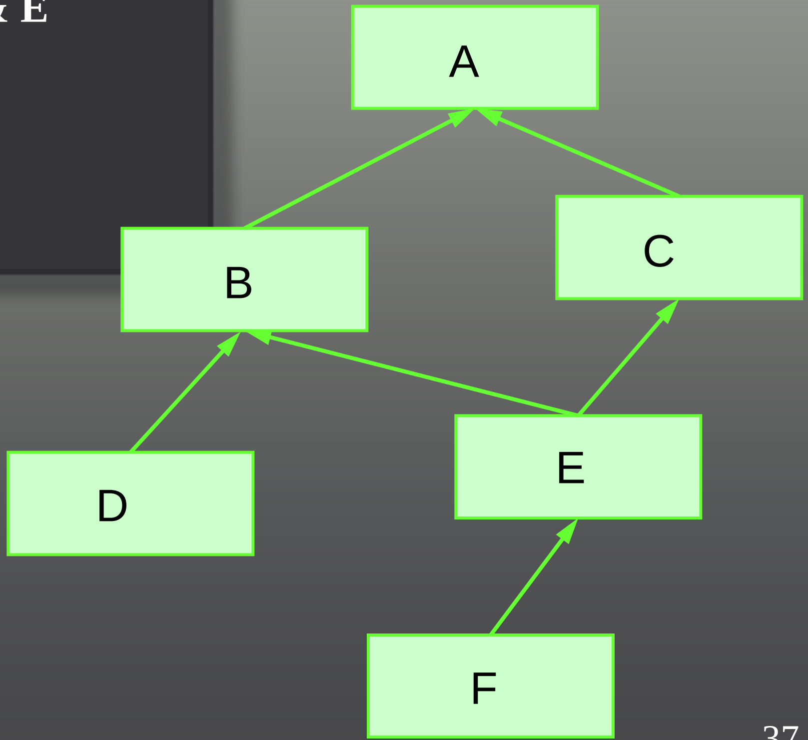
# Bottom-up method

| step | environment |
|---|---|
| 1 unit D | driver D |
| 2 unit F | driver F |
| 3 connected units E+F | driver E |
| 4 connected units D+E+F+B | driver B, driver E |
| 5 connected units C+E+F | driver C, driver E |
| 6 connected units A+B+C+D+E+F | -- |



# Top-down method

| step | environment |
|---|---|
| 1 unit A | stubs B & C |
| 2 connected units A+B | stubs C, D & E |
| 3 connected units A+B+C | stubs D & E |
| 4 connected units A+B+C+D | stub E |
| 5 connected units A+B+C+D+E | stub F |
| 6 connected units A+B+C+D+E+F | -- |



# System testing

| *category* | *features* | *systems* |
|---|---|---|
| functionality | every "what" the system does | dedicated systems |
| volume | voluminous input data | file/Big Data systems |
| stress | input data of high intensity | RT (control) systems |
| usability | user-friendliness | system HCI |
| security | break-in attempts | secure systems |
| performance | system dynamics measurements | RT (control) systems |
| storage | memory use | memory critical systems |
| configuration | optional system configurations | S/H upgrades |

# System testing

| *category* | *features* | *systems* |
|---|---|---|
| compatibility | older versions data | new releases |
| installability | installation procedures | complex installation |
| reliability | statistics (logs, incident | characteristics (MTTF, MTTR) |
| recovery | „destructive' data | fault tolerant systems |
| serviceability | maintenance procedures | administered systems |
| documentation | useful in testing? | administered systems |
| procedure | required personnel activities | command/decision systems |

# Acceptance testing

- Ownership rights transferred from the developer to the client
- Demonstration that all acceptance criteria have been met

→ **requirements specification**

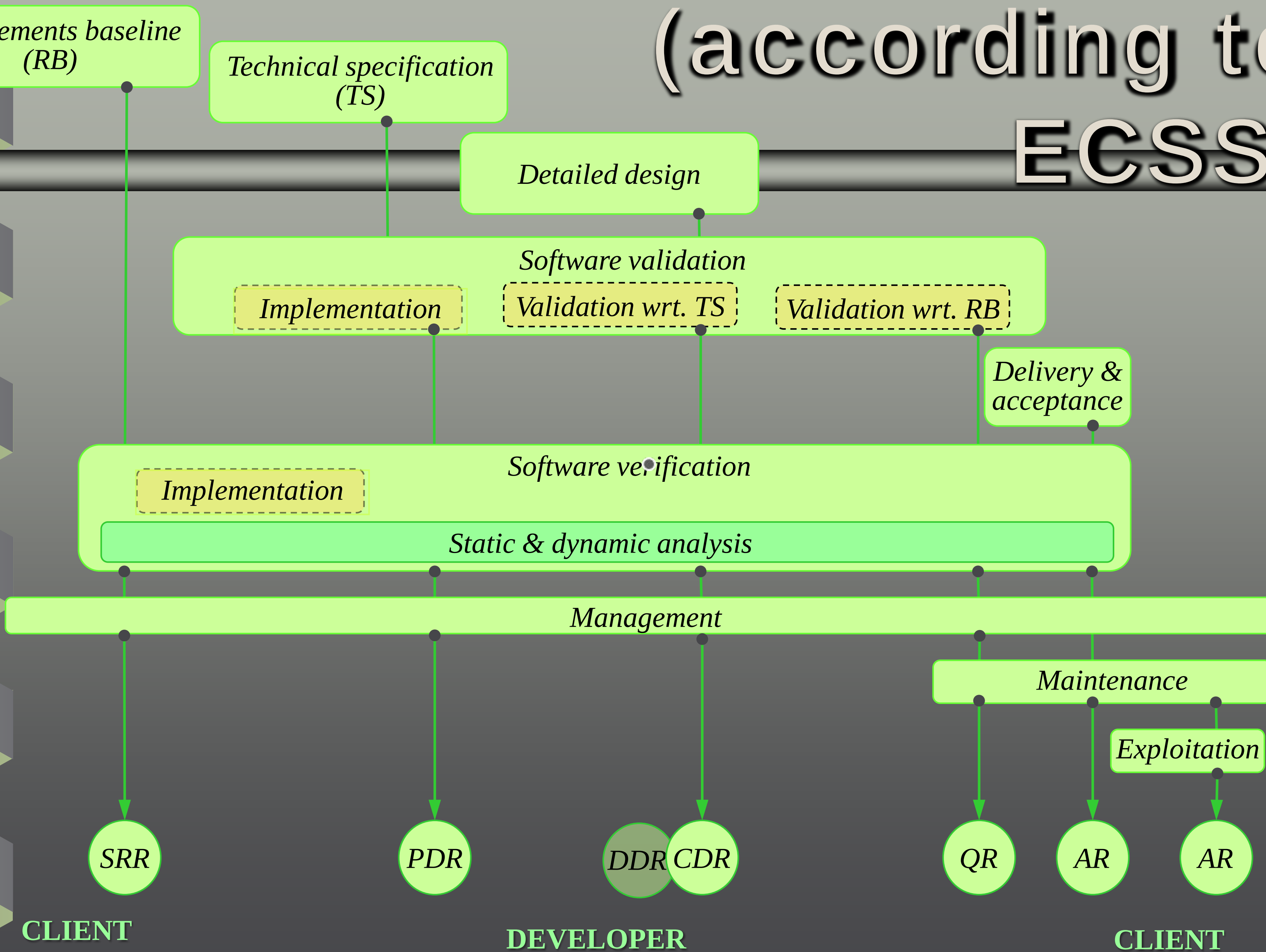
- Acceptance: phased, final
- “α-testing”: **customers test the product at the developer's facility (laboratory).**
- “β-testing”: **customers test the product at their own facility (laboratory)**

# B. VVT processes: life-cycle validation, verification and testing

# Responsibility, time schedule

- VVT processes:
  - early error detection
  - continuity of the development activities
  - better understanding of the product
  - decision criteria for the life-cycle phases

# Life-cycle phases (according to ECSS)



# Life-cycle processes (ECSS)

- Requirements baseline (RB)
  - functional and performance requirements for the planned software provided by the client
- Technical specification (TS)
  - a formal specification (logical model) of what the software is supposed to do,
  - physical design of the software structure mapping individual functions of the logical model to its components,
  - definition of control and data flows between them, the first part of the answer to the question of how the software is supposed to do something).

# Life-cycle processes (ECSS)

- Detailed design
  - design of algorithms and data structures (physical model), the second part of the answer to the question of how the software is supposed to do something,
  - justification of all design decisions,
  - implementation of units (code writing/generation),
  - testing of code units to demonstrate their compliance with requirements.

- Software validation
  - Demonstrating that the system meets all assumed quality goals → **quality attributes**

# Life-cycle processes (ECSS)

- Detailed design
  - design of algorithms and data structures (physical model), the second part of the answer to the question of how the software is supposed to do something,
  - justification of all design decisions,
  - implementation of units (code writing/generation),
  - testing of code units to demonstrate their compliance with requirements.

- Software validation
  - Demonstrating that the system m uality goals → **quality attributes**

- Functionality
- Performance
- Dependability
- Security
- Usability

# Life-cycle processes (ECSS)

- Detailed design
  - design of algorithms and data structures (physical model), the second part of the answer to the question of how the software is supposed to do something,
  - justification of all design decisions,
  - implementation of units (code writing/generation),
  - testing of code units to demonstrate their compliance with requirements.

- Software validation
  - Demonstrating that the syste ality goals → **quality attributes**

RAMS:
- reliability,
- availability,
- maintainability
- safety

Usability

# Life-cycle processes (ECSS)

- Software verification
  - confirm that for each activity (phase) of the life cycle there is an appropriate set of documents specifying the requirements for the product of a given phase,
  - demonstrate that the product of a given phase is correct and fully compliant with these requirements.
- Implementation
  - create code units (coding, adaptation, modification, automatic code generation),
  - integrate system units

# Life-cycle processes (ECSS)

- **Maintenance**
  - keep the system running after bug fixes, modifications, reinstallations, hardware replacements, etc.
- **Delivery and acceptance**
  - Install the system in its target environment,
  - Assess it formally based on the created documentation (RB & TS).
- **Exploitation**
  - Provide support to the end system user (installation, ongoing administration, etc.).

# Life-cycle processes (ECSS)

- **Management**
  - project planning (activities, checkpoints, products, techniques and procedures),
  - risk identification, countermeasure methods,
  - principles of organizing and conducting reviews,
  - management of configuration and information flow in the team, time, budget and risk
  - ECSS management standards (’M’ series):
    - ECSS-M-ST-10-01C – Organization and conduct of reviews
    - ECSS-M-ST-10C – Project planning and implementation
    - ECSS-M-ST-40C – Configuration and information management
    - ECSS-M-ST-60C – Cost and schedule management
    - ECSS-M-ST-80C – Risk management

# Life-cycle milestones

- SRR - system requirements review:
  - The developer and client agree on the requirements baseline specification (is complete and consistent).
- PDR - preliminary design review:
  - The developer and client agree that the technical specification correctly reflects all basic requirements
- DDR - detailed design review:
  - Assessment of the possibility of moving to the next phase (all units designed correctly, realistic testing and integration plan, sufficient budget, unresolved issues addressed, existing software may be re-used)

# Life-cycle milestones

- CDR - critical design review:
  - a key decision to continue or close the project
- QR - qualification review:
  - The tools are adequate and the product is mature enough (TRL-8) for acceptance.
- AR - acceptance review:
  - all required test cases performed and completed correctly by a given software version in its target environment,
  - final approval of the product.

# Product maturity

- Technology readiness levels (NASA):

**TRL 9**
- Actual system “flight proven” through successful mission operations

**TRL 8**
- Actual system completed and “flight qualified” through test and demonstration (ground or space)

**TRL 7**
- System prototype demonstration in a space environment

**TRL 6**
- System/subsystem model or prototype demonstration in a relevant environment (ground or space)

**TRL 5**
- Component and/or breadboard validation in relevant environment

**TRL 4**
- Component and/or breadboard validation in laboratory environment

**TRL 3**
- Analytical and experimental critical function and/or characteristic proof-of-concept

**TRL 2**
- Technology concept and/or application formulated

**TRL 1**
- Basic principles observed and reported

# Quality attributes of a critical software system

- Basic (ECSS standard definitions):
  - reliability

    ***absence of errors that prevent the system from properly performing all functions required in its RB***
  - safety

    ***no threat to its environment (people, environment, property and infrastructure)***
  - maintainability

    ***can always be brought to a state in which any required function will be performed properly***
  - security

    ***correctly and completely achieves only the goals consistent with the owner's intentions***

# Quality attributes of a critical software system

- Complex (ECSS standard definitions):
  - availability
    *System is capable of performing the required function at a given moment or time interval*
  - dependability
    *Ability to build trust in the quality of system services in the long term*

- RAMS characteristics
  - **reliable + available + maintainable + safe**

# Classification of critical systems

- Severity number (SN):

| Effect | SN | Dependability | Safety |
|---|---|---|---|
| Catastrophic | 1 | Progressive break-down (propagation of a series of failures) | Loss of life, health or permanent disability of crew members or ground staff |
| | | | Loss of the system |
| | | | Permanent loss of connection to the manned flight control system |
| | | | Destruction of the launch pad |
| | | | Serious damage to the natural environment |

# Classification of critical systems

- Severity number (SN):

| Effect | SN | Dependability | Safety |
|---|---|---|---|
| Critical | 2 | Mission loss | Temporary inability to perform certain activities or illness of crew or ground staff |
| | | | Serious damage to the link to the manned flight control system |
| | | | Serious damage to ground infrastructure |
| | | | Significant damage to private or public property |
| | | | Other environmental damage |

# Classification of critical systems

- Severity number (SN):

| Effect | SN | Dependability | Safety |
| --- | --- | --- | --- |
| Significant | 3 | Significant threat to the mission | Mission dependent |
| Negligible | 4 | Minor threat to the mission | Mission dependent |

# Classification of critical systems

- Non-execution or incorrect execution of the code and other anomalies in its operation cause the system to fail with the following consequences:
  - **catastrophic (category 'A')**
  - **critical (category 'B')**
  - **significant (category 'C')**
  - **negligible (category 'D')**

# C. Planning of VVT processes

# Verification process

- Verification of the Requirements Baseline (RB) document by recipients (to do list):
  - Comprehensive description of the operating (target) environment
  - Characteristics of the system and devices
  - Key points where to control the system and observe its operation
  - Possible system malfunctions and ways to eliminate their effects
  - Specification of the initial system settings
  - Specification of user scenarios

# Verification process

- Verification of the Technical Specification (TS) document by recipients (to do list):
  - System hardware and software requirements are consistent
  - Software requirements are verifiable
  - System architecture is feasible
  - All hardware and software implementation limits have been identified
  - An appropriate verification method is defined for each requirement

# Verification process

- Verification of the system architecture design by developers (to do list):
  - System architecture design accurately reflects the requirements
  - Detailed system design is implementable
  - all dynamic aspects of system operation are correctly considered (processes/threads, their priorities, synchronization mechanisms, resource sharing management)

# Verification process

- Verification of the system detailed design by developers (to do list):
  - Is correct, internally consistent and clearly follows the system architecture design
  - Is testable:
    - *data entry points and triggers, measurement data collection points*
    - *temporary and invariant values in key places of the system structure*
    - *fault injection possible*
  - ability to perform maintenance and operational activities
  - all dynamic aspects of system operation are correctly considered (processes/threads, their priorities, synchronization mechanisms, resource sharing management)

# Verification process

- System code verification by developers (to do list):
  - its structure and content consistent with requirements (TS, RB), architecture and detailed design
  - is correct, testable and compliant with established coding standards
  - all possible consequences of run-time errors are under control of the code
  - there are no memory leaks
  - 100% code execution coverage for assignment and conditional statements in the event of possible catastrophic (category ’A’) or critical (category ’B’) consequences

# Verification process

- Verification of the unit testing plan and results by developers (to do list):
  - unit tests are consistent with the system design and requirements documents
  - Each unit test ensures examining (at least):
    - *execution of each conditional code statement (while, for, if) for the limit values of its predicate*
    - *access (read or write) to each global variable*
    - *input data outside of their valid ranges, causing incorrect function computations*
    - *high volume/intensity data inputs to test the unit's performance limits as specified in the requirements*
  - all results obtained are as expected and the completion criteria for each test have been met
  - all unexpected results and anomalies of each tested unit are documented in the report.

# Verification process

- Verification of the system integration by developers (to do list):
  - consistency with the system architecture design document
  - testing objectives for system interfaces (adequate and complete list)
  - obtained results obtained are consistent with the expected ones

# Verification process

- Verification of the system validation process by developers wrt RB and TS documents (to do list):
  - results of the validation process were obtained based on test cases, test procedures, inspections and design reviews covering the entire scope of requirements included in TS/RB documents
  - all obtained results of the validation process are consistent with the expected ones

# Verification process

- Verification of the system documentation by developers (to do list):
  - the content of the documentation is adequate, complete and consistent
  - all documents are prepared within the deadlines set up in the project time schedule
  - management of the process of creating/merging documents follows the previously defined procedures

# Verification process

- Hard real-time system analysis:
  1. the system is predictable, all worst case scenario events are handled within the required time limits (TS, RB)
     - *an adequate analytical model was used,*
     - *alternatively (if not possible) valid simulation experiments were carried out*
     - *feasibility of the architectural structure design demonstrated*
  2. time analyzes were updated at the detailed construction stage...
  3. ... and (again) repeatedly, during code verification, unit testing, and integration phases (based on information collected during dynamic analysis of the target system code).

# Validation process

- TS validation activities:
  - Test specification (test cases)
    - ✓ *for each requirement of each code unit (input data, expected results, test completion criteria)*
  - Test design (test scenarios)
    - ✓ *volume and stress tests, data limit and/or special values*
    - ✓ *testing the system ability to isolate or reduce the effects of errors (soft-fail systems, fault tolerant systems, interactive systems)*
    - ✓ *correct operation in various valid configurations of the target environment (supervised mode)*
    - ✓ *data interfaces (protocols, data ranges, time dependencies)*
    - ✓ *user interfaces (average user error rate, average time to learn)*

# Validation process

- RB validation activities:
  - Test specification (test cases)
    - ✓ *for each single requirement - the mission's intended inputs, expected results, and acceptance criteria*
  - Test design (test scenarios)
    - ✓ *volume and stress tests, data limit and/or special values*
    - ✓ *testing the system ability to isolate or reduce the effects of errors (soft-fail systems, fault tolerant systems, interactive systems)*
    - ✓ *correct operation in various valid configurations of the target environment (random mode)*
    - ✓ *data interfaces (protocols, data ranges, time dependencies)*
    - ✓ *user interfaces (average user error rate, average time to learn)*

# Delivery and installation process

- Activities:
  - Installation in the target environment
    - ✓ *installation plan and installation procedures development*
    - ✓ *testing of installation procedures (installed code, databases and services can be properly activated to function and close afterwards)*
    - ✓ *conducting introductory training for the end user staff (or even cyclic if requested in RB)*
    - ✓ *providing resources and information necessary to carry out the installation*
    - ✓ *testing the system ability to isolate or reduce the effects of errors (soft-fail systems, fault tolerant systems, interactive systems)*
    - ✓ *documenting all relevant events (incidents) during installation*

# Acceptance process

- Activities:
  - the recipient (client) prepares the acceptance testing plan
  - the recipient performs all tests specified in the acceptance testing plan
    - ✓ ***the tests must include generation of any executable code from the source code (!)***
    - ✓ ***evaluation of all obtained test results must refer to RB***
  - developer and recipient perform a formal acceptance review
    - ✓ ***after completion of the software delivery, installation and acceptance processes***

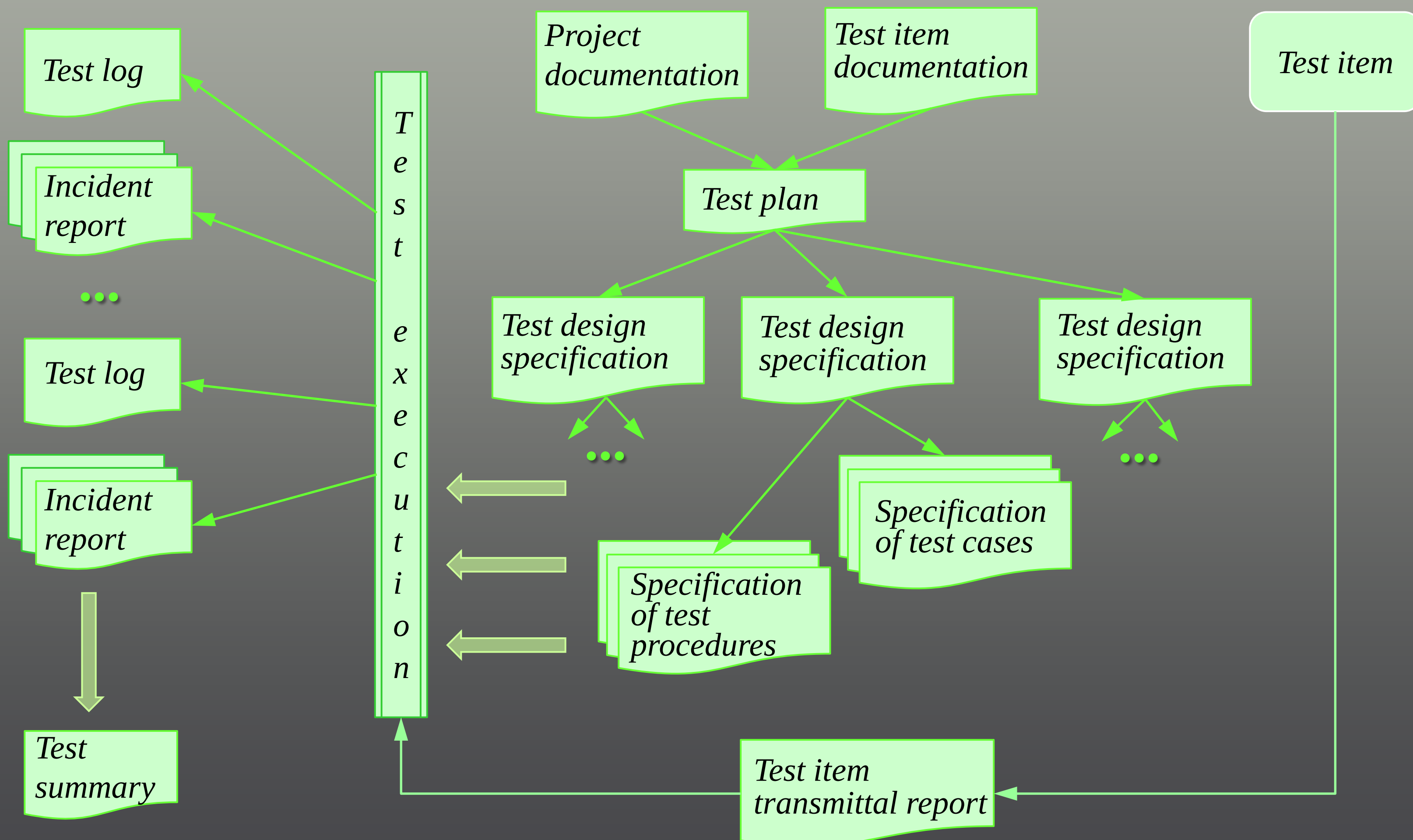
# Experiment management

1. Quality objectives
   – compliance with functional specifications, performance characteristics, code characteristics (Halstead. McCabe), test completion degree, etc.,
2. Anticipated problems
   – description of properties and how they could occur
3. Testing strategies
   – capable of detecting anticipated problems should affect their definition
   – project plan, Pareto effect

# Experiment management

4. Product delivery
   - incremental integration with ongoing analysis/testing, monitoring trends of detected anomalies
5. Change management, staff training, tools
   - project plan, Pareto effect

# Test documentation (IEEE std.)



# Permanent (static) part of test documentation

- Test plan
  - Time schedule, milestones/checkpoints, management rules
- Test design specification
  - Rationale, explanation and justification of test cases structure
- Specification of test cases
  - Input data and expected results, entry and exit conditions, events and expected reaction

# Permanent (static) part of test documentation

- Specification of test procedures
  - how to perform experiments and measure their advance
- Test completion criteria
  - Conditions to be met by each test procedure
- Test item transmittal report
  - method of delivery and format of items to be tested

# Variable (dynamic) part of test documentation

- Test log
  - recorded activities and data
- Incident report
  - list of incidents requiring further investigation
- Test summary
  - decision of the project management staff and conclusions

# Test procedures

- Disposition of test items
- Exceptional situations
- Experiment costs
- Acceptance criteria

# Disposition of test items

- Item identifier
- Item:
  - version, documentation, responsible person
- Method of delivery:
  - localization, medium
- Status:
  - deviations from documentation, previous version  and/or plan, any modifications in progress
- Authorization:
  - person approving disposition

# Exceptional situations

- Any incident during the experiment requiring explanation:
  - input data, expected results, observed anomalies, date and time, step in the scenario, state of the environment, repetition attempts, people performing the test, witnesses
- Determining consequences wrt:
  - continuation of the test plan, test design, scenarios, etc.

# D. Static analysis techniques

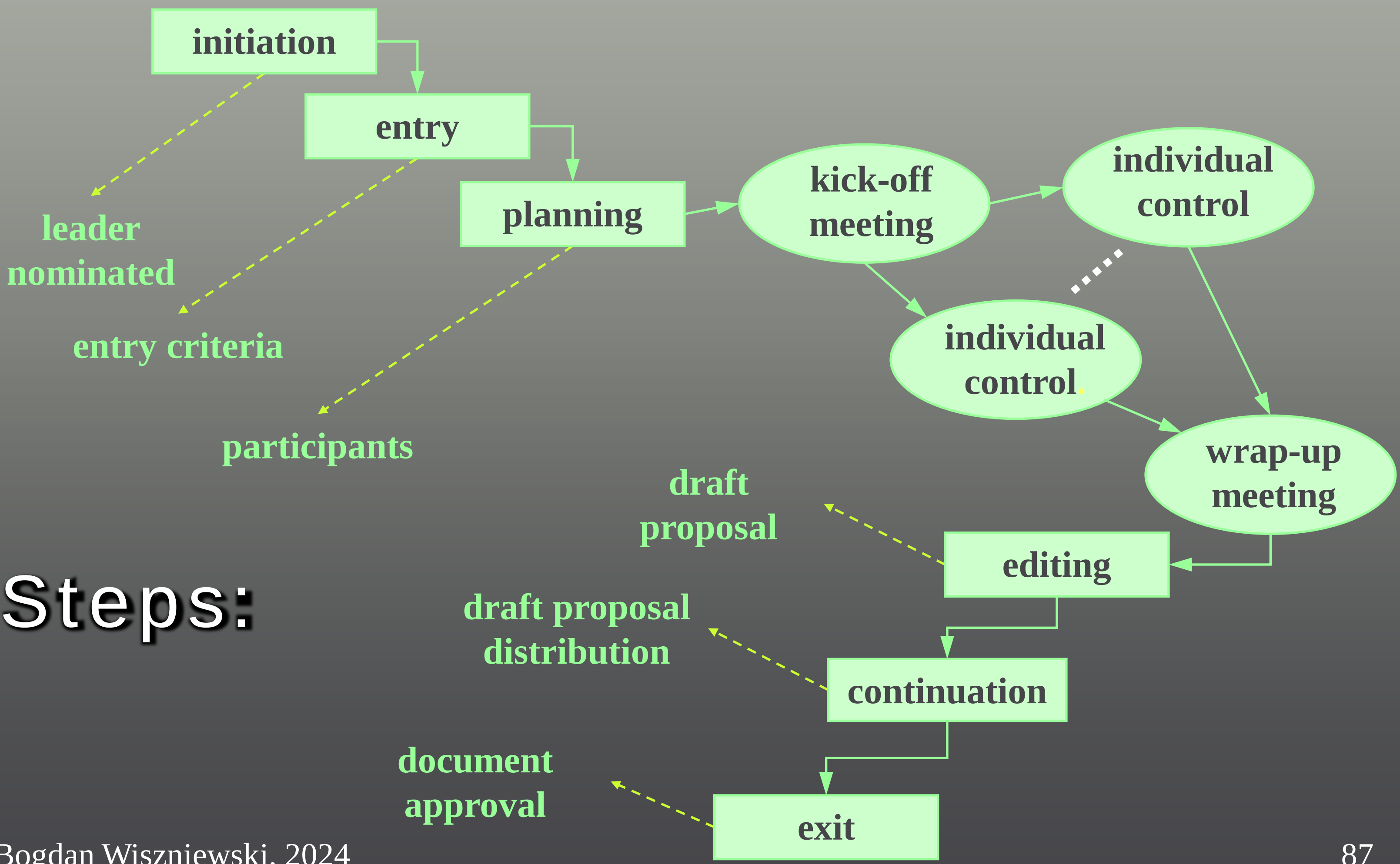
# Software inspections

- Visually inspect the code or design of a system component to detect:
  - errors,
  - deviations from project standards,
  - missing or incorrect comments,
  - potential portability problems,
  - other problems not ”machine checkable”

# Software inspections

- Inspection is not a part of the design process:
  - decisions are not made, focus on individual issues only, does not suggest changes or corrections, but
  - allows to detect, identify and remove defects and formally confirm product quality

# Software inspections




- Steps:

# Software inspections

- Inspection team members (roles)
  - Author: *created the work product being inspected.*
  - Moderator (leader): *plans the inspection and coordinates it.*
  - Inspector: *examines the work product to identify possible defects.*
  - Reader: *reads through the documents, one item at a time. The other inspectors then point out defects.*
  - Recorder/Scribe: *documents the defects that are found during the inspection.*

# Reviews

- Developer (programmer) leads the team through a selected fragment of code
- The team asks questions and comments on potential errors

***(!) a narrow and highly interactive technique***

# Software audits

- Assessing software processes and products for compliance with requirements, standards, and contractual agreements;
- Ensuring software quality, accuracy, and functionality while reducing legal risks and optimizing performance efficiency;
- Strictly defined criteria and goal, independent assessment team

# Software audits

- Types:
  - Technical audit: the software is developed with respect to industry standards.
  - Security audit: the software can protect sensitive information.
  - Usability and accessibility audit (UX audit): there are no issues with User Experience in the already-deployed software.

# Identification of critical elements

- Failure Mode and Criticality/Element Analysis (FMECA/FMEA)
  - NASA since the 1960s, currently the space, aviation, nuclear and automotive industries
  - ECSS standard, ISO 9000 norm
  - analysis of the effects of defects revealing individually in the products of the architectural/detailed design phases, production phases (coding, testing, integration) as well as flaws of the production process itself
  - most (>80%) defects are detected in the production phase

# Failure mode and effects analysis (FMEA)

- Steps:
  1. Identification of system elements and activities of the production process
  2. List all potential product defects and errors in the activities of individual phases of the production process
  3. List all probable consequences of the potential defects and errors
  4. List possible causes of the identified defects and errors
  5. Analyze all identified defects to :
     a. assess the materialization of risks
     b. planning risk mitigation
  6. Implementation of preventive actions and monitoring their effectiveness.

# Failure mode and effects analysis (FMEA)

- Criticality (CN) is the combination of end effect probability (PN) and severity (SN), CN = SN x PN
  - Criticality number (CN) to rank the risk level;
  - Severity number (SN) to rank severity for the worst-case scenario adverse end effect or state, e.g. catastrophic (4), critical (3), significant (2), negligible (1);
  - Probability number (PN) to classify of the ranges of probabilities of propagation of the effects of revealing a defect beyond the analyzed system unit, e.g.

| Level | Range | PN |
|---|---|---|
| High | $P > 10^{-1}$ | 4 |
| Moderate | $10^{-3} < P \leq 10^{-1}$ | 3 |
| Low | $10^{-5} < P \leq 10^{-3}$ | 2 |
| Negligible | $P \leq 10^{-5}$ | 1 |

# Analysis of critical system elements

- Identification of critical elements:

| Effect | SN | Probability levels $10^{-5}$ | $10^{-3}$ | $10^{-1}$ | $1$ |
|---|---|---|---|---|---|
| | | PN | | | |
| | | 1 | 2 | 3 | 4 |
| Catastrophic | 4 | 4 | 8 | 12 | 16 |
| Critical | 3 | 3 | 6 | 9 | 12 |
| Significant | 2 | 2 | 4 | 6 | 8 |
| Negligible | 1 | 1 | 2 | 3 | 4 |

– the analyzed unit is critical when:

***can always lead to a system catastrophic failure regardless of the defect propagation probability level or its index CN≥6***

E. Models for dynamic analysis:

- error,
- program,
- environment

# Where do errors come from?

- The error concept
- Error detection
- Characteristics of code objects under test
- Sources of errors

# The concept of a program error

- Program error
  - an event initiated by a user or the program environment,
  - the program code produces an unexpected result
- Program failure
  - the program crashes
  - the program is unable to perform some of its functions correctly

# Error detection

- Proof of correctness:

| | |
|---|---|
| Objective: | *to prove that the program is free of errors (is correct)* |
| Environment: | *axiomatic* |
| Reasoning: | *deduction* |

# Error detection

- Testing:

| | |
|---|---|
| Objective: | *to demonstrate that the program has errors* |
| Environment: | *testing or target* |
| Reasoning: | *inductive* |

# Error detection

- Performance testing:

Objective: *measure physical parameters*

Environment: *testing or target*

Reasoning: *metrics, characteristics*

# Characteristics of code objects

- Linguistic metrics:
  - Lines of code (LOC),
  - Statement count (SC),
  - Halstead’s metrics.
- Structural metrics:
  - Cyclomatic complexity (McCabe)
- Functional metrics:
  - Computational complexity (time, memory)

# Halstead's metrics

- Program length

$$Nlok = N_1 + N_2$$

- Estimated program length

$$HLOC = n_1 \cdot \log_2 n_1 + n_2 \cdot \log_2 n_2$$

$N_1$ operators, $N_2$ operands

$n_1$ unique operators, $n_2$ unique operands

# Halstead's metrics

- Program volume

$$VOLM = (N_1+N_2) \cdot log_2(n_1+n_2)$$

- Estimated number of errors

$$B = (N_1+N_2) \cdot log_2(n_1+n_2)/3000$$

$N_1$ operators, $N_2$ operands

$n_1$ unique operators, $n_2$ unique operands

# Structural metrics

- Cyclomatic complexity (McCabe):

$$M = L - N + 2P$$

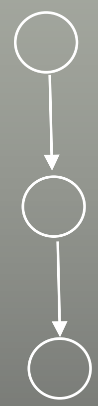
Number of edges (L),

Number of nodes (N),

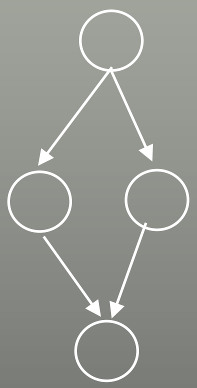
Number of connected components (P)

The maximum number of linear, independent paths through a program

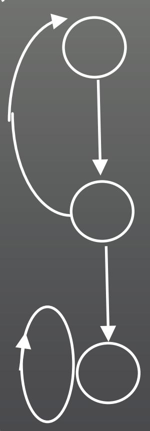
# Cyclomatic complexity



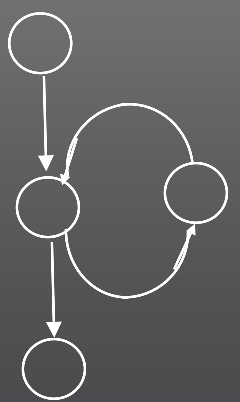
*L = 2, N = 3, P = 1*
*M = 2-3+2 = 1*



*L = 4, N = 4,*
*P = 1*
*M = 4-4+2 = 2*



*L = 4, N = 3, P = 1*
*M = 4-3+2 = 3*



*L = 4, N = 4,*
*P = 1*
*M = 4-4+2 = 2*

# Cyclomatic complexity

- Text (decision instructions) and detailed design

*→ Missing or redundant path*

- Number of (paths) test cases ≥ M

*(!) Control flow direction is not taken into account, e.g.M(if-then-else) = M(while-do)*

- Ignore language (syntax) complexity

# Functional metrics

- Complexity level (algorithms):

| *Symbol* | *Complexity* | *Example* |
|---|---|---|
| Θ(1) | constant | hash tables |
| Θ(log n) | logarithmic | binary search |
| Θ(n) | linear | GCD of n-digit numbers |
| Θ(n log n) | linearithmic | Fast Fourier Transform (DFT) |
| $\Theta(n^c)$ | polynomial | path tracking (robots) |
| $\Theta(c^n)$ | exponential | generation of prime numbers |

# Sources of errors

- Requirements specification:
  - completeness
  - consistency

# Sources of errors

- Design:
  - correctness
  - testability

# Sources of errors

- Coding ("translation" of an algorithm into some program code):
  - textual (typos, omissions, etc.)
  - misunderstanding semantics of the implementation language,
  - not understanding semantics of the algorithm,
  - not understanding (knowing) the requirements.

# Models

- Program:
  - Control flow,
  - Events,
  - Data flow,
  - State transitions

# Models

- Error:
  - Control flow errors
  - Data flow errors
  - State errors
  - Text anomalies

# Models

- Run-time environment:
  - Sequential (stream) processing
  - Even driven sequential proccessing
  - Concurrent processing
  - Parallel processing
  - Distributed processing

# Example

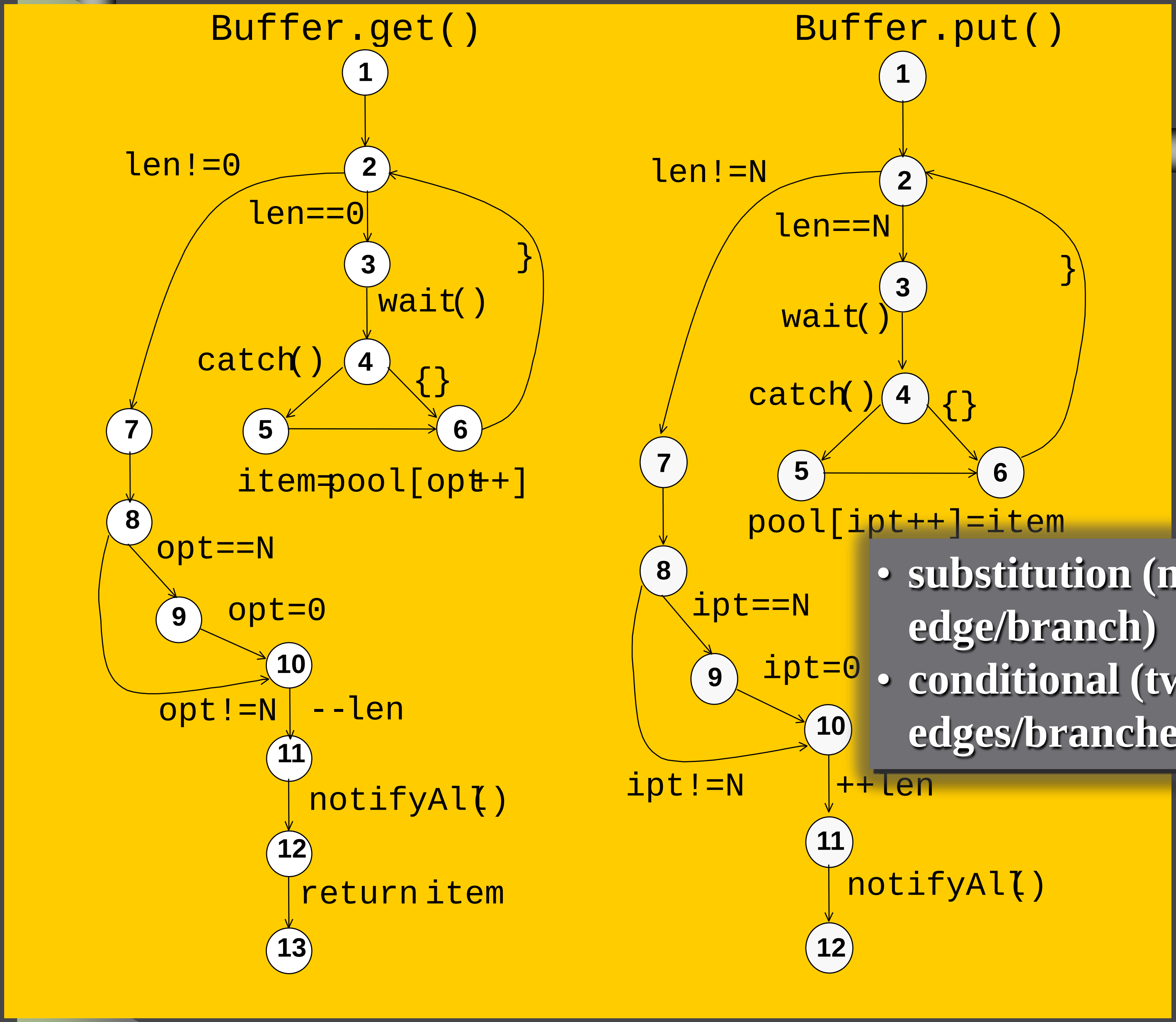
```
public class Buffer {
int N = 10; //total buffer capacity
ipt = 0; //input index
opt = 0; //output index
len = 0; //current buffer load

/** shared resource **/
char *pool = new char[N];

/** consumer **/
/* 1 */  public synchronized char get(){
         char item;
/* 2 */  while (len == 0){
/* 3 */     try {wait(); //buffer is empty
/* 4 */   }catch(Interrupted Exception){
/* 5 */     }
/* 6 */  } //semaphore opened
/* 7 */  item = pool[opt++];
/* 8 */  if (opt == N)
/* 9 */      opt = 0; //modulo N
/*10 */  --len; //one element taken
/*11 */  notifyAll(); //buffer is not full
/*12 */  return item;
/*13 */  }
/** producer **/
/* 1 */  public synchronized void put(char item){
/* 2 */  while (len == N){
/* 3 */     try {wait(); //buffer full
/* 4 */   }catch(Interrupted Exception){
/* 5 */     }
/* 6 */  } //semaphore opened
/* 7 */  pool[++ipt] = item;
/* 8 */  if (ipt == N)
/* 9 */      ipt=0; // modulo N
/*10 */  ++len; // one element added
/*11 */  notifyAll(); //buffer is not empty
/*12 */  }
}
```

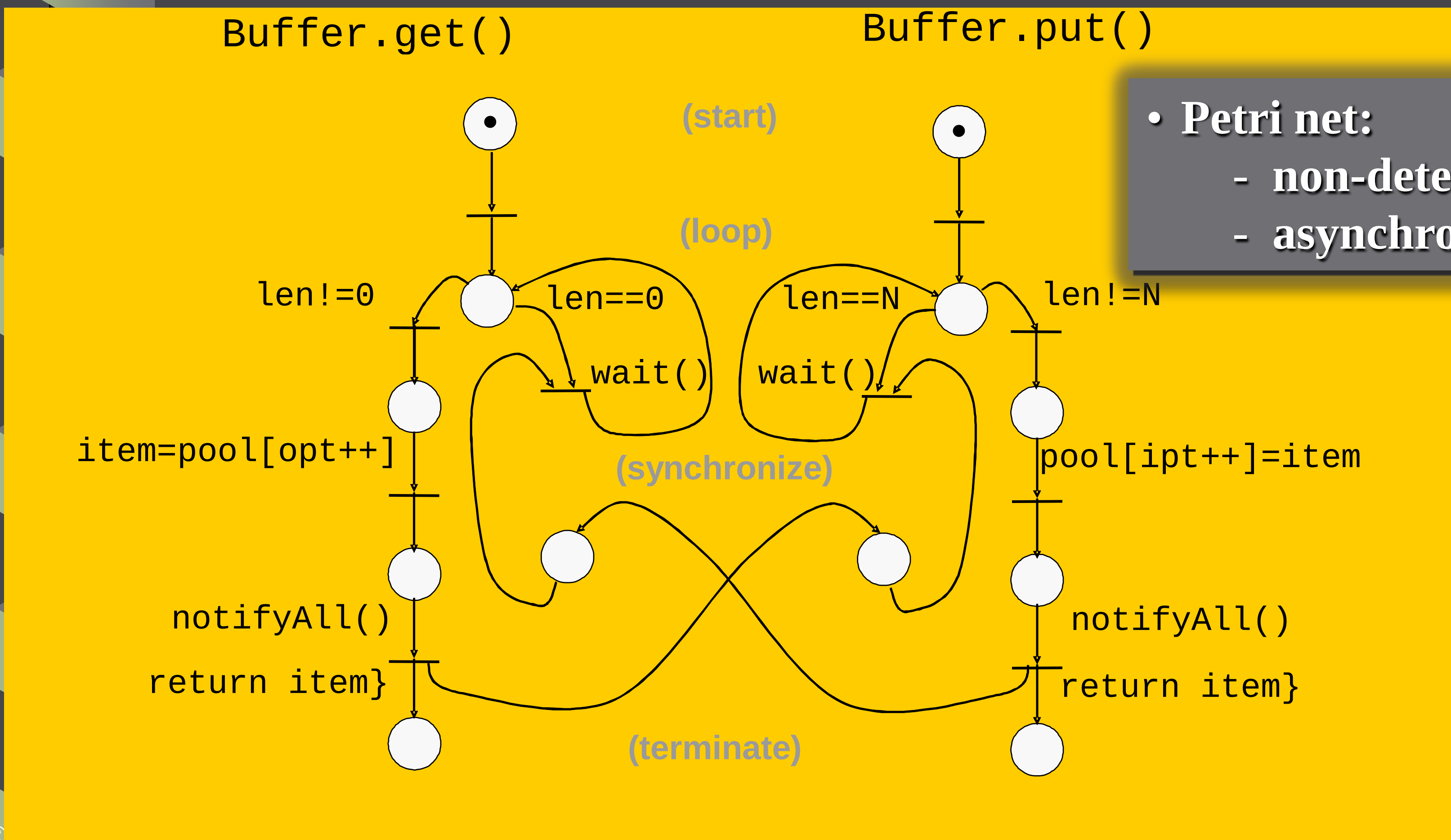
- Shared table `pool` with **N** elements
- Method `Buffer.get()` to fetch one character
- Method `Buffer.put()` to insert one character

# Control flow



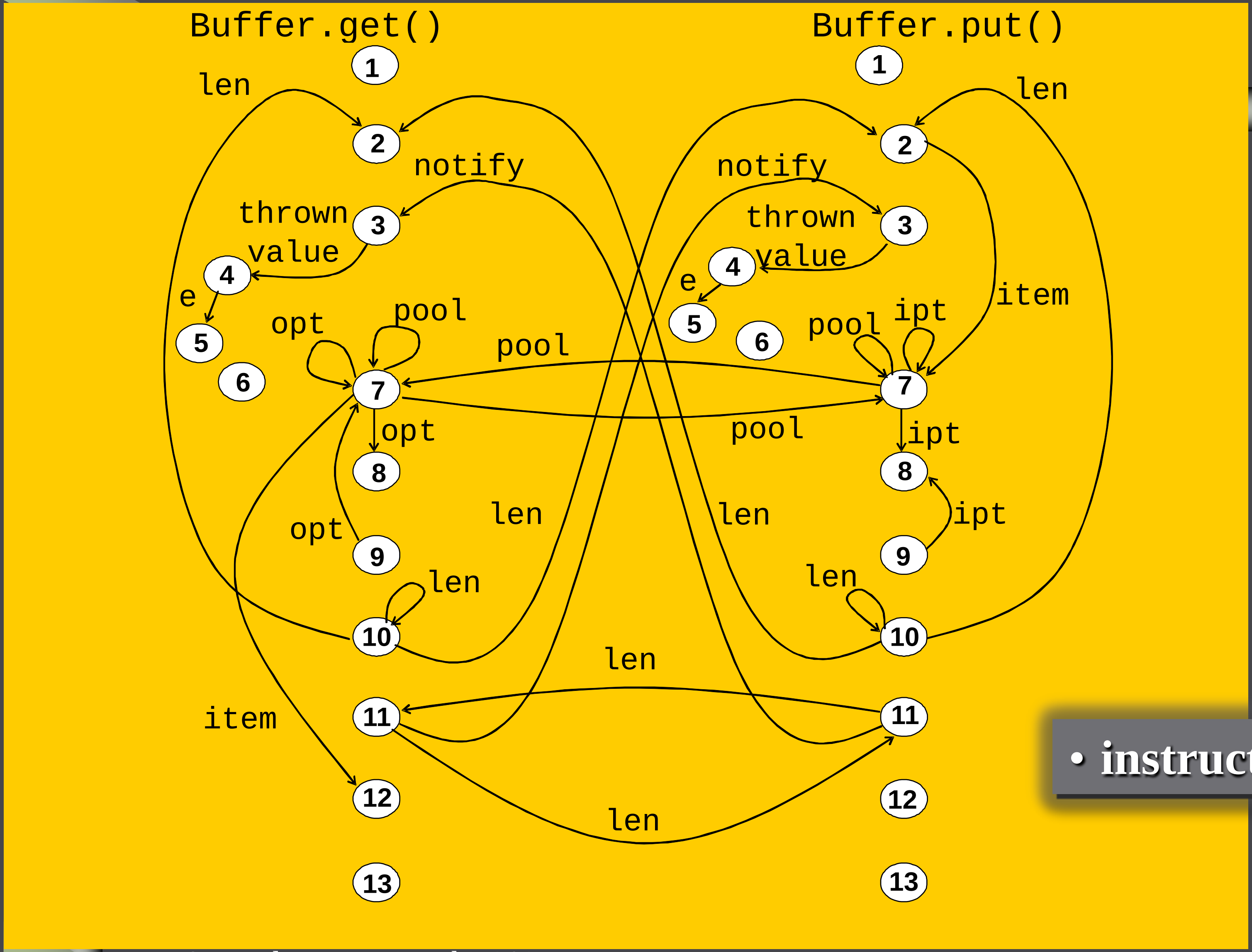

- substitution (none or one outgoing edge/branch)
- conditional (two or more outgoing edges/branches)

# Events



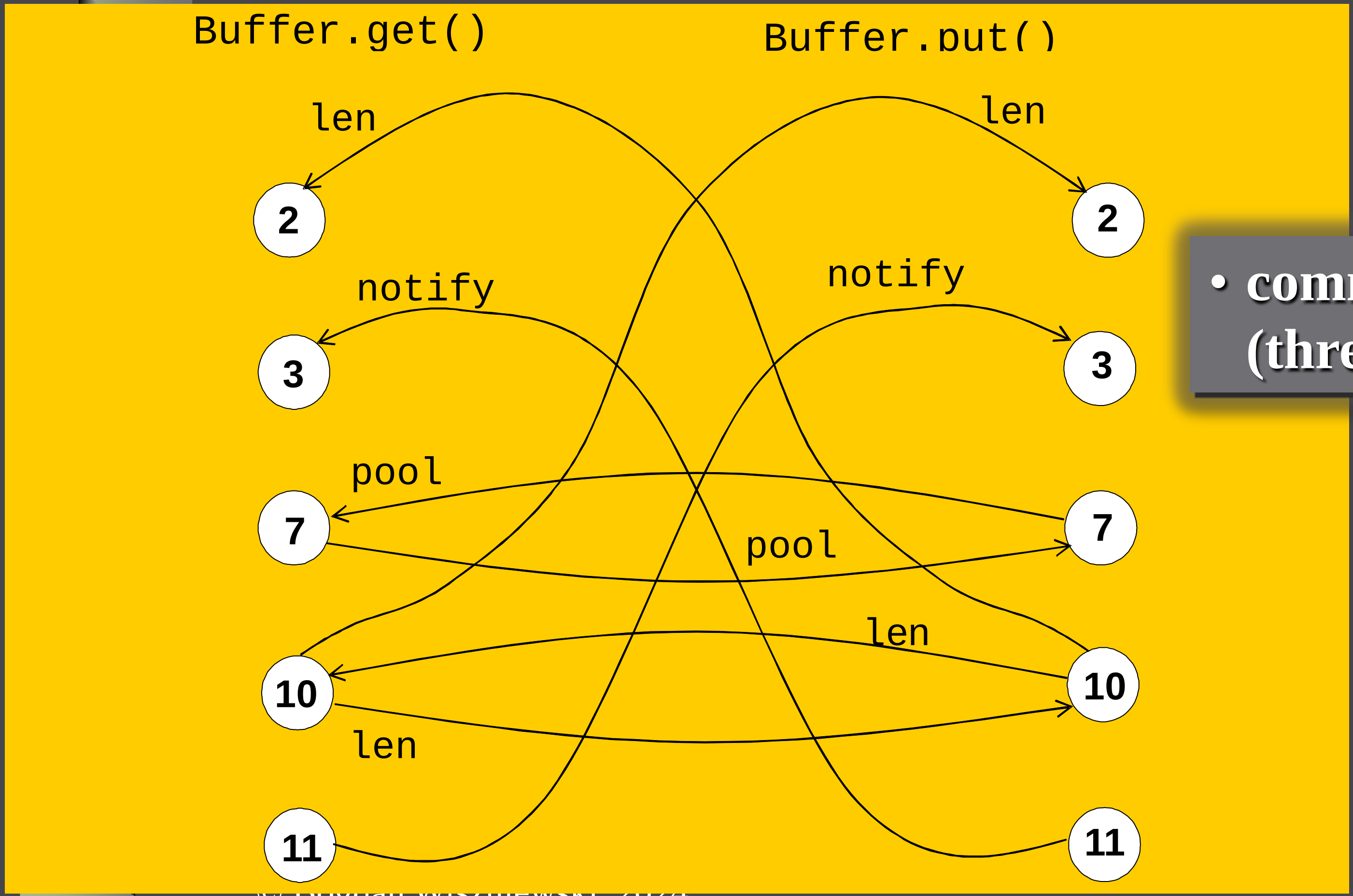

- **Petri net:**
  - **non-determinism**
  - **asynchronism**

# Data flow



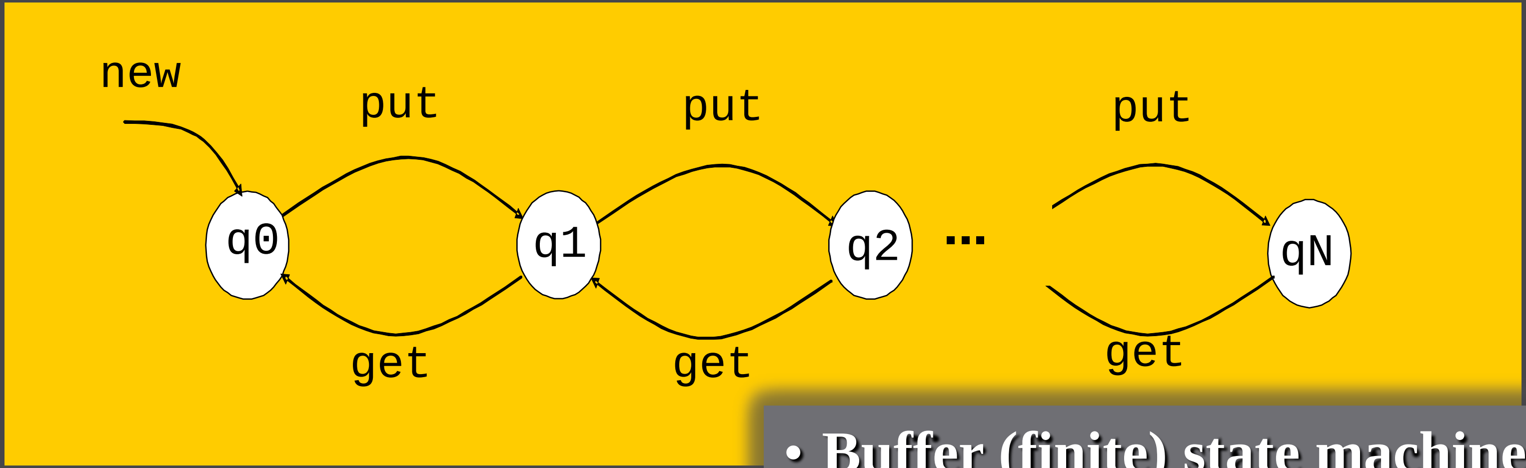

- **instruction level**

# Data flow



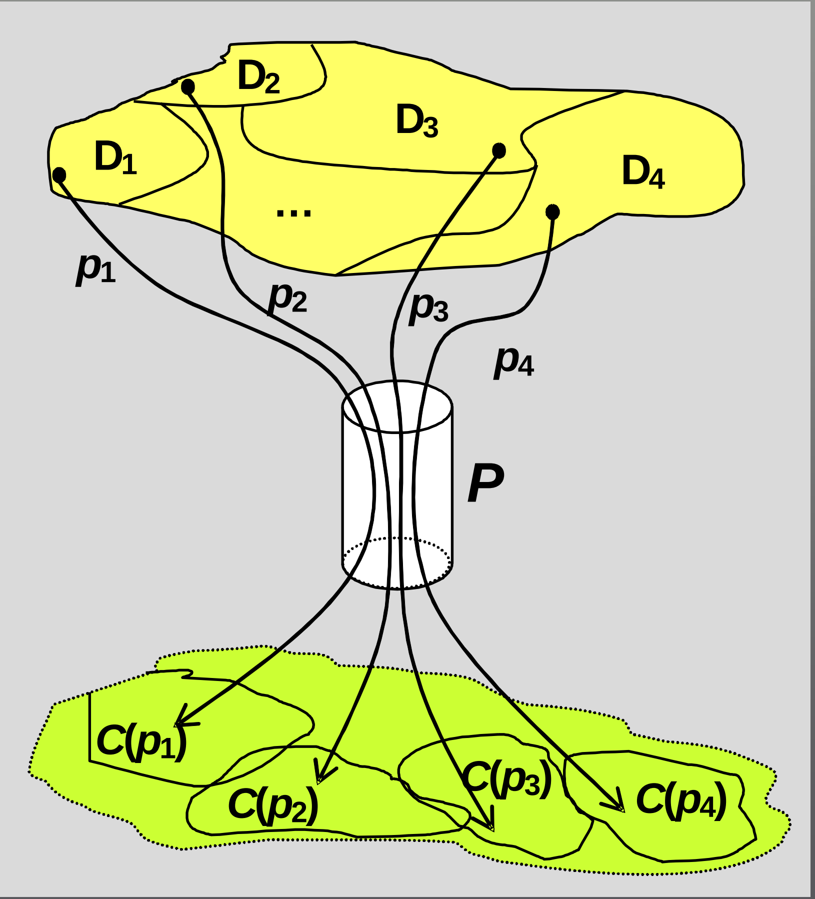

- **communication level (threads)**

# State transitions




- **Buffer (finite) state machine**

# Control flow errors



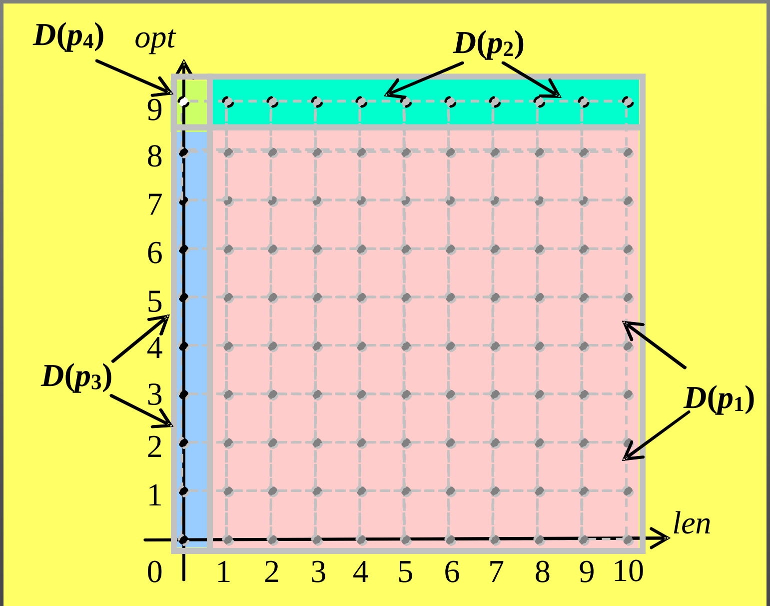

Path (execution) conditions:

$$p_1(\bar{x}) = (len_0 \neq 0) \wedge (opt_0 + 1 \neq 10)$$

$$p_2(\bar{x}) = (len_0 \neq 0) \wedge (opt_0 + 1 = 10)$$

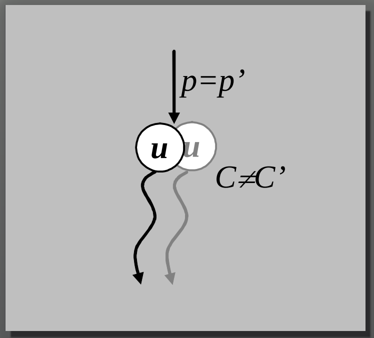
$$p_3(\bar{x}) = (len_0 = 0) \wedge (opt_0 + 1 \neq 10)$$

$$p_4(\bar{x}) = (len_0 = 0) \wedge (opt_0 + 1 = 10)$$

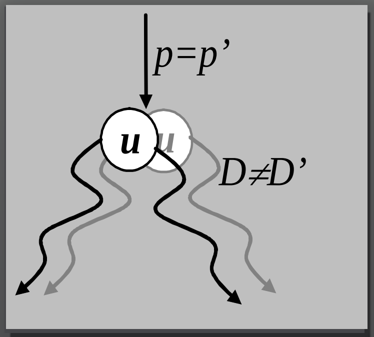


# Control flow errors

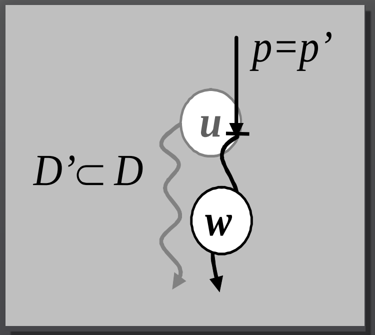
1. Path ‘computation’ error:




2. Path ‘domain’ error:

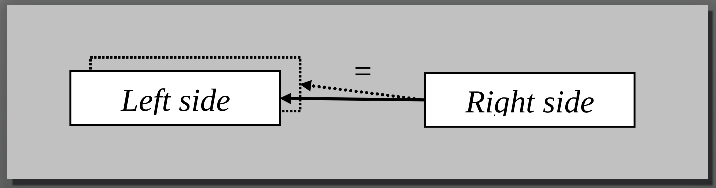



3. ‘Subcase’ (missing path) error:

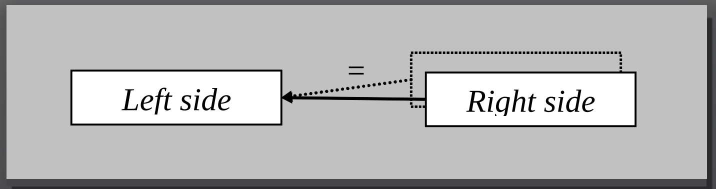


# Control flow errors

1. Value assigned to a wrong variable:



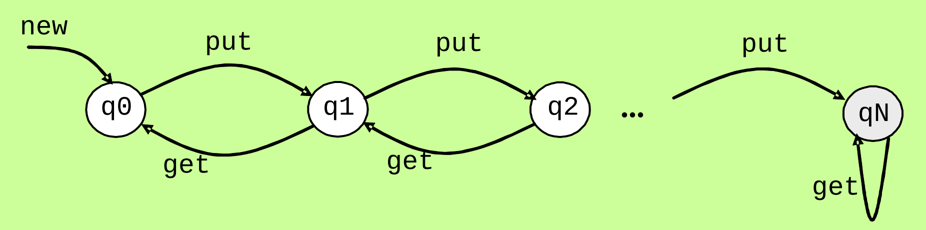

2. Variable assigned a wrong value:



# State errors

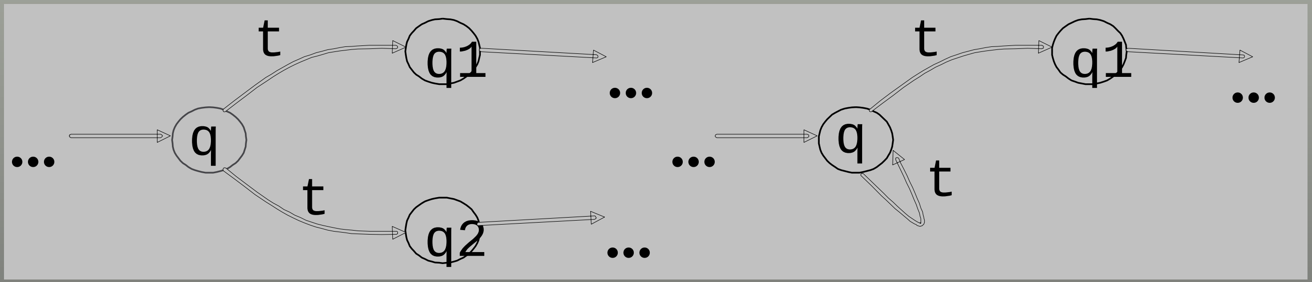
- Deadlock

```
public synchronized char get() {
        char item;
        while (len == 0) {
            try {
                wait(); //buffer is empty
            } catch (InterruptedException e) {}
        } //semaphore opened
        item = pool[opt++];
        if (opt == N) opt = 0; // modulo N
        --len; //one element taken
/*      notifyAll(); //buffer is not full */
        return item;
 }
```

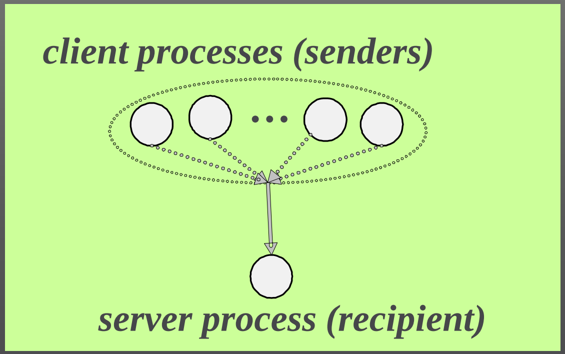


# State errors

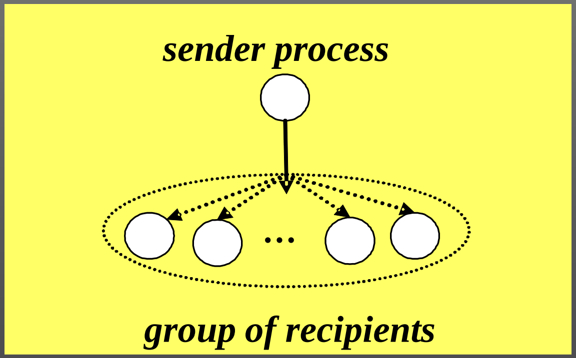
- Races




- on reception:




- on sending:



# Text anomalies

- Interpretation of syntax

```
public synchronized char get() {
    char item;
    while (len == 0);
        try {
         wait(); //buffer is empty
       } catch (InterruptedException e) {}
    } //semaphore opened
    item = pool[opt++];
    if (opt == N);
          opt = 0; // modulo N
    notifyAll(); //one element taken
     --len; //one element taken
    return item;
```

*redundant semicolon* (while)

*empty statement* (while)

*redundant semicolon* (if)

*empty statement* (if)

# Text anomalies

- Side effects

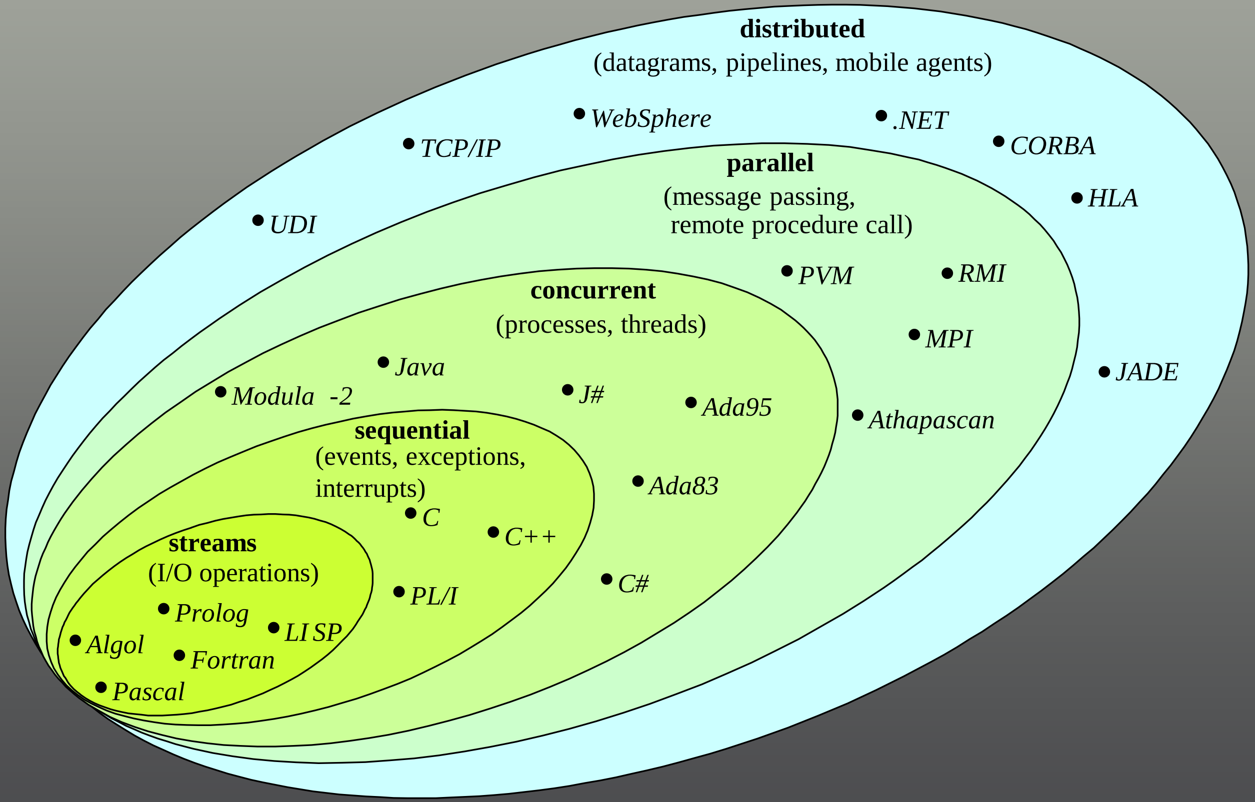
```
int main() {
        int x,y;
        int *z;
        z=&x;
        z++=1; /* initialization of x */
        z=2; /* alleged initialization of y */
}
```

# Text anomalies

- Implicit type conversion

```
…
void ff(int); // function with one int argument
…
int ival=3.14; // value 3.14 narrowed to 3, ival=3;
ff(3.14); // value 3.14 narrowed to 3,
          // ff(3) called;
ival=4.0; // conversion of 4.0 to 4 (not narrowed),
ival=4;
…
double fval=5; // promotion of 5 to 5.0 of a „wider”
               // type, fval=5.0;
int val=1;
fval=val+3.14; // promotion of 1 to 1.0 of a „wider”
               // type, fval=1.0+3.14;
…
```

# Run-time environment models

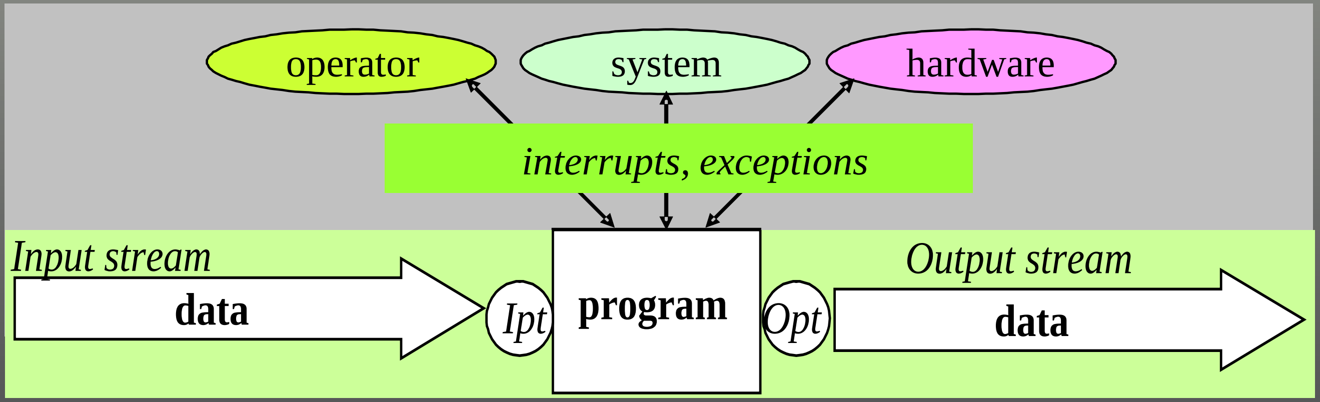


# Run-time environment models

- Sequential stream processing

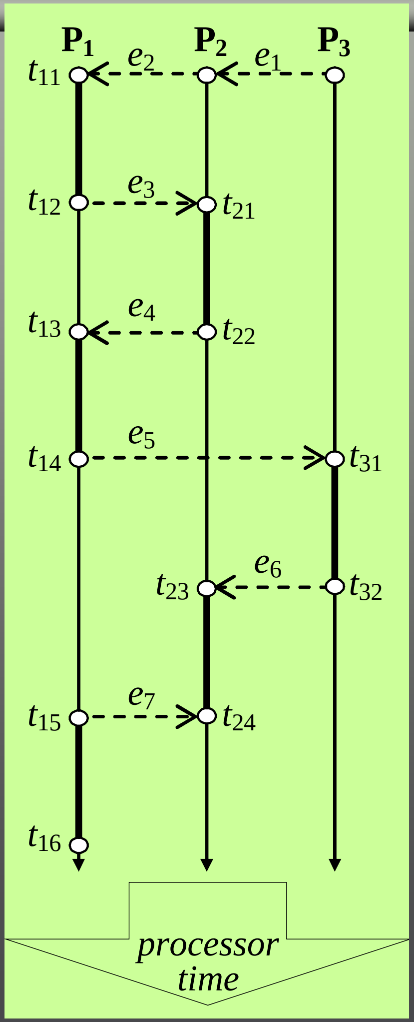
# Run-time environment models

- Event driven sequential stream processing



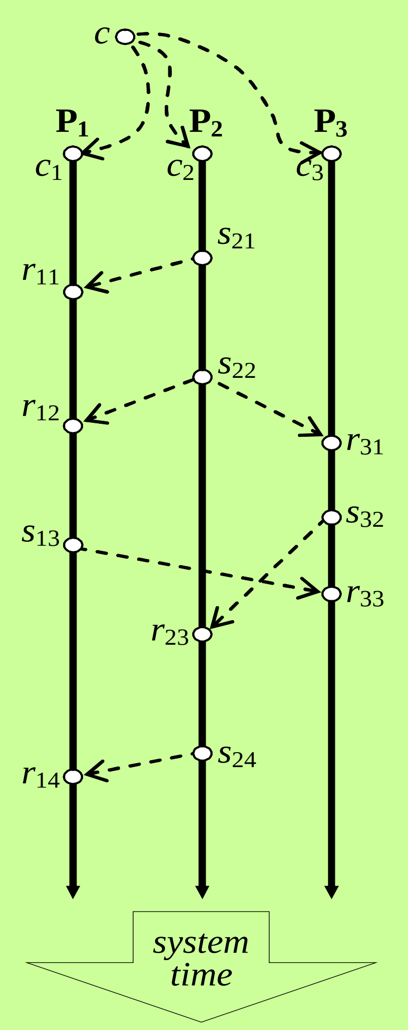
# Run-time environment models

- Concurrent processing



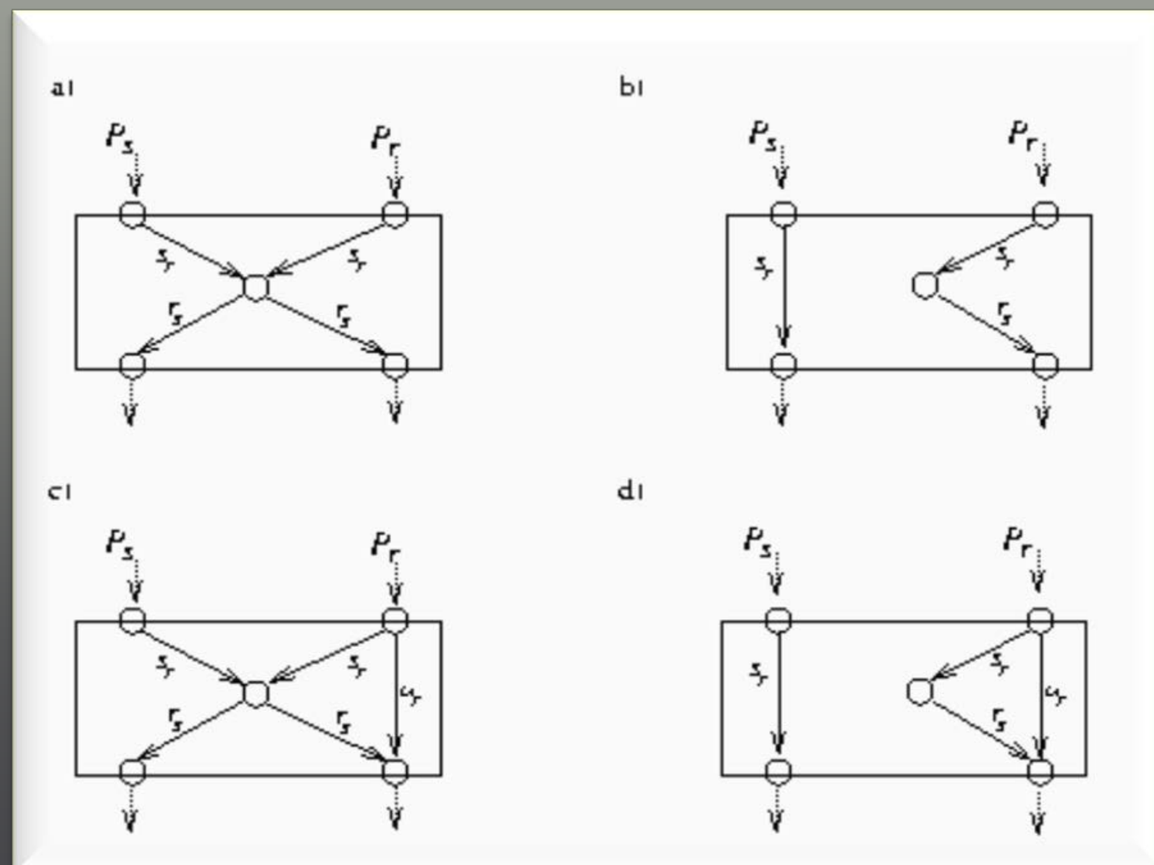
# Run-time environment models

- Parallel processing



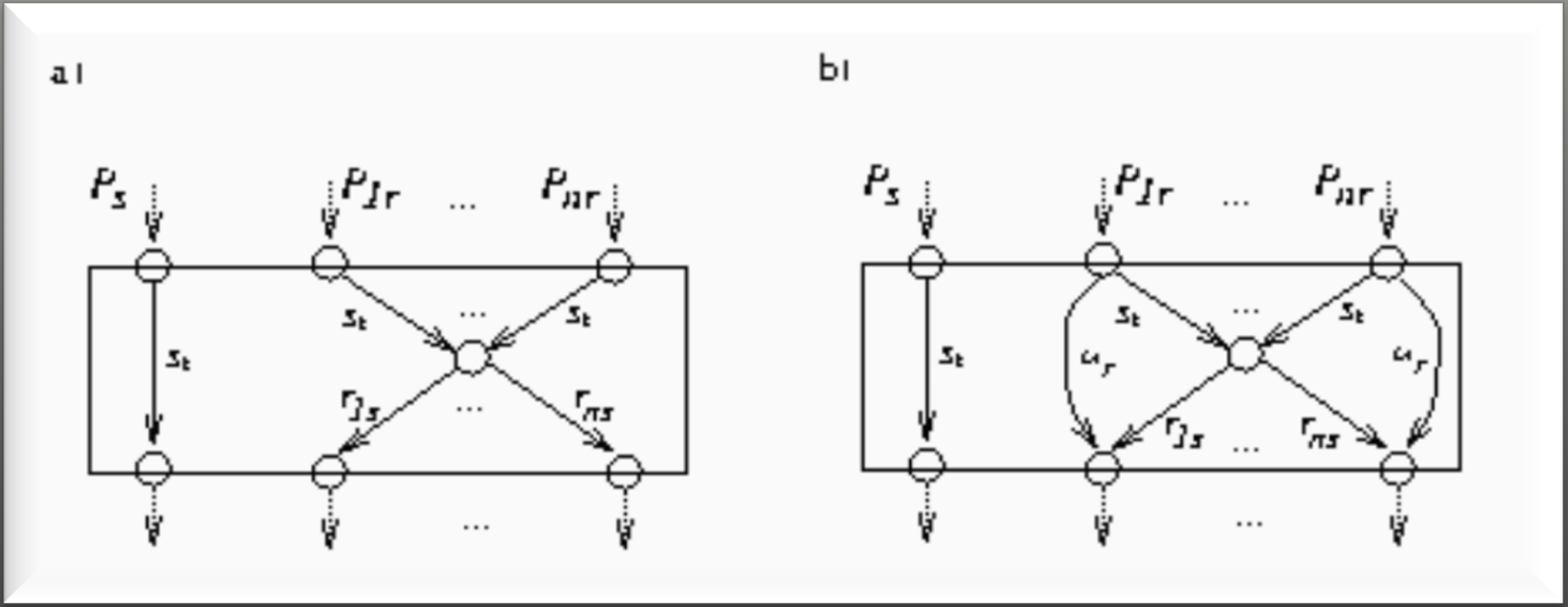
# Run-time environment models

- Communication events (1-1)



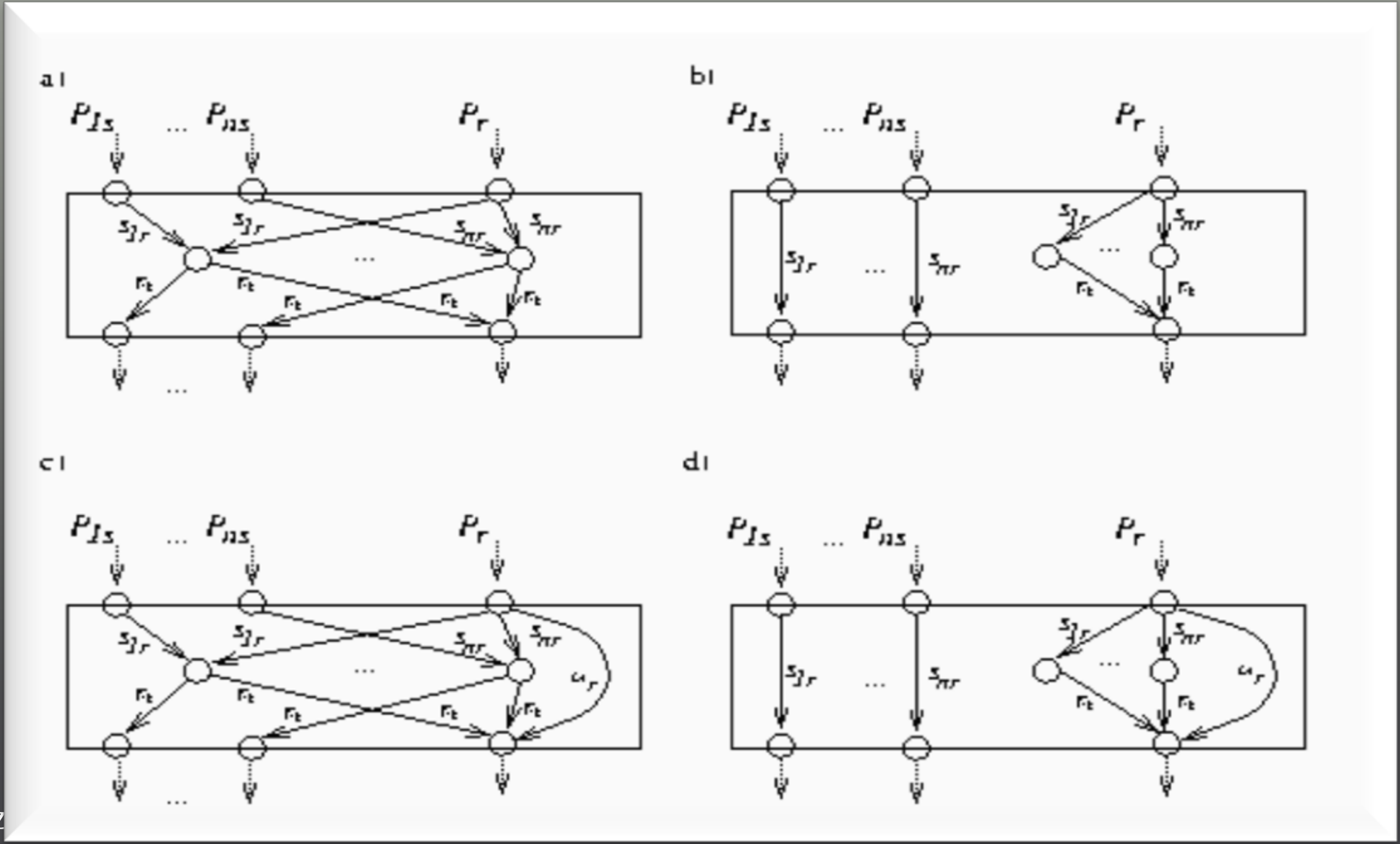
# Run-time environment models

- Communication events (1-n)



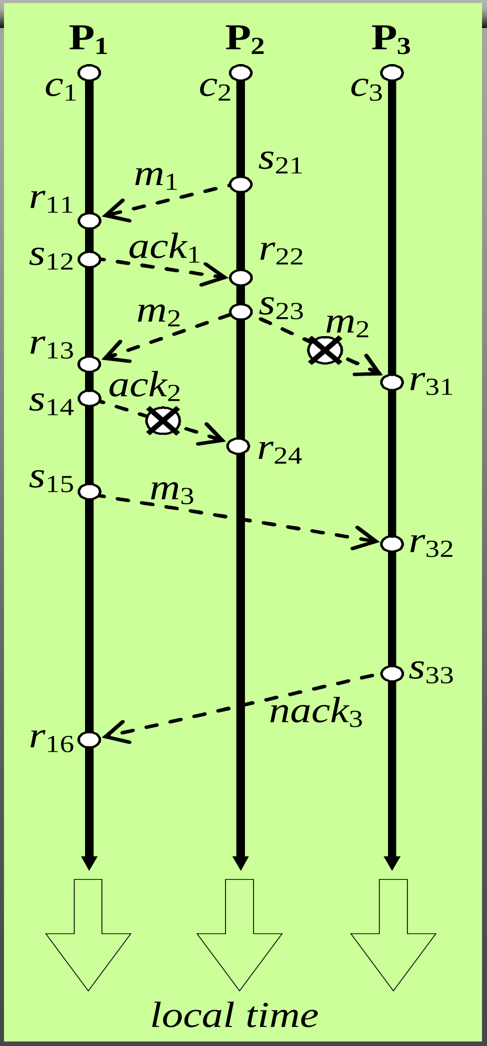
# Run-time environment models

- Communication events (n-1)



# Run-time environment models

- Distributed processing



# Dynamic analysis techniques

- Black-box testing:
  - Program ≡ function,
  - Test cases based on requirements specification
  - Potentially all errors but practically in an infinite time

# Dynamic analysis techniques

- White-box (structural) testing:
  - Program ≡ structure,
  - Test cases based on technical (architectural/detailed design) specification or the program code
  - Not all errors but in a (practically) predictable time

# F. Black-box testing

# Black-box strategies


*specification*

| input | output |
|---|---|
| T1 | R1 |
| T2 | R2 |
| ... | ... |
| Tn | Rn |

*program*

T1, T2, ..., Tn→ R1, R2, ..., Rn

# Black-box strategies

- Mathematical property:

$$T = \{t_i \mid i=1,\ldots,N\},\ F'(T) = F(T) \Rightarrow F'(x) \equiv F(x)$$

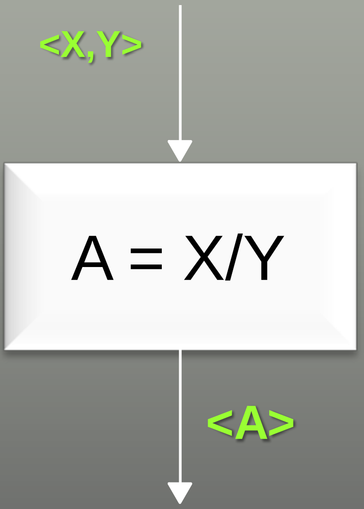
- Limitations:
  - ! Undecidability of function equivalence (even of primitive recursive functions!)
  - ! Approximate binary arithmetic (floating point error, rounded value, register overflow error)

# Black-box strategies

- Special values
- Transcendent values
- Polynomial equivalence
- Monte-Carlo testing

# Special values

- Example




<...,1> ?   A=X, A=X*Y
<4,2> ?   A=X-Y
<0,...> ?   A=X, A=X*Y, decl(A)

<a,b>   <a != 0, b = 0>
<a !=0, b!=0, a>>b>, np. a=max(float), b=min(float)
<a =0, b=0> ?

<a !=0, b !=0>, b<a, b=prime

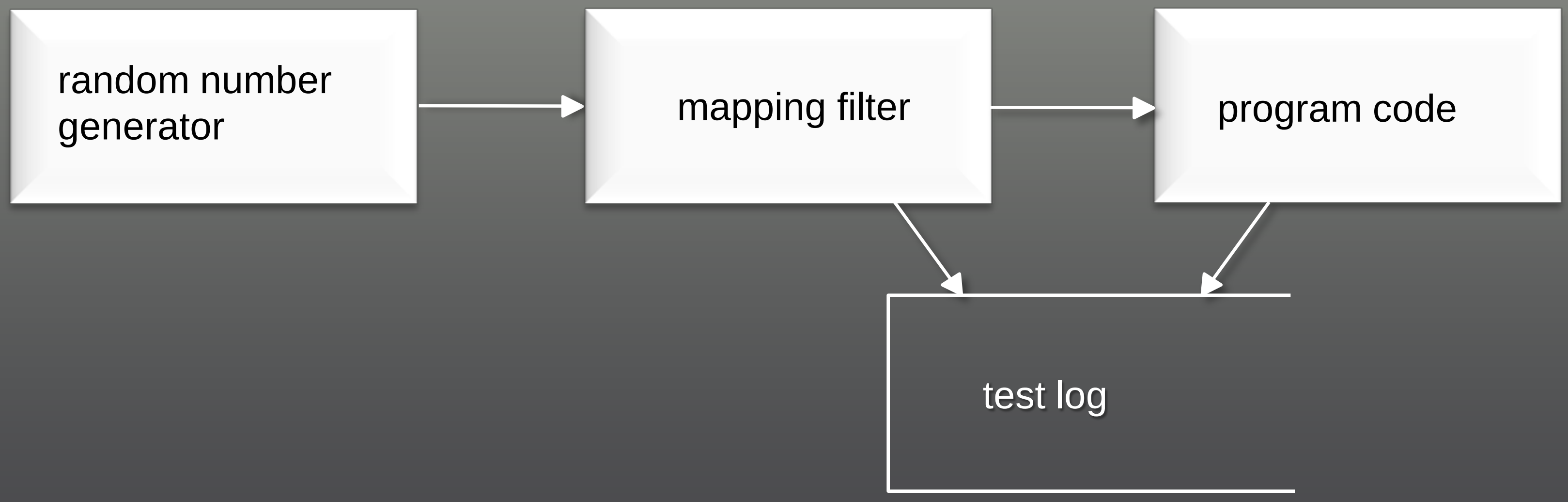
# Polynomial equivalence

*(!) Standard math functions are computed using polynomials*

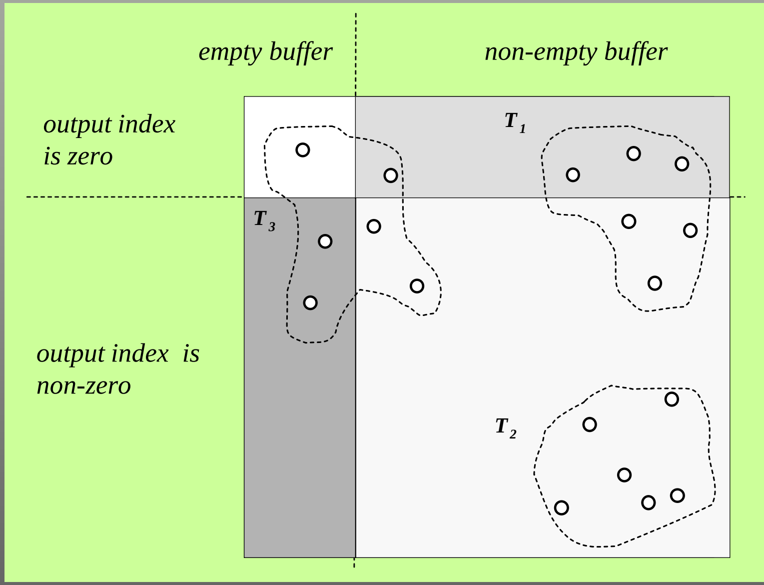
- Classic polynomial algebra:
  - Class of polynomials cf(n,x),
  - Tested F, specified $F' \in CF$
  - $T \equiv \{t_1, t_2, \ldots, t_{n+1}\}$
  - $F'(T) \equiv F(T) \Rightarrow F'(x) \equiv F(x)$

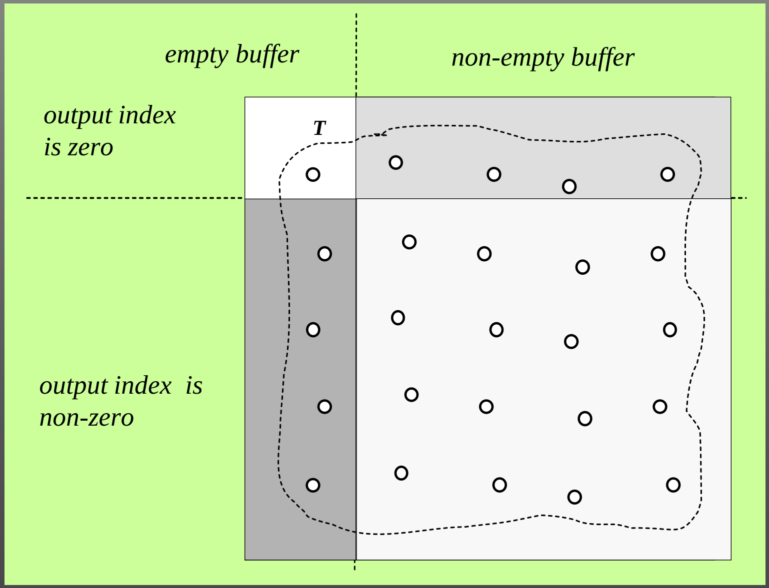
# Monte-Carlo testing

*(!) Exercise the program for its most typical and common input values*



# Monte-Carlo testing



# G. White-box testing

# White-box testing strategies

- Structural model (program, system)
  - control flow testing
  - data flow testing
  - mutation testing

# White-box testing strategies

- Test evaluation:
  - quantitative (metrics) → *rule of thumb*
  - qualitative (model) → *errors are deviations*
- Passing a test:
  - all required test cases exercised
  - all results obtained consistent with the expected ones

→ *test strategy*

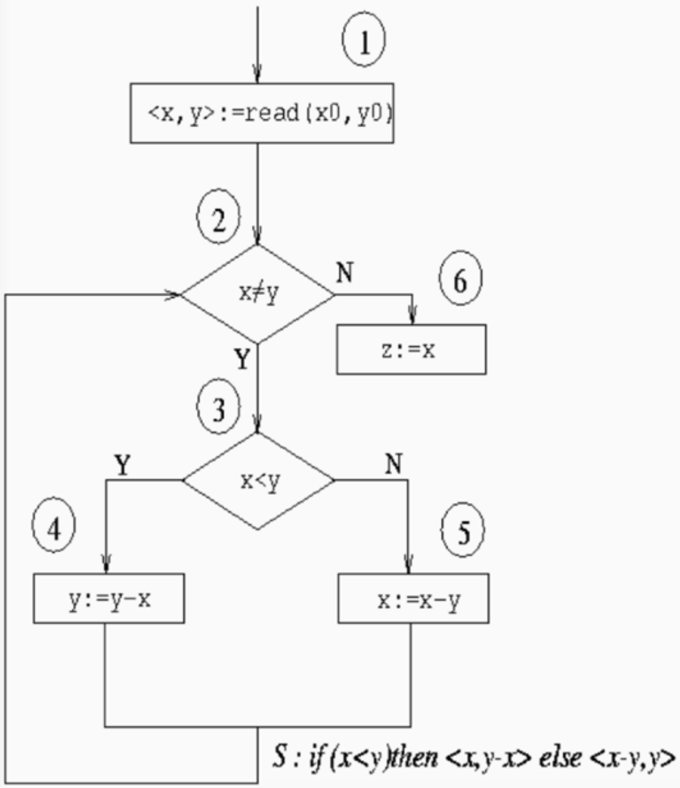
# White-box testing strategies

- Branch testing
- Path testing
  - boundary-interior method
  - domain testing
  - computational equivalence of paths
  - simple loop patterns

# White-box testing strategies

- Data flow testing
  - definition-use chains
- Mutation testing
  - Text anomalies

# Branch testing




**(!) Each predicate „true" and „false”**

***while w. "2":***

***(2,3), (2,6)***

***if w. "3":***

***(3,4), (3,5)***

# Path testing

*(!) Incorrect control flow implies incorrect results*

*(!) Paths can exercise control flow systematically*

- Program model:
  - Control flow graph: **G(a,n,s,e)**
  - Input variables: $x=<x_1, x_2, \ldots, x_n>$
  - Program (input) domain: $D=X_1 \times X_2 \times \ldots \times X_n$
  - Program path: $p=(n_0, n_1, \ldots, n_k)$
  - Path condition: $p(x)$
  - Path domain: $d(p)=\{ x \mid p(x)\}$
  - Path computation: $c(p): d(p) \rightarrow R$

# Path testing

- Strategies:
  - boundary-interior method
  - domain testing
  - computational equivalence of paths
  - simple loop patterns

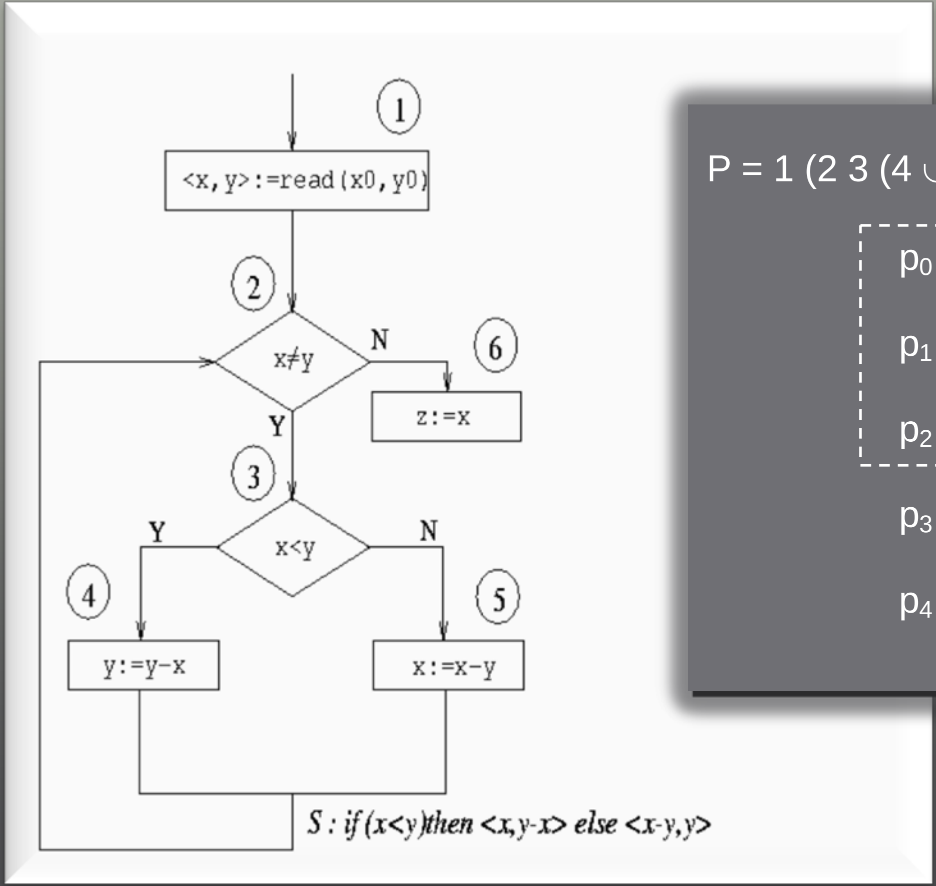
# Boundary-interior method

*(!) Problem with loops*

- Intuitive criterion:
  - Each loop ZERO and non-zero number of iterations,
  - Each loop MAX number of iterations

→ *Similar but more demanding then branch testing*

# Example




P = 1 (2 3 (4 ∪ 5))* 2 6

$p_0$ = 1 2 6,

$p_1$ = 1 2 3 4 2 6

$p_2$ = 1 2 3 5 2 6

$p_3$ = 1 2 3 4 2 3 5 2 6

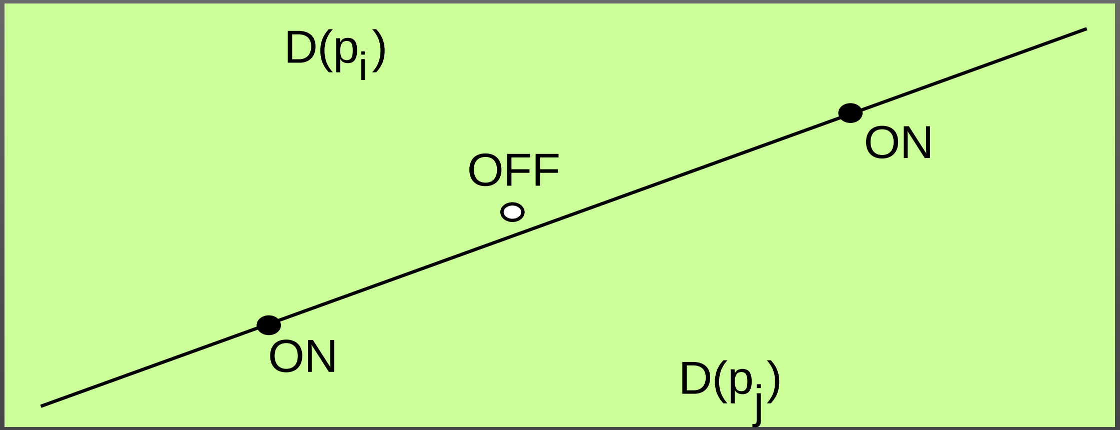
$p_4$ = 1 2 3 5 2 3 4 2 6

...

# Domain testing

*(!) Looking for domain errors*

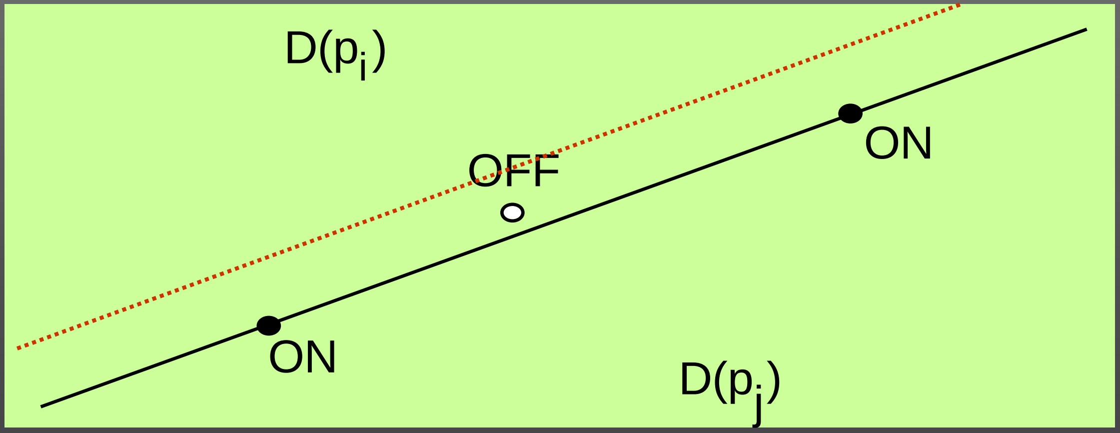
- Assumptions:
  - Predicates p(x) are linear functions on X,
  - Path computations c(p) are different,
  - No coincidental correctness.



# Domain testing

*(!) Looking for domain errors*

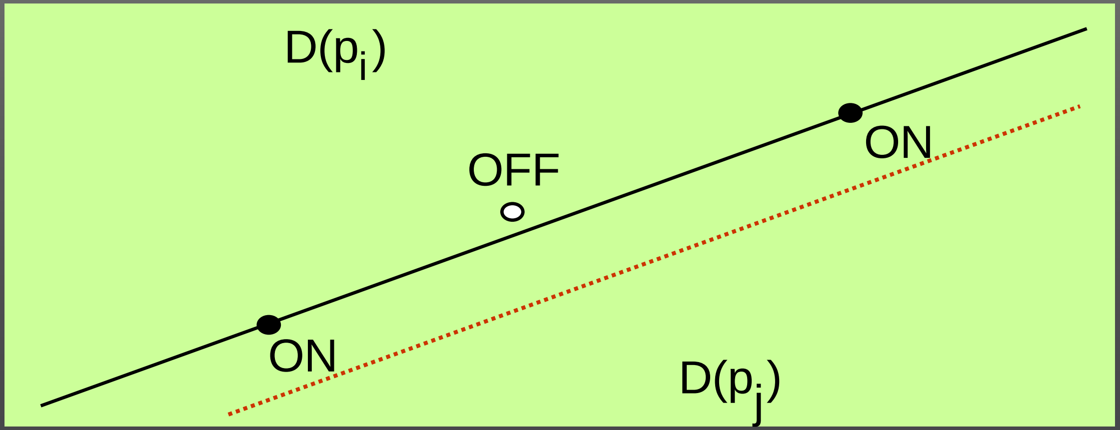
- Assumptions:
  - Predicates p(x) are linear functions on X,
  - Path computations c(p) are different,
  - No coincidental correctness.



# Domain testing

*(!) Looking for domain errors*

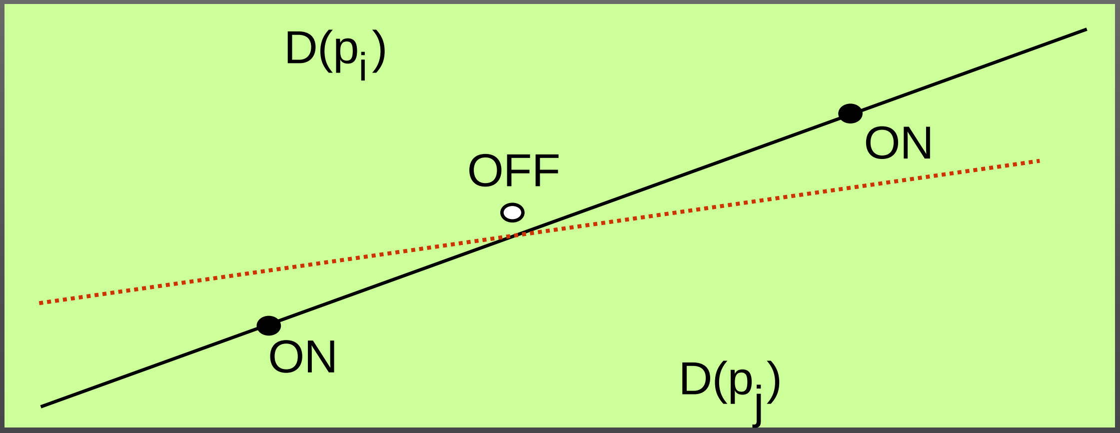
- Assumptions:
  - Predicates p(x) are linear functions on X,
  - Path computations c(p) are different,
  - No coincidental correctness.



# Domain testing

***(!) Looking for domain errors***

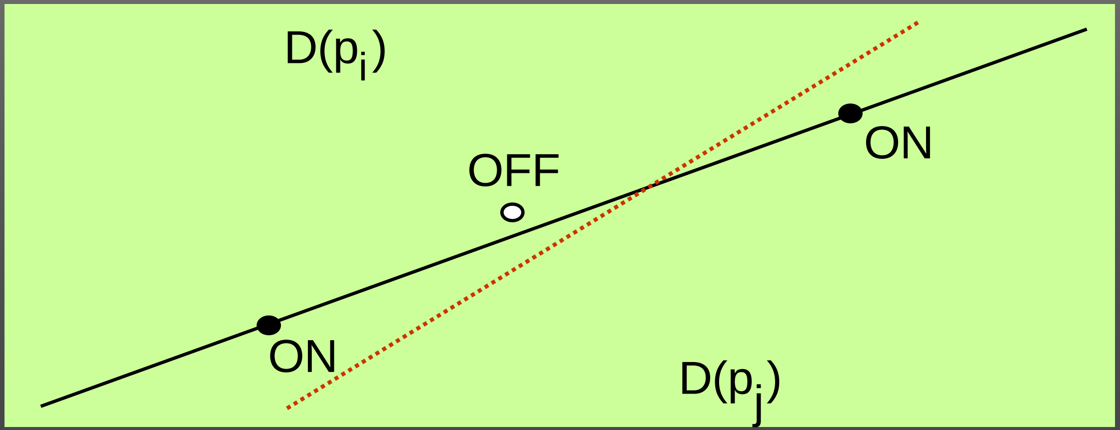
- Assumptions:
  - Predicates p(<u>x</u>) are linear functions on X,
  - Path computations c(p) are different,
  - No coincidental correctness.



# Domain testing

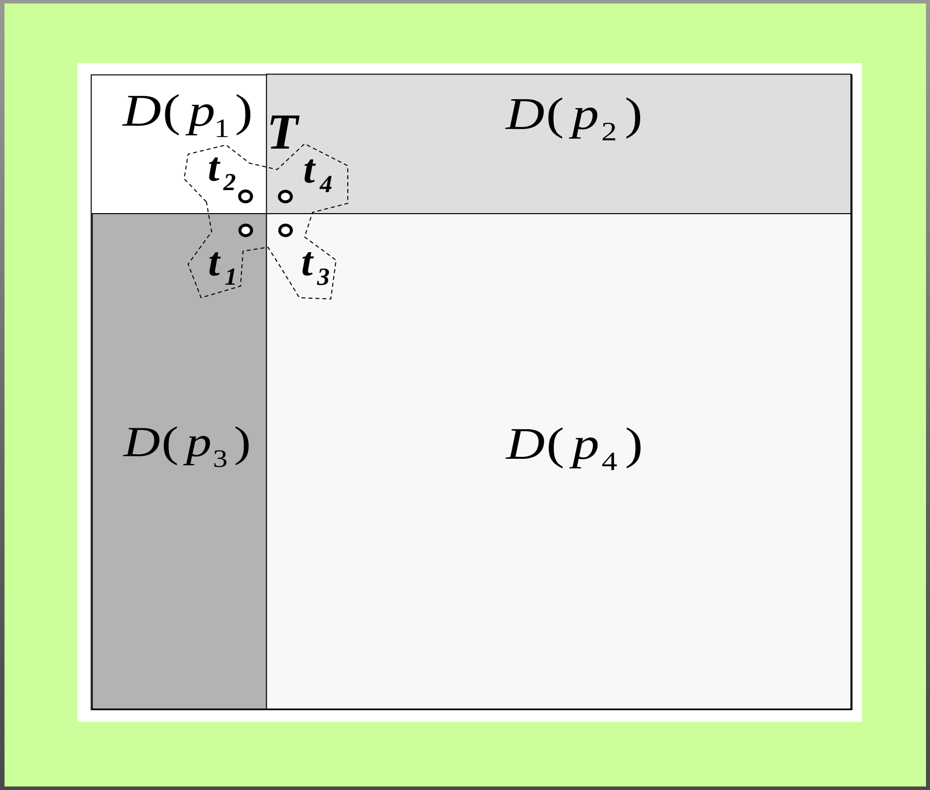
*(!) Looking for domain errors*

- Assumptions:
  - Predicates p(x) are linear functions on X,
  - Path computations c(p) are different,
  - No coincidental correctness.



# Example

- Domain testing



# Computational equivalence of paths

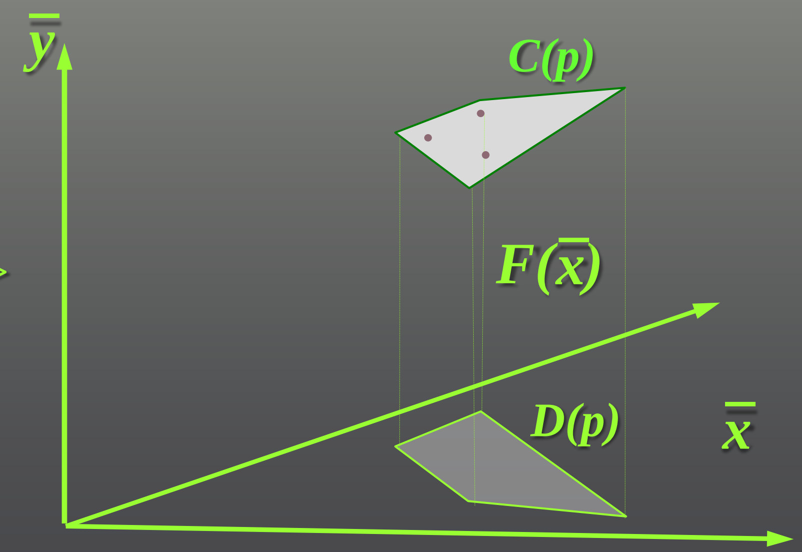
- Input data: $x=\langle x_1, x_2, \ldots, x_n\rangle$
- Input domain: $D=X_1\times X_2\times\ldots\times X_n$
- Output variables: $y=\langle y_1, y_2, \ldots, y_m\rangle$
- Path computation space: ***linear, (n+m)-dimensional***

**Test case:**

**path p**

**(n+m)-vector** $t=\langle d_1, d_2, \ldots, d_n, r_1, r_2, \ldots, r_m\rangle$

**n+m vectors** $\{t_1, t_2, \ldots, t_{n+m}\}$



# Example

- Path computation testing:
  - path

$p_1$:    1-2-7-8-10-11-12-13

  - computation

$$C(p_1) : < len_0, opt_0, item, e, pool[10], len_0, opt_0 > \rightarrow < len_0, opt_0, item, e, pool[10], len_0 - 1, opt_0 + 1 >$$

  - hyperplane

$$\begin{bmatrix} opt \\ len \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} opt_0 \\ len_0 \end{bmatrix} + \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

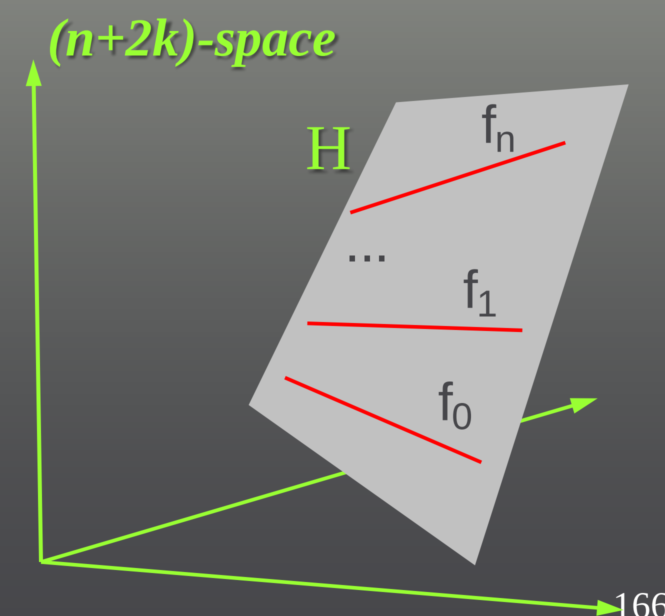
# Simple loop patterns

- Input variables: $x=\langle x_1, x_2, \ldots, x_n\rangle$
- Input domain: $D=X_1\times X_2\times\ldots\times X_n$
- Program variables: $z=\langle z_1, z_2, \ldots, z_k\rangle$
- Program computation space: (n+2k)-dimensional

$f_0 = g_{sv}g_{ve},\ \ f_1 = g_{sv}hg_{ve},\ \ \ldots,\ f_n = g_{sv}h^n g_{ve},\ \ldots$

$$f_n = \begin{cases} f_0 & n=0 \\ f_{n-1}H & n>0 \end{cases}$$

$$H = (g_{ve})^{-1}h(g_{ve})$$




- test completion criterion:

$S = 1 + \lceil (2k-1)/n \rceil$ paths

# Example

```
            int asynBCD(int number,int count){
1: 2        char symbol;
2: 3        for(;;)
3: 4        {for(;;)
4: 5        {receive(symbol);
5: 6 7      if((symbol==SPACE)||(symbol==STOP))
6: 11       break;
7: 8        count++;
8: 9 10     if(count>9)
9:          return ERROR;
10: 3       }
11: 12      number=10*number+count;
12: 13 14   if(symbol==STOP)
13:         return (number);
14: 2       }
15:         }
```

***Input variables***:

**`symbol, number, count`**

→ ***n=3***

***Program variables:***

**`number, count`**

→ ***k=2***

# Data flow

- Simple chain:

  <definition, use>

  d: x = f(5)+3y

  ...

  u: z = 2x-y

- Use chain:

  <all-definition, use>

  d: x = f(5)+3y

  ...

  u: z = 2x-y

  ...

  a: w = x*z

- Live chain:

  <all-definition, all-use>

  d: x = f(5)+3y

  ...

  u: z = 2x-y

  ...

  a: w = x*z

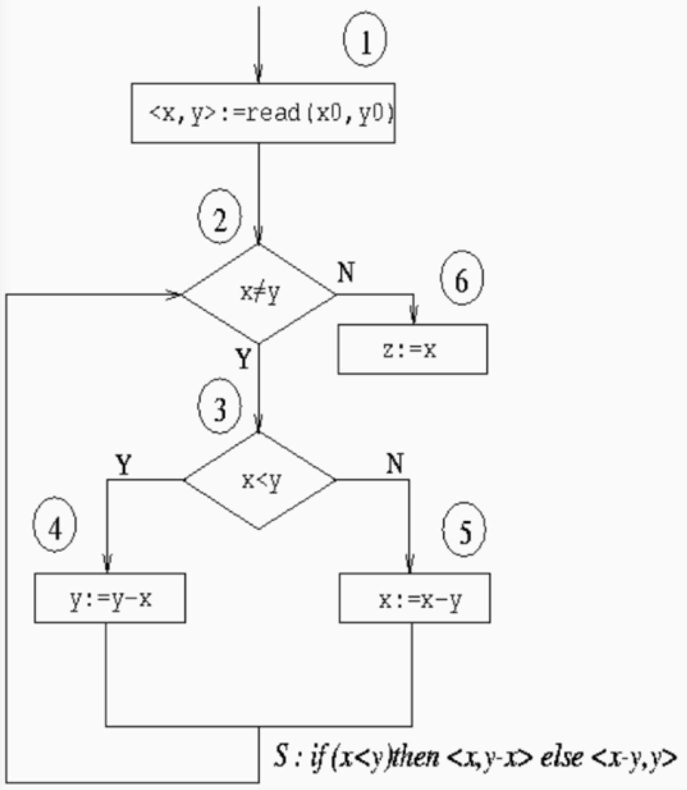
  ...

  b: y = x+w

*criterion*

→ *Exercise each chain*

# Examples




***"d-u" chains (simple):***

**<1,3>, <4,2>, ...**

***"ad-u" chains (use):***

**<1,4,3>, <4,5,3>, ...**

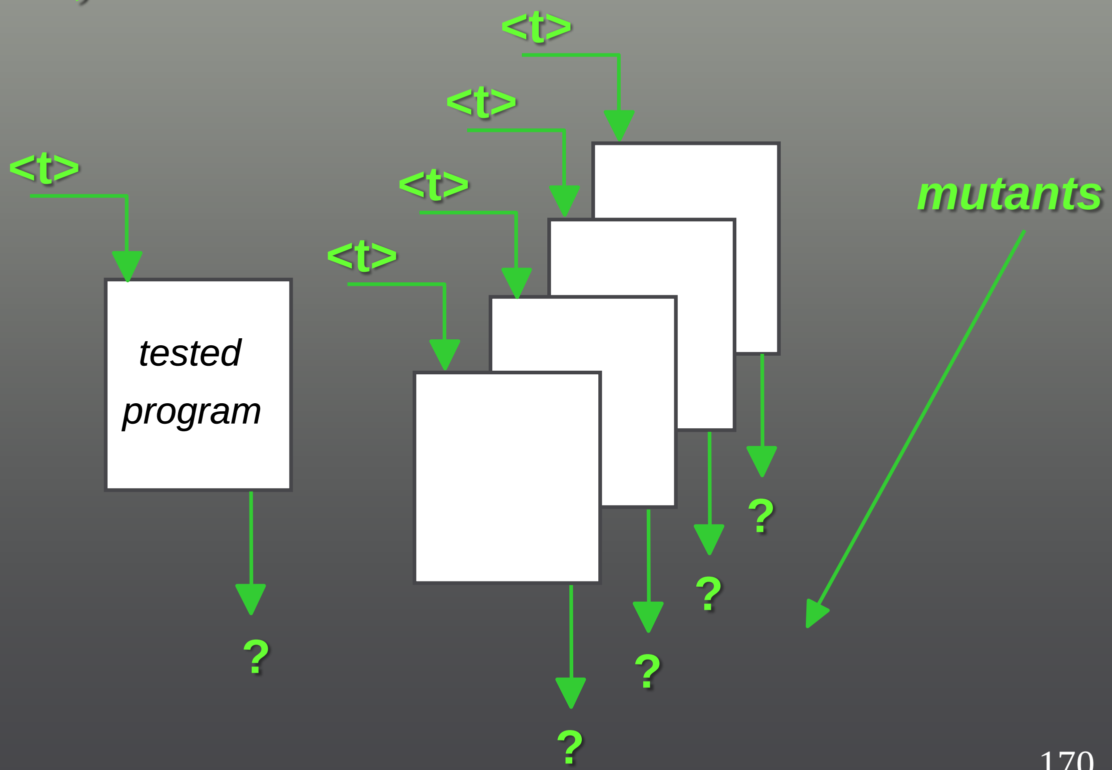
***"ad-au" chains (live):***

**<1,4,6>, <1,5,6>, ...**

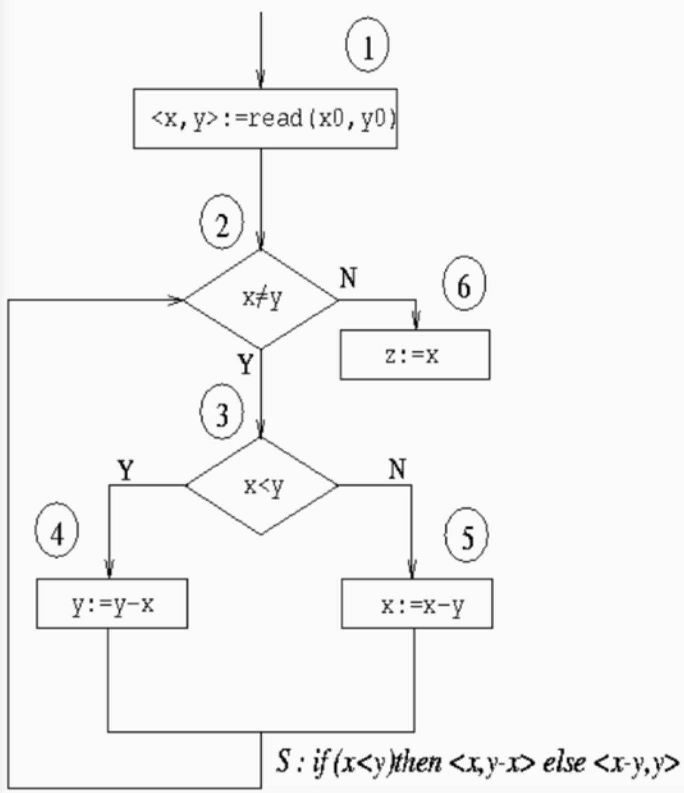
# Mutation testing

***(!) errors result from occasional "typos" in the program text***

→ test harness, Monte-Carlo



# Example




$M_1 \rightarrow$ *while (x=y) ...*
$M_2 \rightarrow$ *if (x=y) ...*

| <x,y> | P | $M_1$ | $M_2$ |
|---|---|---|---|
| <9,3> | 3 | 9 | 3 |
| <9,6> | 3 | - | ↑ |

# Structure of test cases

- Input data
- Expected results
- Environment settings
- Scenario context

# Test script

```
/* TeSS 1 */
{
 /* request the master to go first (1) */
 <
  (before 0 22 [])
 >
 /* reach the voting configuration by slaves (2) */
 <
  (before 1 26 [print stid; print ntid;])
  (before 2 26 [print stid; print ntid;])
  (before 3 26 [print stid; print ntid;])
 >
 /* reach the reporting configuration by slaves (3) */
 <
  (before 1 41 [print data;])
  (before 2 41 [print data;])
  (before 3 41 [print data;])
 >
}
```

Logging the state

# Test script

```
/* TeSS 2 */
{
 /* request the master to go first (1) */
 /* spoil v_size of slave #3 before voting */
 <
  (before 0 22 [])(after 3 23 [set v_size=0;])
 >
  /* reach the voting configuration by slaves (2) */
 <
  (before 1 26 [])
  (before 2 26 [])
  (before 3 26 [])
 >
 /* reach the reporting configuration by slaves (3) */
 <
  (before 1 41 [print data;])
  (before 2 41 [])
  (before 3 41 [])
 >
 /* make slave #1 winning the race */
 <
  (after 0 22 [])(after 1 41 [])
 >
}
```

Value enforcement

# Attributes of test cases

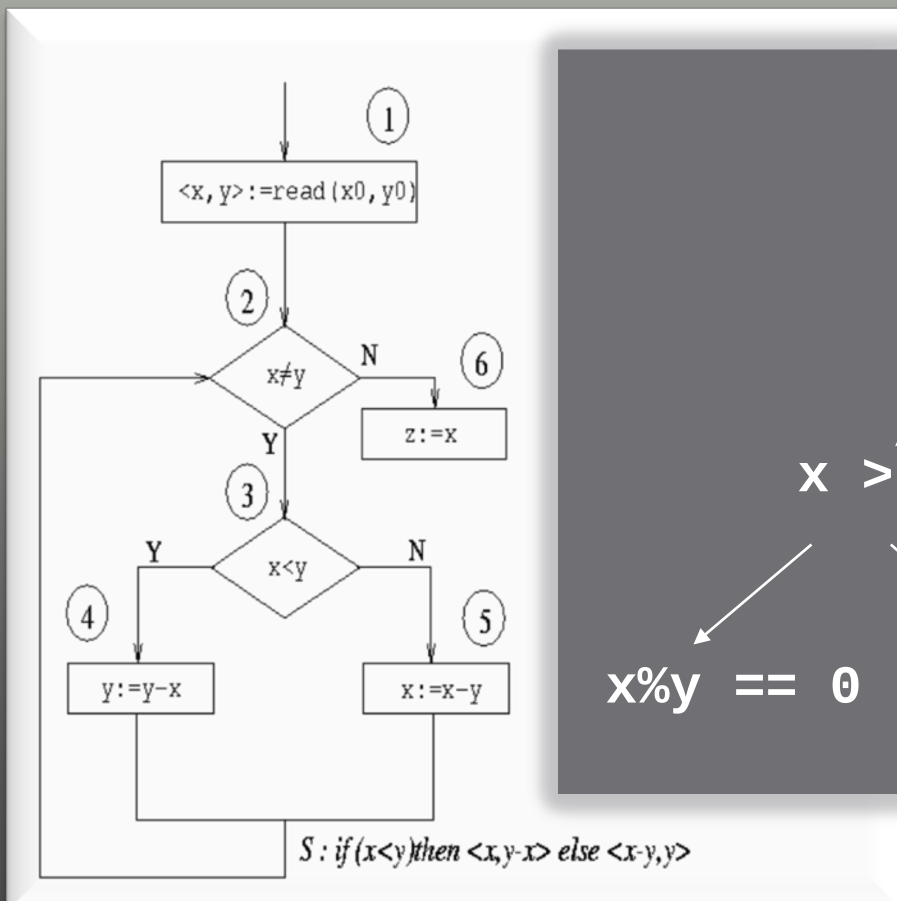
- Representativeness
  - A single case represents a subset
- Feasibility
  - Non-empty set of input data exists, eg. path condition is satisfied
- Observability
  - Deterministic automaton
- Reproducibility
  - All input data identified (path condition interpretation)
  - Timing conditions under tester’s control

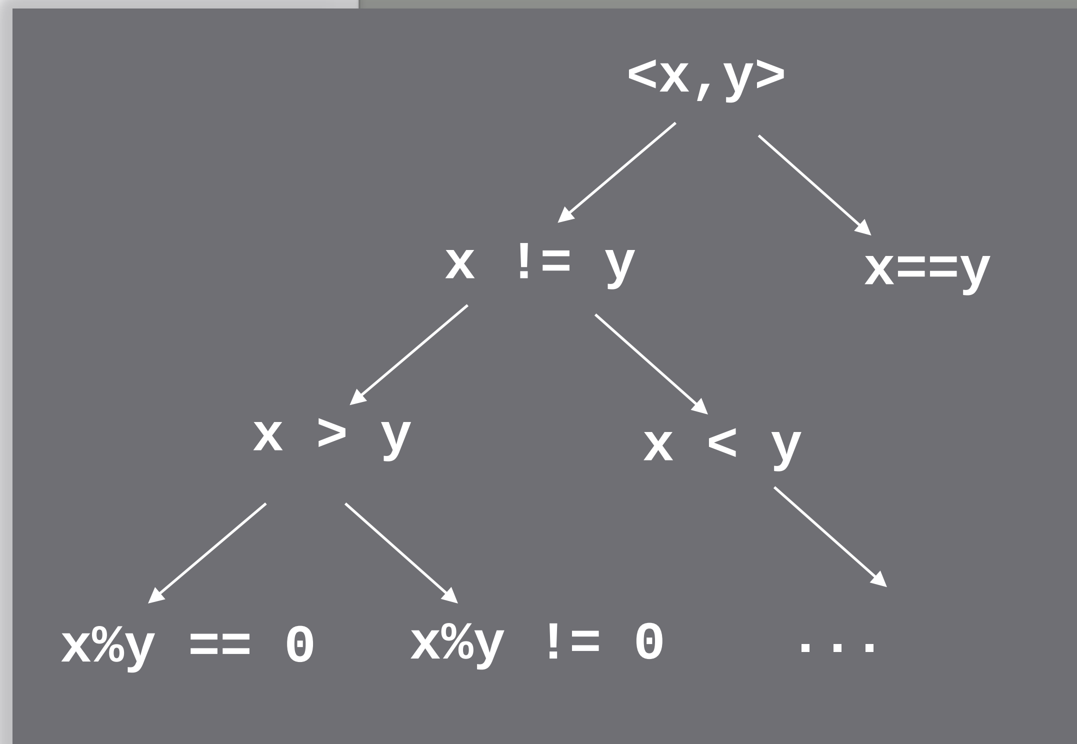
# Feasibility

```
                int asynBCD(int number,int count){
1: 2            char symbol;
2: 3            for(;;)
3: 4            {for(;;)
4: 5            {receive(symbol);
5: 6 7          if((symbol==SPACE)||(symbol==STOP))
6: 11           break;
7: 8            count++;
8: 9 10         if(count>9)
9:              return ERROR;
10: 3           }
11: 12          number=10*number+count;
12: 13 14       if(symbol==STOP)
13:             return (number);
14: 2           }
15:             }
```

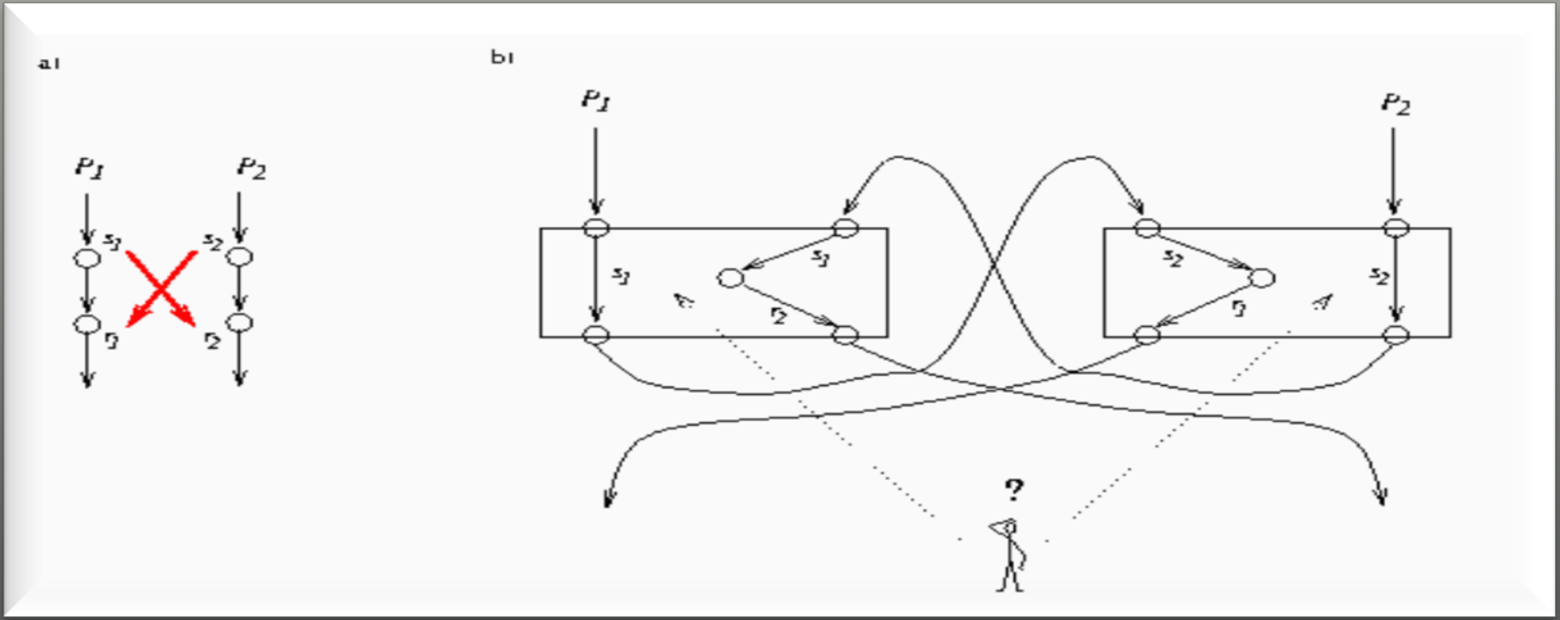
!? $p = 1\ 2\ (\ 3\ 4\ 5\ 7\ 8\ 10\ )^{10}\ 3\ 4\ 5\ 7\ 8\ 9$

# Representativeness



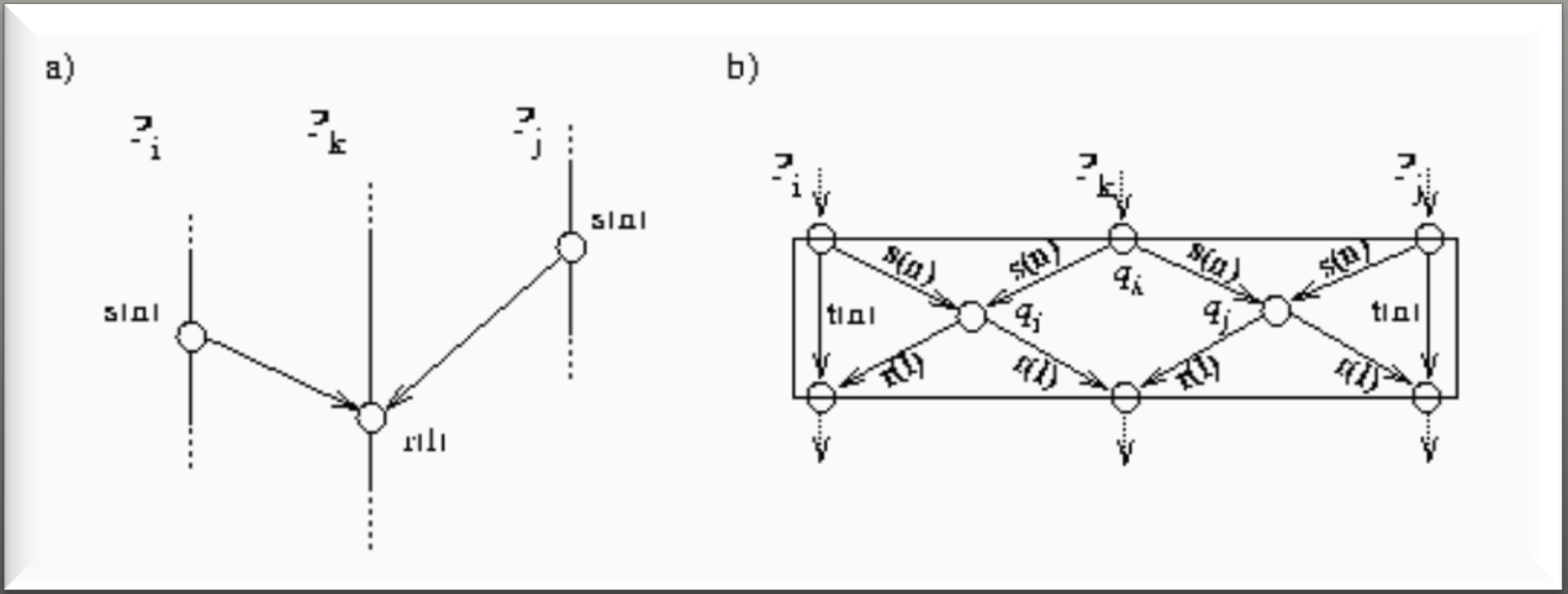



# Observability




→ *Testing error*

# Reproducibility

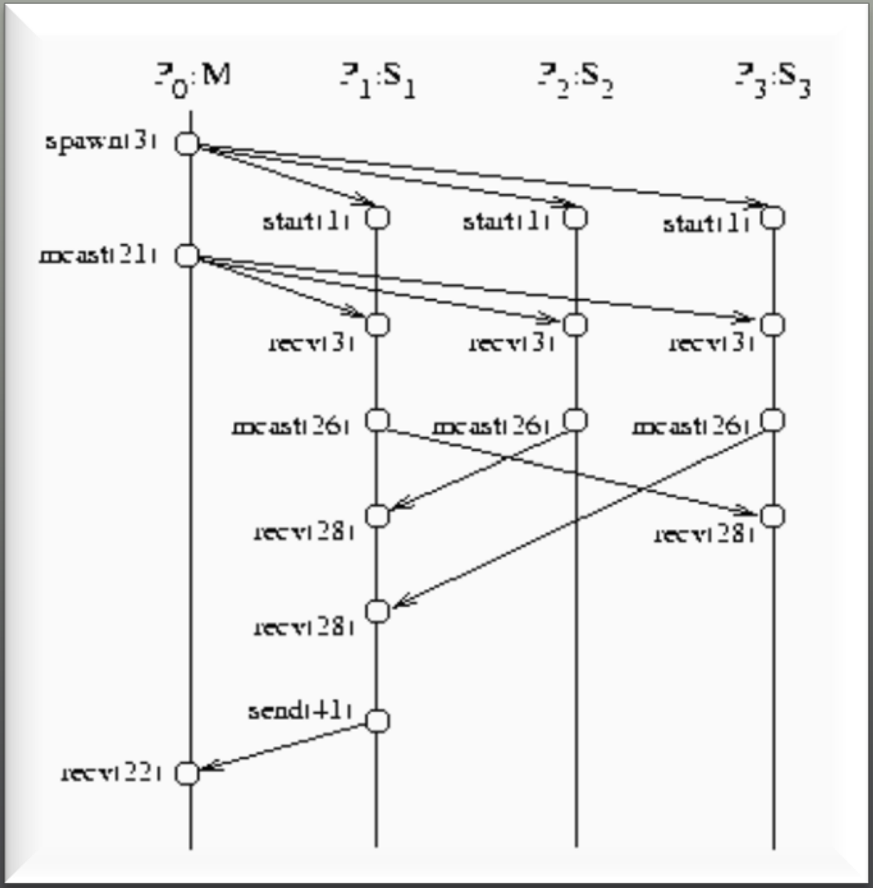


# Logging results

- Checkpoint
  - static (“compiled in”)
  - dynamic (breakpoint)
- Log
  - centralized
  - distributed
- Result analysis
  - on-line (state or event detection),
  - off-line (% of test coverage, error localization)
  - replay (visualization, state recovery)

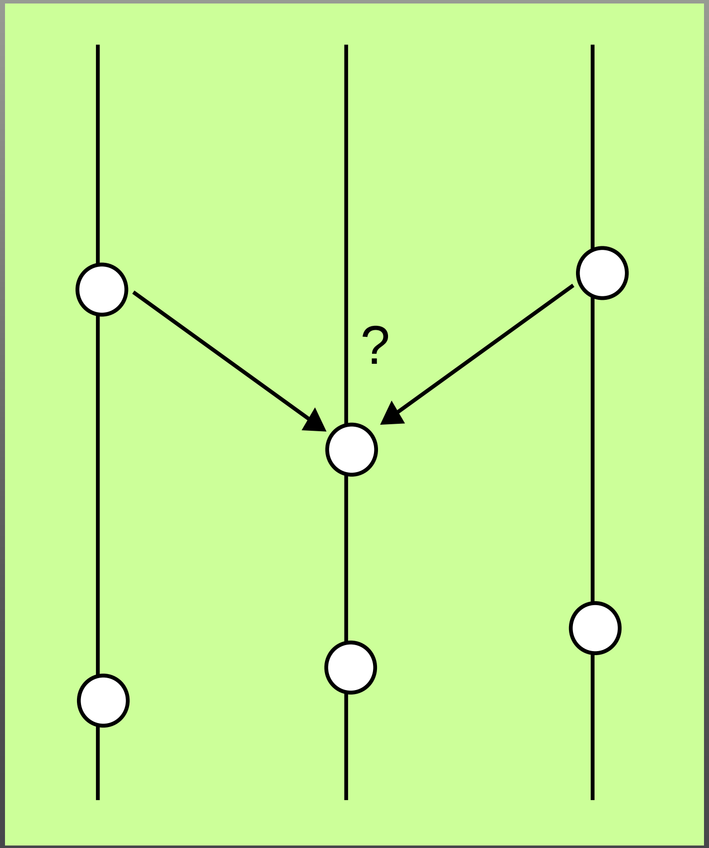
# Test scenario execution mode

- random
- supervised
- deterministic



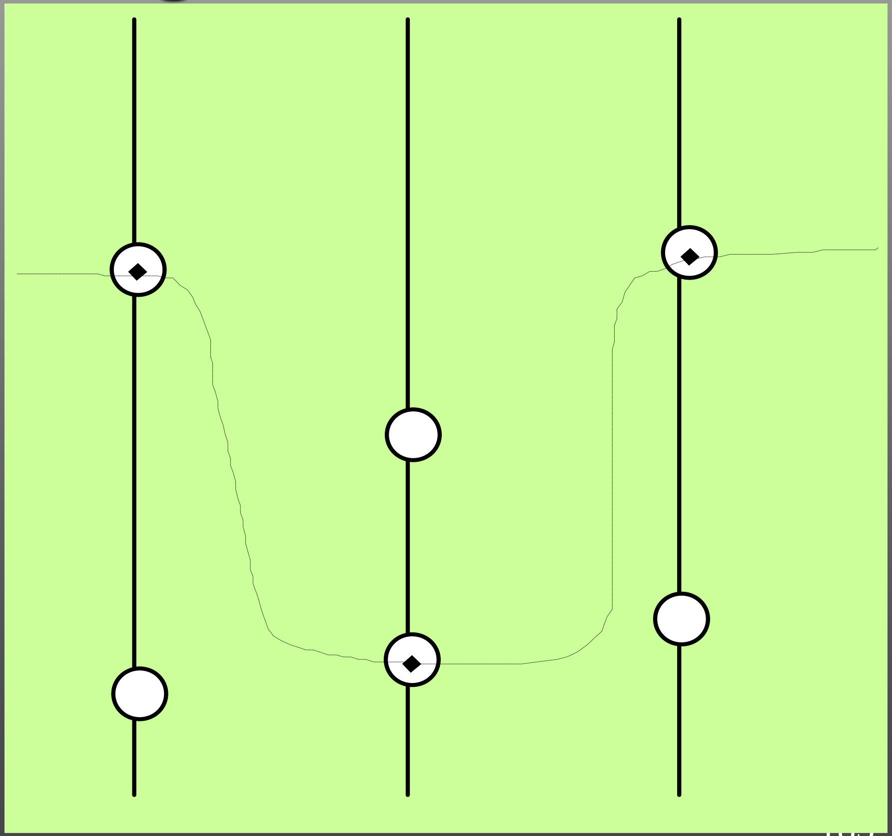
# Test scenario types

- One-thread-One-time (OtOt):
  race detection



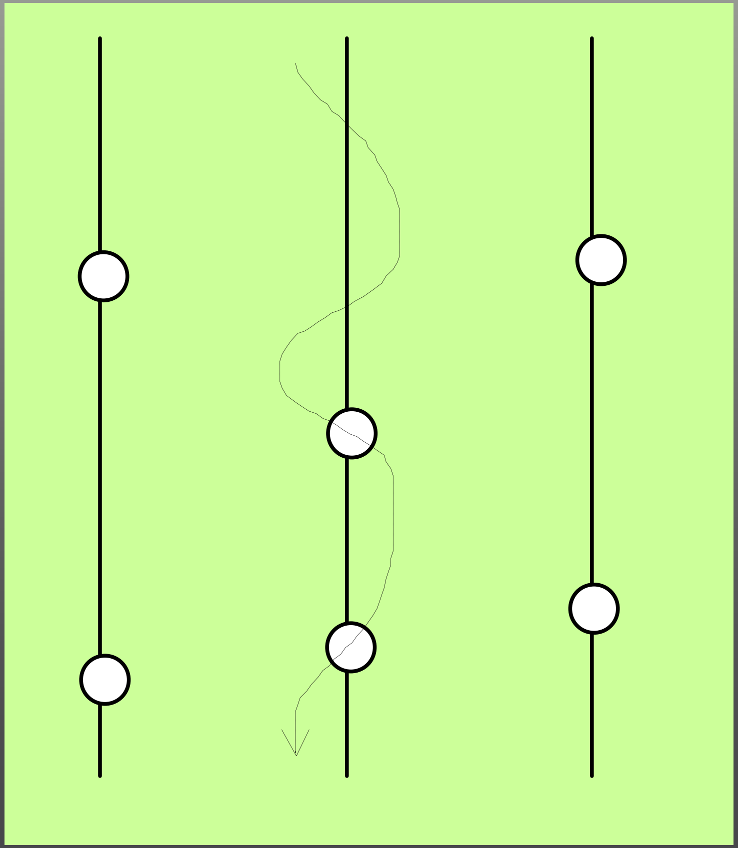
# Test scenario types

- Many-threads-One-time (MtOt):
  global state monitoring



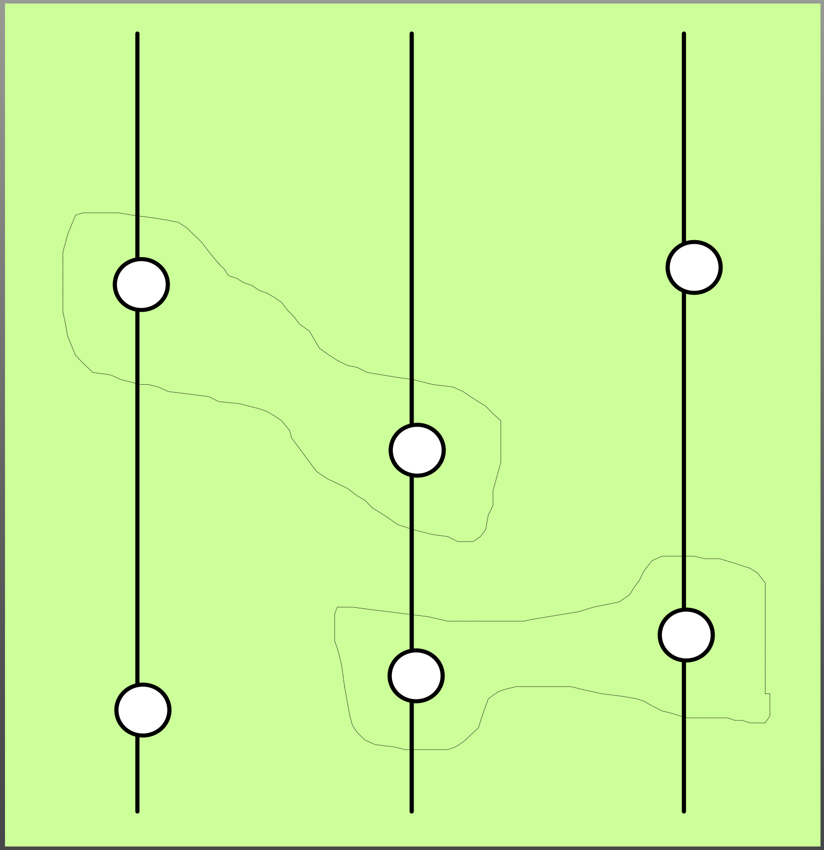
# Test scenario types

- One-thread-Many-times (OtMt):
  single process path testing

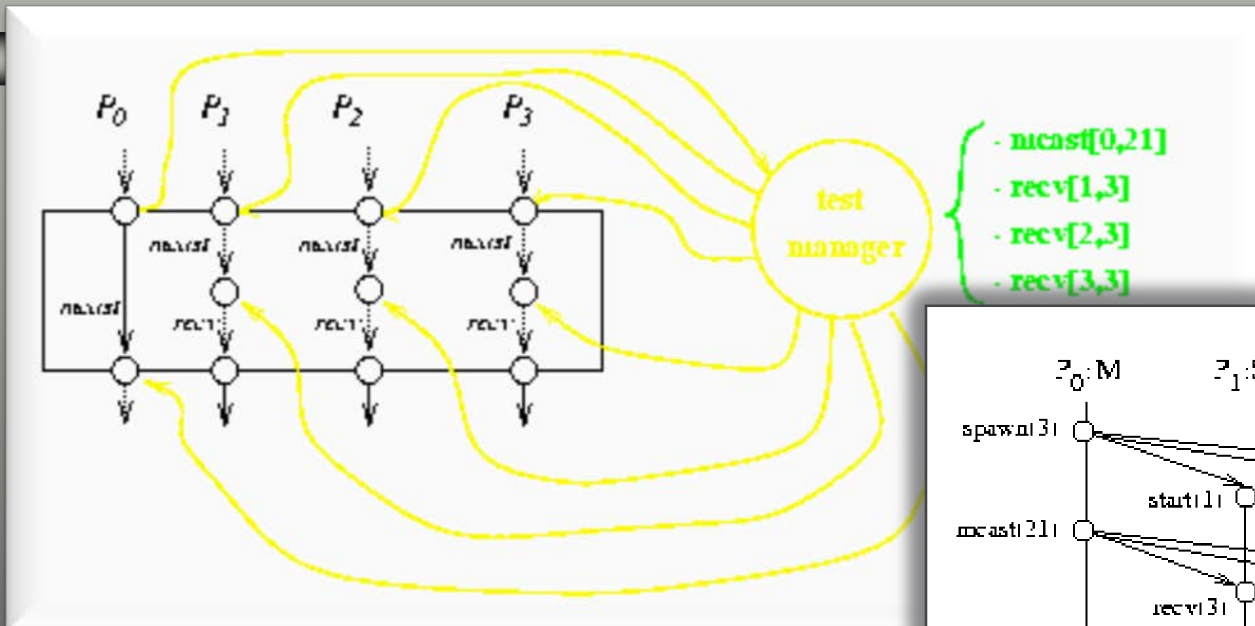


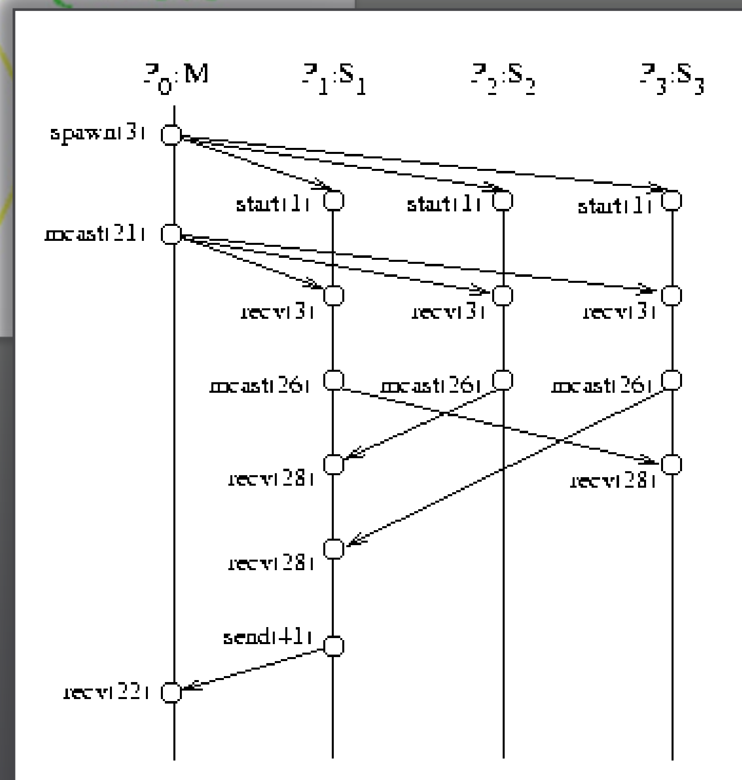
# Test scenario types

- Many-threads-Many-times (MtMt):
  - event monitoring

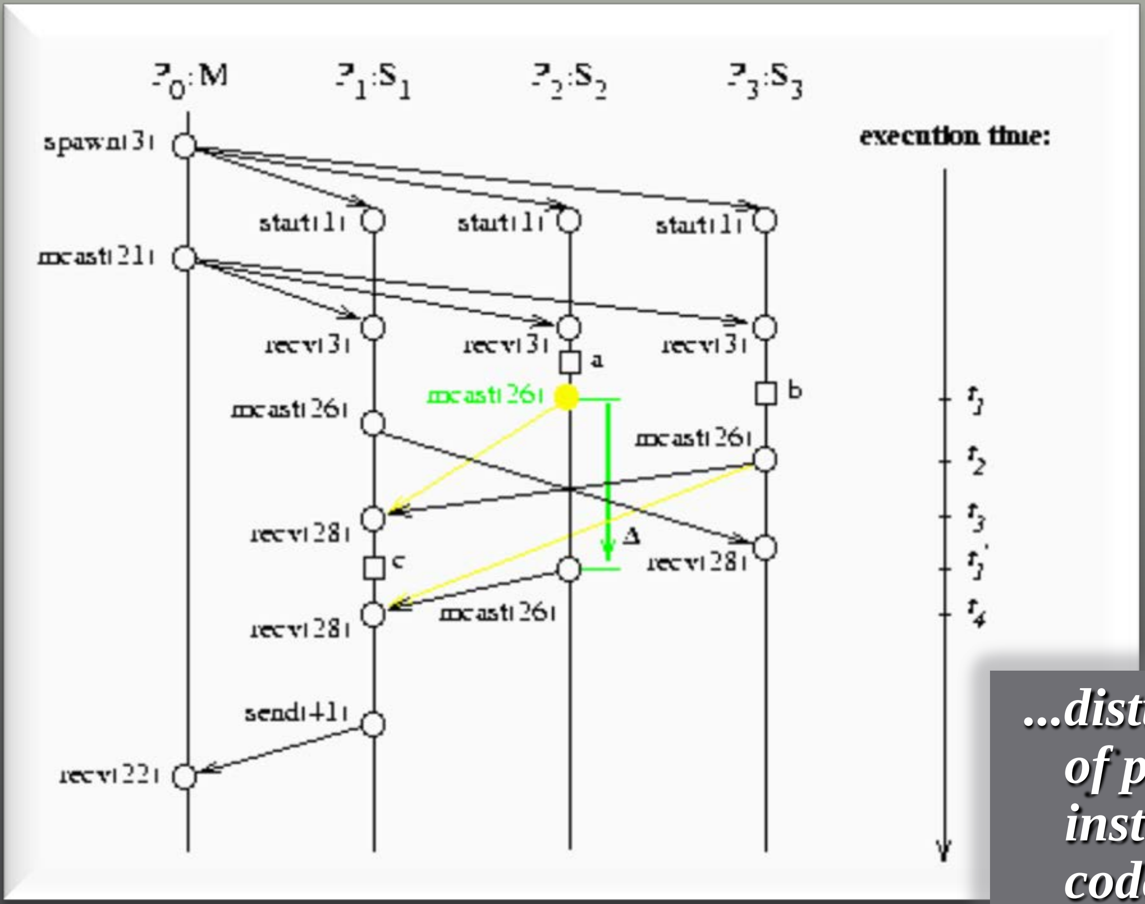


# Instrumenting code and environment



# Probe effect




***...disturbing internal timing of processes by instrumenting the system code***

# Log structure

- Heading:
  - Unique identifier
  - Comment
  - Records (table of content)
- Record:
  - date, time, test case ID,
  - event, local state, context
- Event:
  - Statement executed, signal sent/received, exception raised, variable value changed
- State:
  - Object memory content
- Context:
  - history, condition, global state

# What is worth logging?

- Potential error occurrences:
  - Arithmetic instructions (function calls, assignments),
  - Predicate (condition) evaluation,
  - Type conversion, actual vs formal parameters,
  - Return statements,
  - Dynamic variables,
  - Systems diagnostics, exception handlers,
  - Message packing/unpacking,
  - Message tagging,
  - Races,
  - Communications actions matching

# Error localization

***(!) Knowing that the program has a bug doesn't mean knowing what causes it***

- Debugging:
  - Post-mortem print-out, core dump,
  - Trace file (log)
  - Building a hypothesis,
  - Elimination of hypotheses
- Tools:
  - Print-out
  - Breakpoint trap
  - Instant replay

# Is it possible to do without testing?

- Programmers make mistakes when, when creating a program, they are unable to remember all the details needed to make it correct
- There are no bug-free programs, they are only poorly tested
- Programs considered correct may still have errors
- We can mistake correct program behavior for a wrong one (and vice versa)
- Errors reveal throughout the entire life of a program