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 ciritical 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

© Bogdan Wiszniewski, 2024

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

© Bogdan Wiszniewski, 2024

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

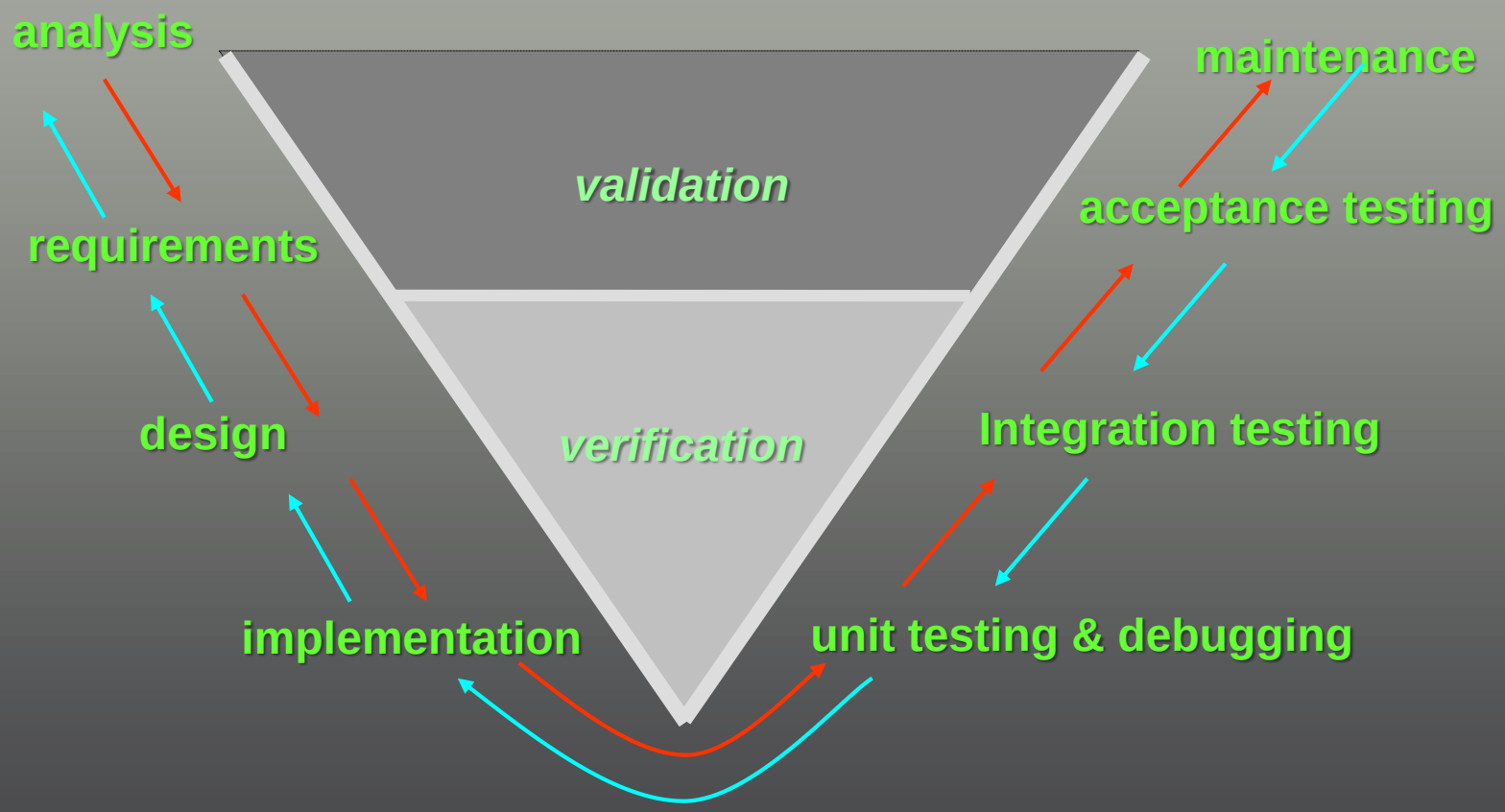
A. Product life-cycle vs. its testing cycle
B. VVT processes
C. Planning of VVT processes
D. Static analysis techniques
E. Error, program and runt-time environment models
F. Black-box (functional) testing
G. White-box (structural) testing

© Bogdan Wiszniewski, 2024

A. Product life-cycle vs. its testing cycle

© Bogdan Wiszniewski, 2024

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?***

© Bogdan Wiszniewski, 2024

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?*

© Bogdan Wiszniewski, 2024

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?*

© Bogdan Wiszniewski, 2024

Systematic approach

© Bogdan Wiszniewski, 2024

Systematic approach

bugs

Systematic approach

run-time environment models

bugs

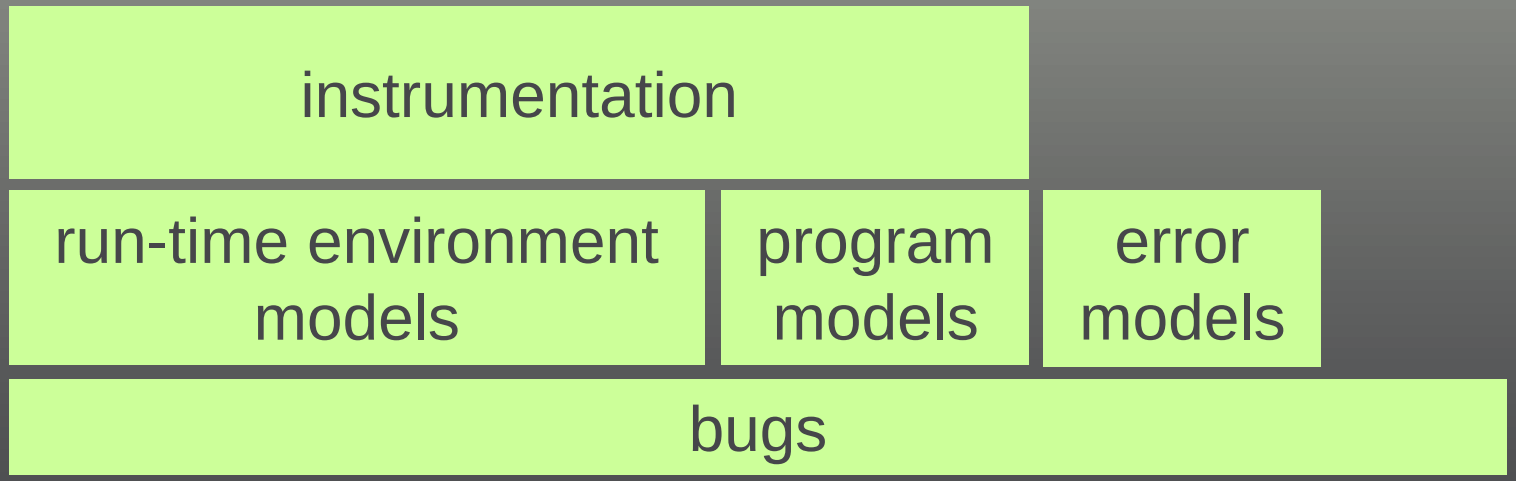
Systematic approach

run-time environment models	program models
bugs	

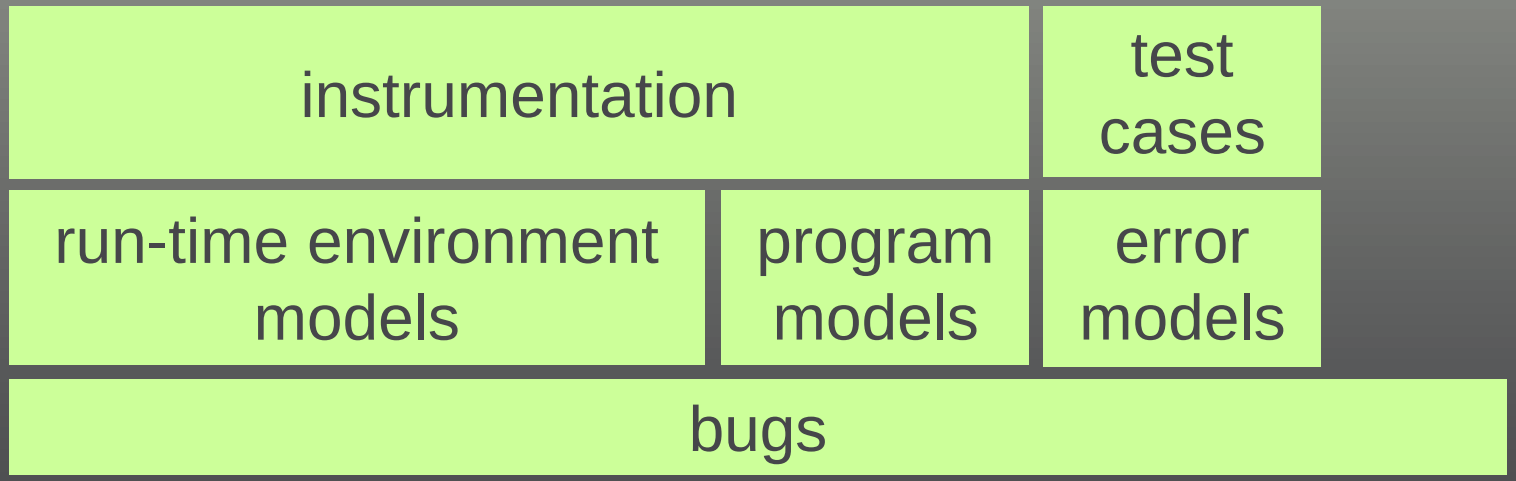
Systematic approach

run-time environment models	program models	error models
bugs		

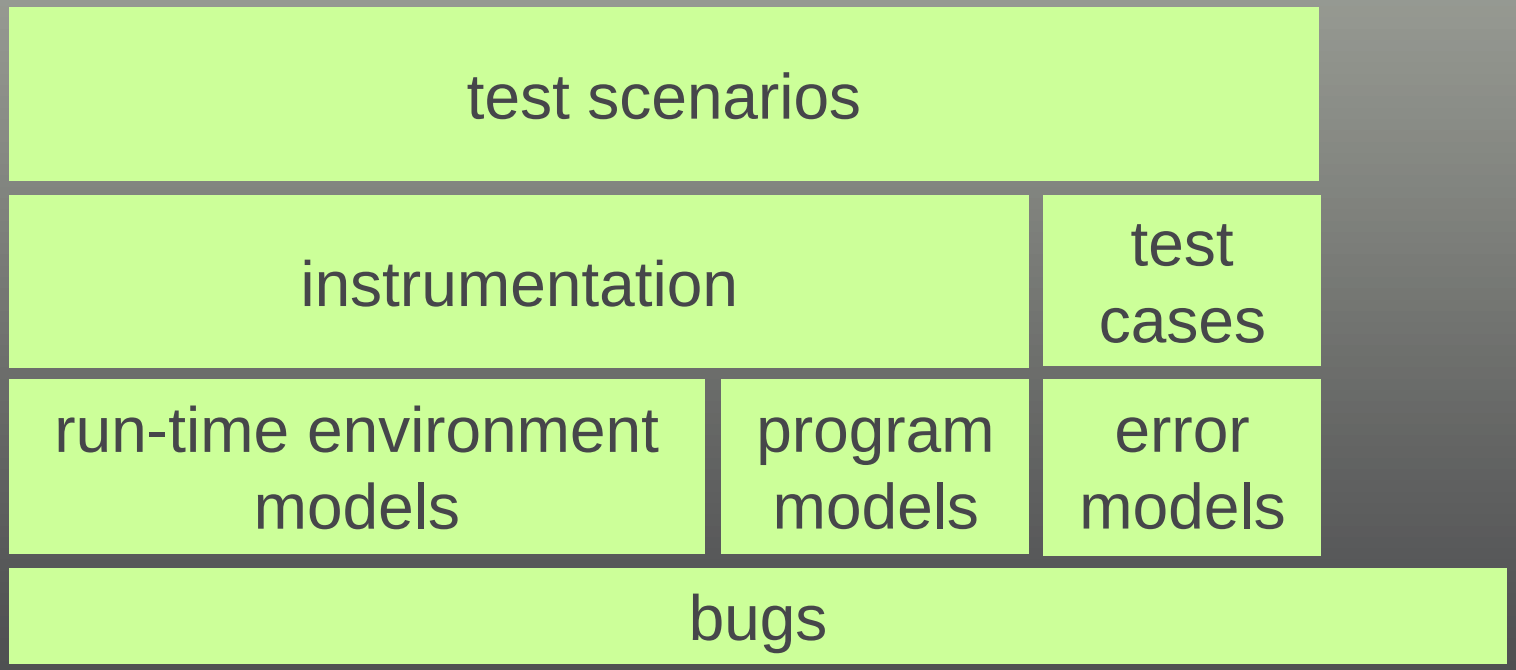
Systematic approach



Systematic approach

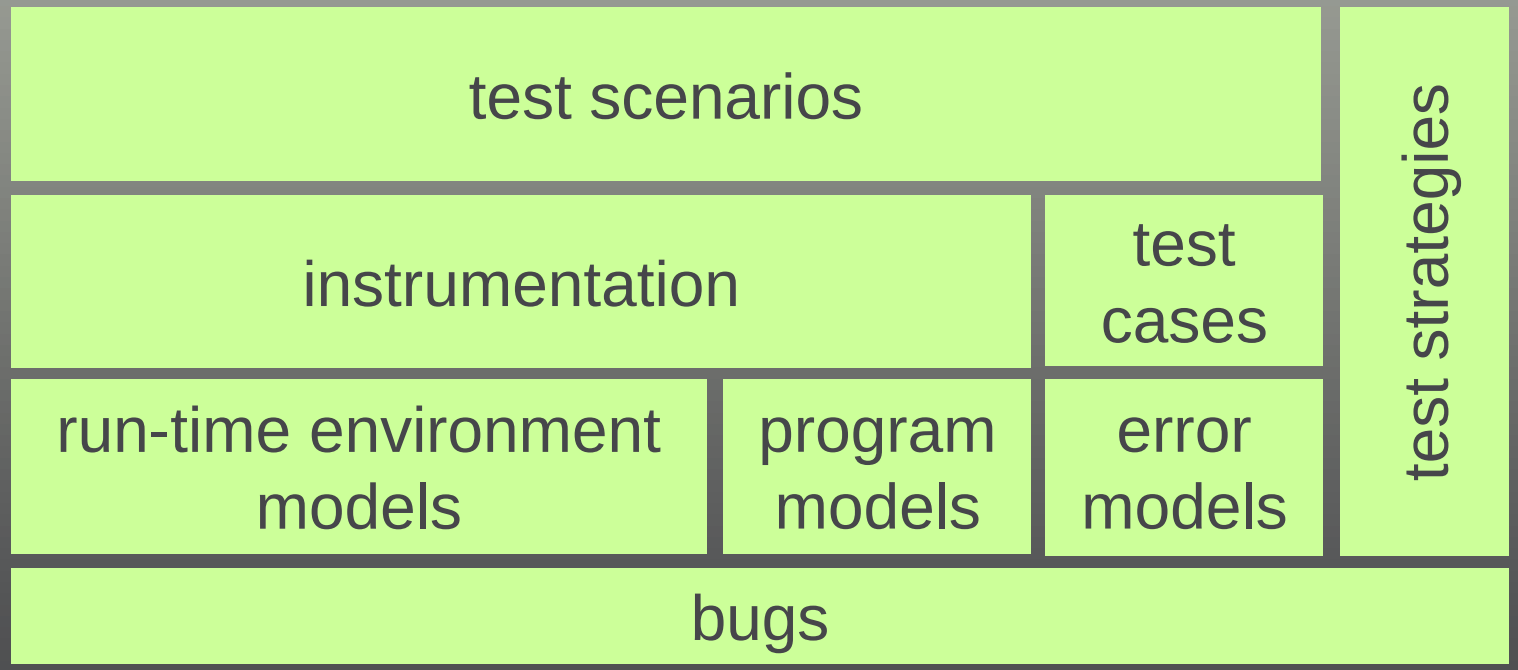


Systematic approach

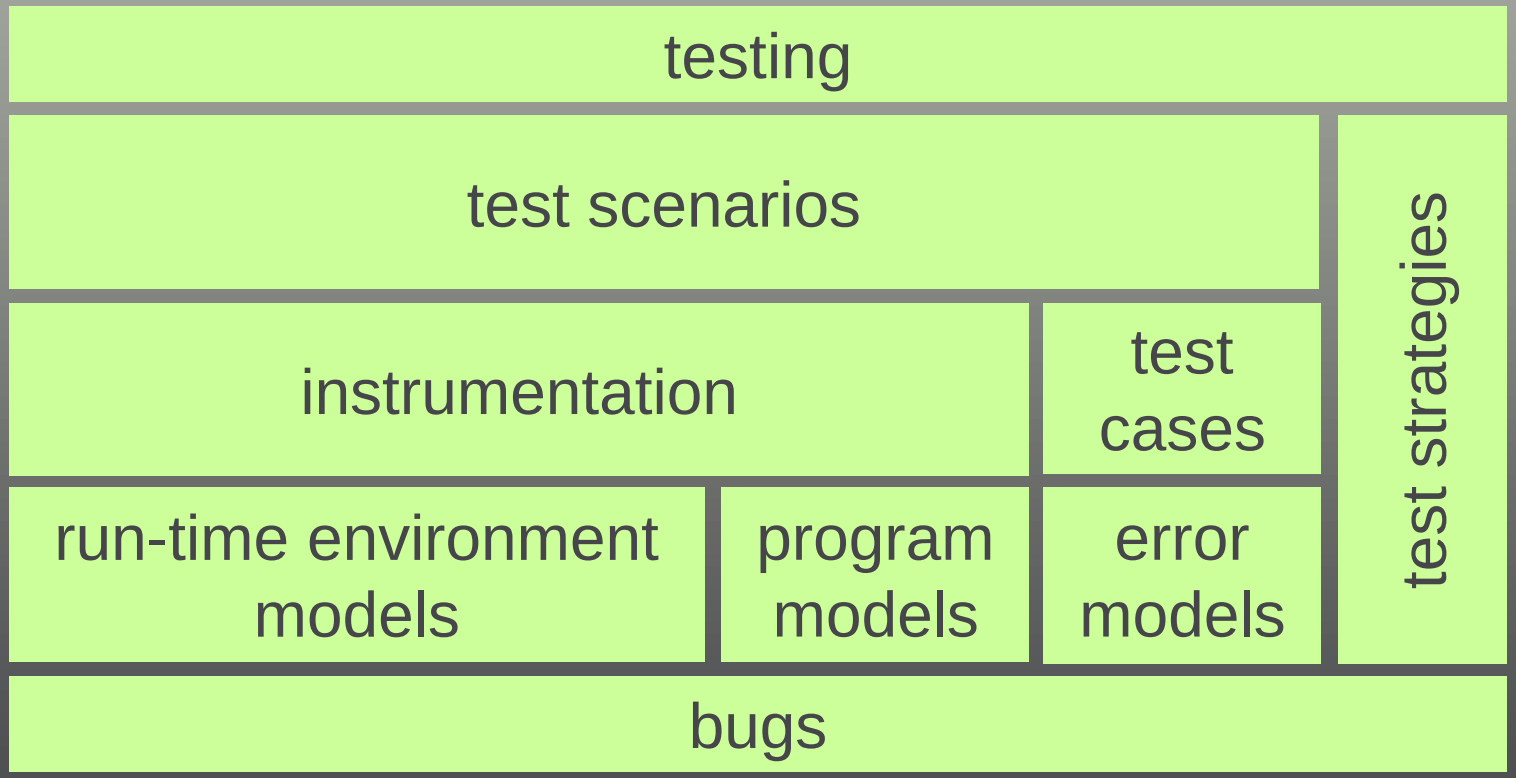


© Bogdan Wiszniewski, 2024

Systematic approach



Systematic approach



Dynamic analysis

- **Test case**

  A single element selected from an enumerable set of program behaviors

© Bogdan Wiszniewski, 2024

Dynamic analysis

- **Test completion criterion**

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

Dynamic analysis

- **Testing strategy**

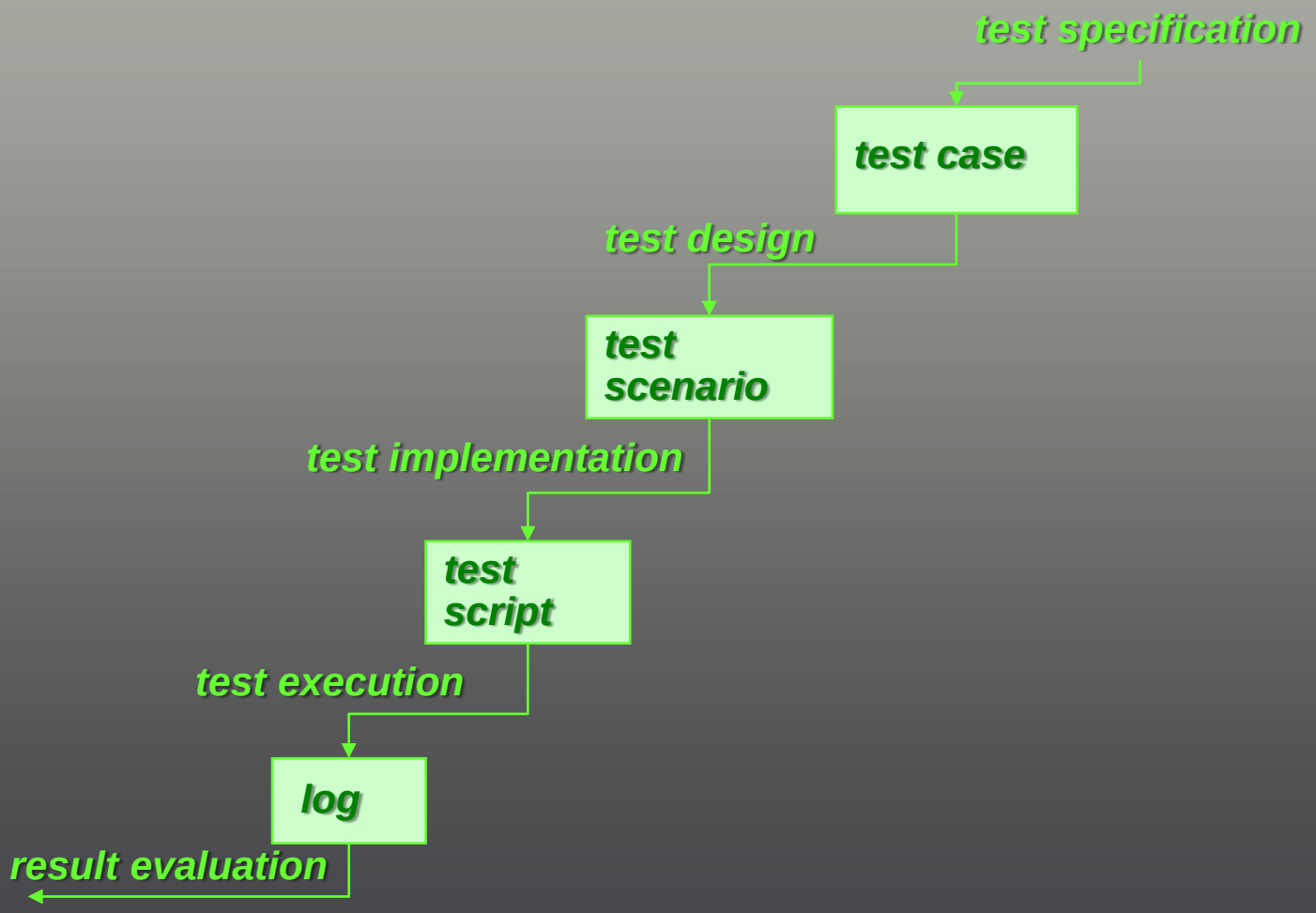
  A set of rules for selecting test cases to a (possibly finite) set according to some adopted criterion

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

© Bogdan Wiszniewski, 2024

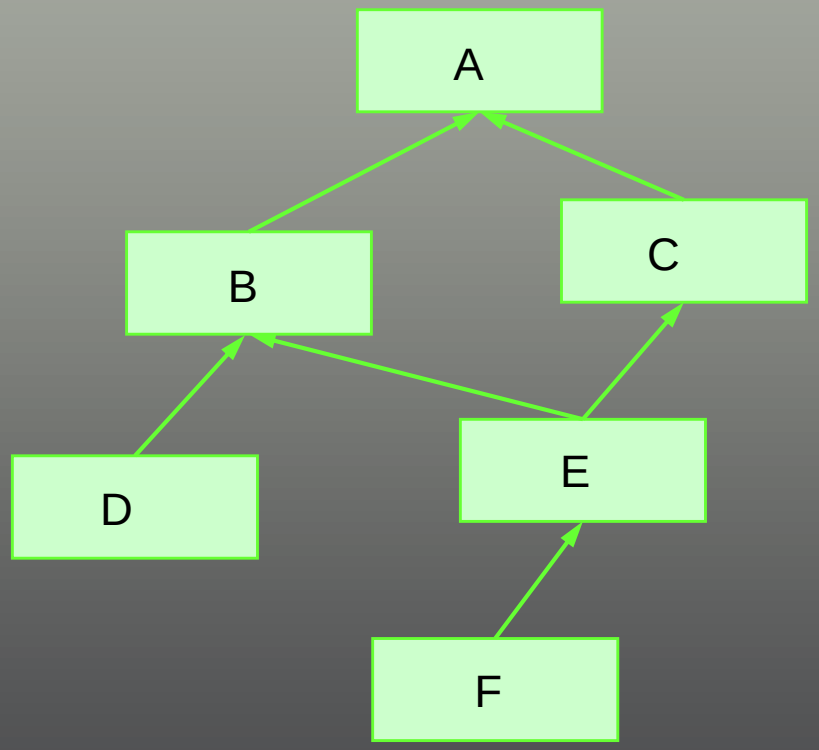
Unit testing

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

© Bogdan Wiszniewski, 2024

Integration testing

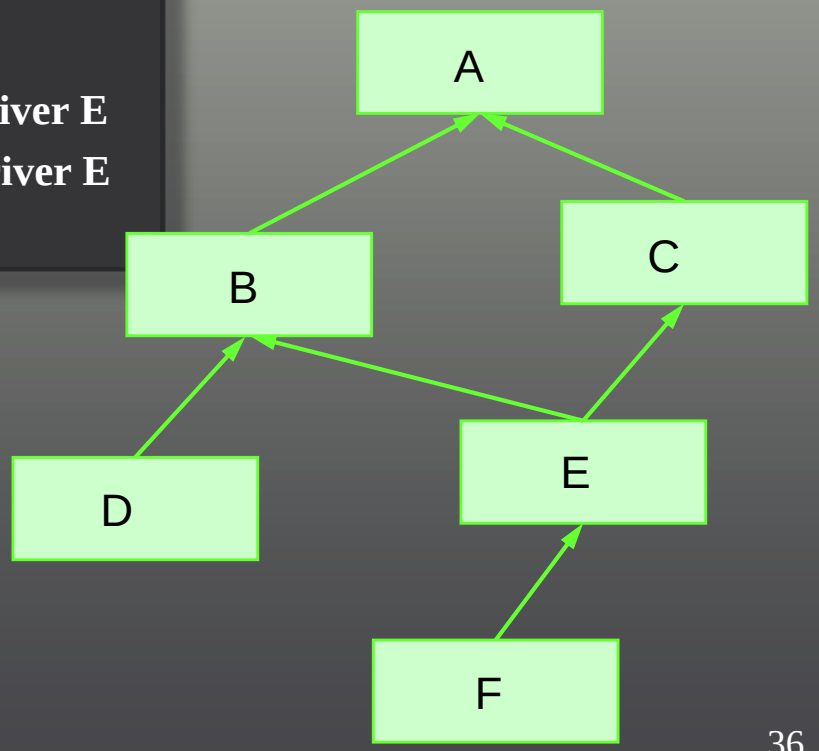
- **Strategies:**
  - **incremental**
  - **big-bang**



© Bogdan Wiszniewski, 2024

Bottom-up method

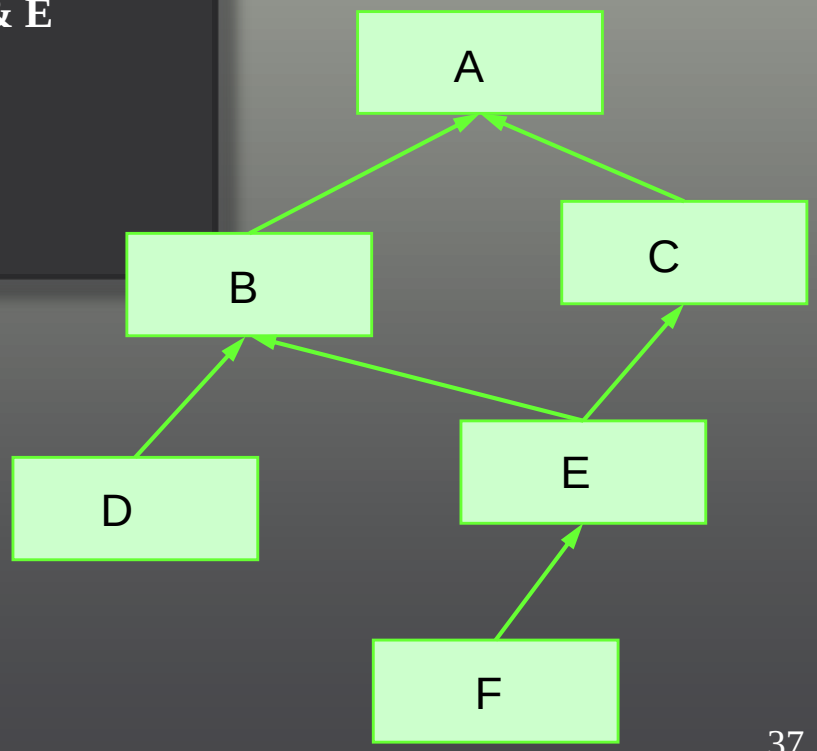
step	environment
1 unit D	driver D
2 unit F	driver F
3 connected units E+F	driver E
4 connected units D+E+F+B	driver B, driver E
5 connected units C+E+F	driver C, driver E
6 connected units A+B+C+D+E+F	--



© Bogdan Wiszniewski, 2024

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	--



© Bogdan Wiszniewski, 2024

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

© Bogdan Wiszniewski, 2024

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

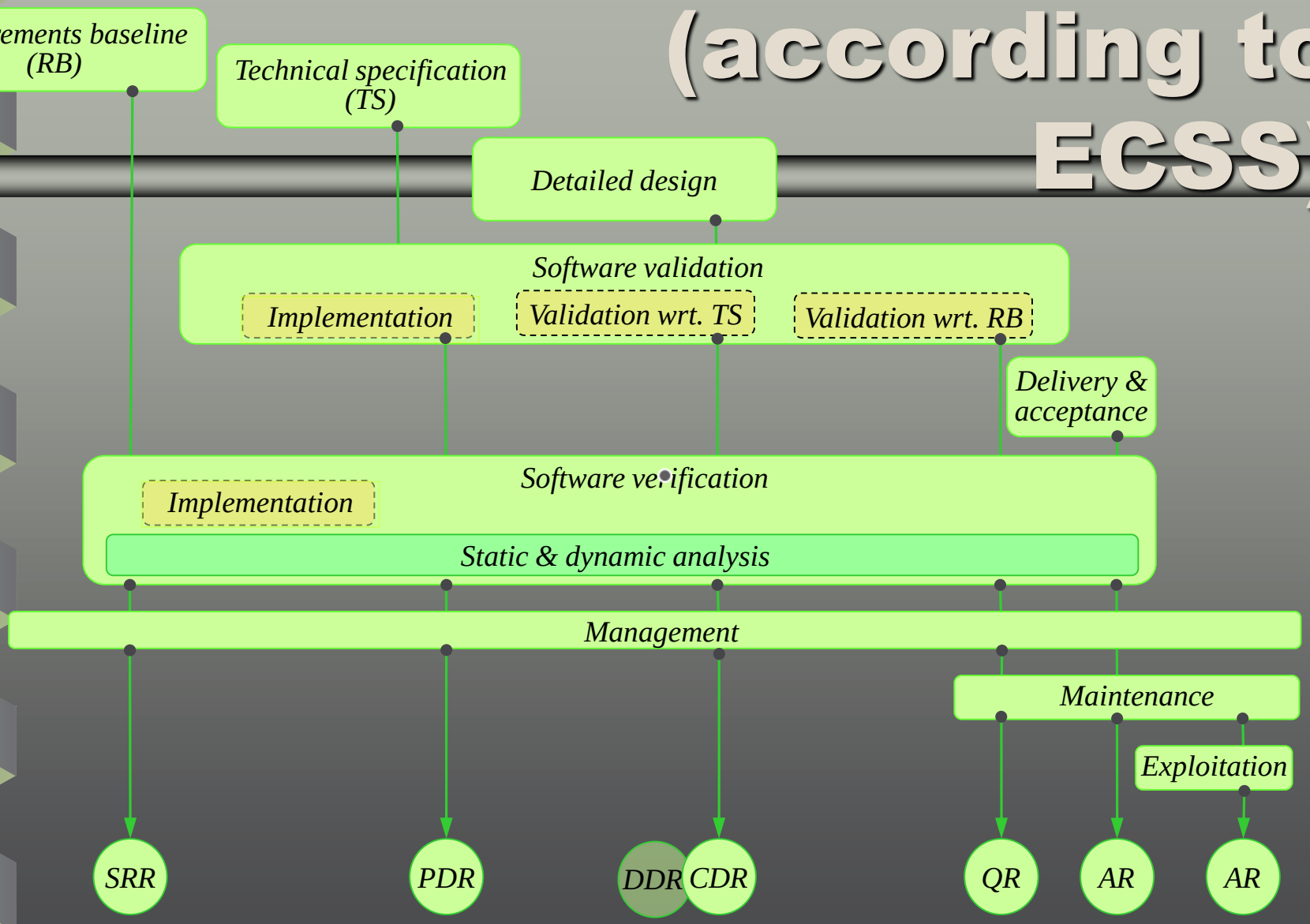
© Bogdan Wiszniewski, 2024

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

© Bogdan Wiszniewski, 2024

Life-cycle phases (according to ECSS)



© Bogdan Wiszniewski, 2024

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).

© Bogdan Wiszniewski, 2024

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**

© Bogdan Wiszniewski, 2024

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 ality 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

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.).

© Bogdan Wiszniewski, 2024

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

© Bogdan Wiszniewski, 2024

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)

© Bogdan Wiszniewski, 2024

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.

© Bogdan Wiszniewski, 2024

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

© Bogdan Wiszniewski, 2024

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

© Bogdan Wiszniewski, 2024

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

© Bogdan Wiszniewski, 2024

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

© Bogdan Wiszniewski, 2024

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

© Bogdan Wiszniewski, 2024

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

© Bogdan Wiszniewski, 2024

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')

© Bogdan Wiszniewski, 2024

C. Planning of VVT processes

© Bogdan Wiszniewski, 2024

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

© Bogdan Wiszniewski, 2024

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

© Bogdan Wiszniewski, 2024

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)**

© Bogdan Wiszniewski, 2024

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

© Bogdan Wiszniewski, 2024

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.**

© Bogdan Wiszniewski, 2024

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

© Bogdan Wiszniewski, 2024

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).**

© Bogdan Wiszniewski, 2024

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)*

© Bogdan Wiszniewski, 2024

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*

© Bogdan Wiszniewski, 2024

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

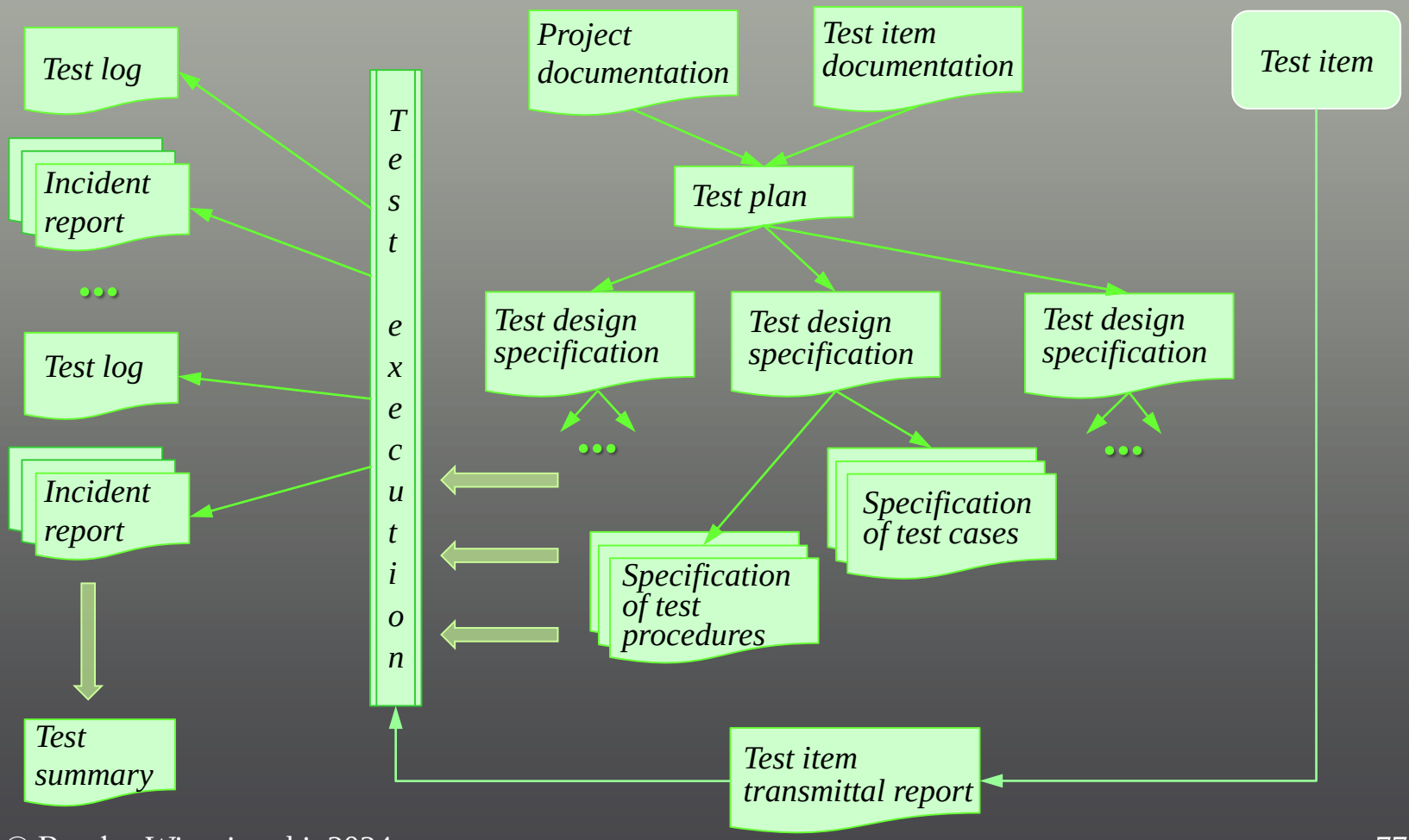
© Bogdan Wiszniewski, 2024

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

© Bogdan Wiszniewski, 2024

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

© Bogdan Wiszniewski, 2024

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

© Bogdan Wiszniewski, 2024

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

© Bogdan Wiszniewski, 2024

Test procedures

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

© Bogdan Wiszniewski, 2024

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

© Bogdan Wiszniewski, 2024

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.

© Bogdan Wiszniewski, 2024

D. Static analysis techniques

© Bogdan Wiszniewski, 2024

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”

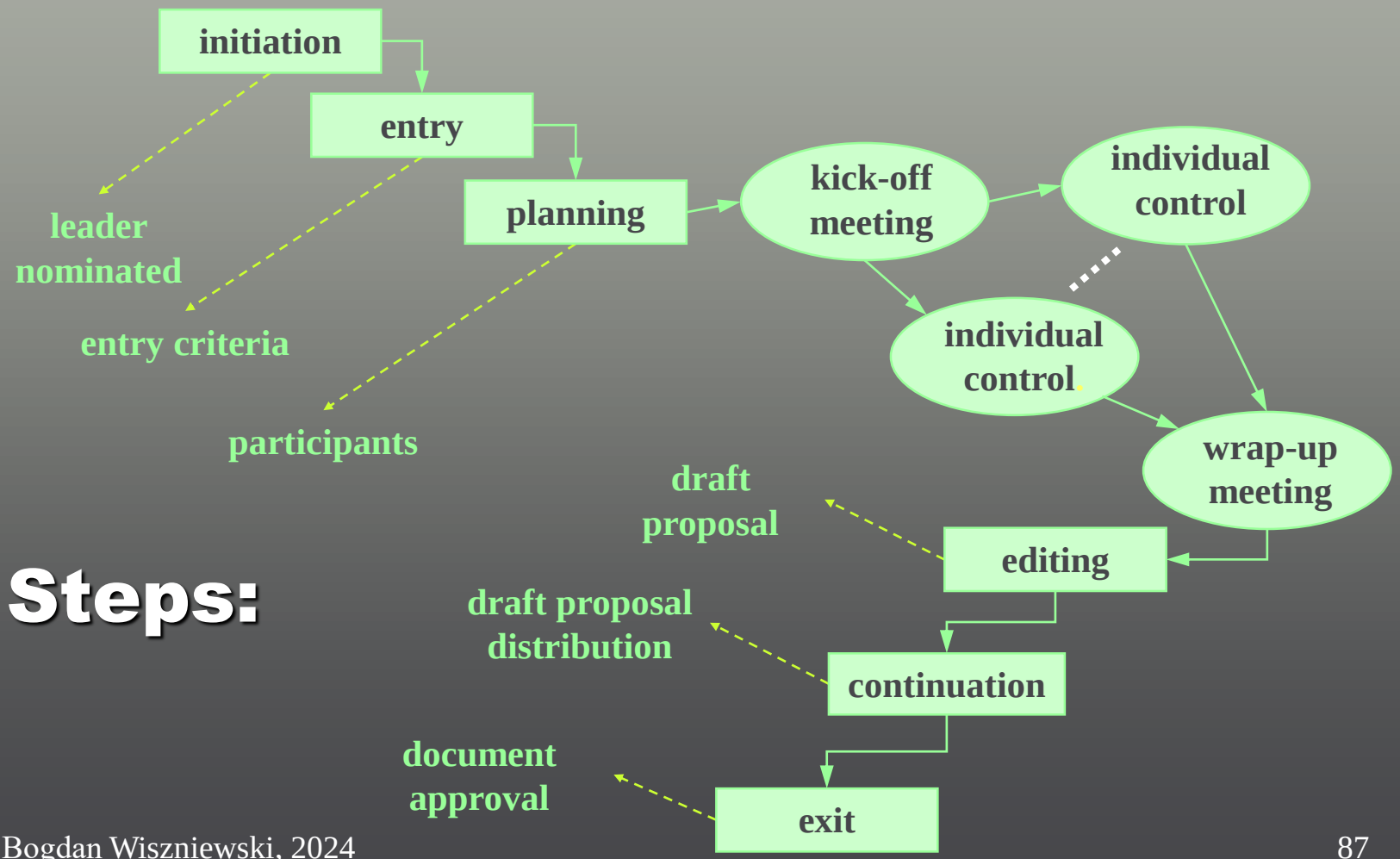
© Bogdan Wiszniewski, 2024

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

© Bogdan Wiszniewski, 2024

Software inspections



- **Steps:**

© Bogdan Wiszniewski, 2024

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.***

© Bogdan Wiszniewski, 2024

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

© Bogdan Wiszniewski, 2024

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

© Bogdan Wiszniewski, 2024

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.

© Bogdan Wiszniewski, 2024

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

© Bogdan Wiszniewski, 2024

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.

© Bogdan Wiszniewski, 2024

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

© Bogdan Wiszniewski, 2024

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

© Bogdan Wiszniewski, 2024

E. Models for dynamic analysis:

- error,
- program,
- environment

© Bogdan Wiszniewski, 2024

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***

© Bogdan Wiszniewski, 2024

Error detection

- **Testing:**

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

© Bogdan Wiszniewski, 2024

Error detection

- **Performance testing:**

Objective: ***measure physical parameters***

Environment: ***testing or target***

Reasoning: ***metrics, characteristics***

© Bogdan Wiszniewski, 2024

Characteristics of code objects

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

© Bogdan Wiszniewski, 2024

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

© Bogdan Wiszniewski, 2024

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

© Bogdan Wiszniewski, 2024

Miary strukturalne

- **Liczba cyklomatyczna (McCabe):**

The maximum number of linear, independent paths through a program

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

Liczba krawędzi (L),

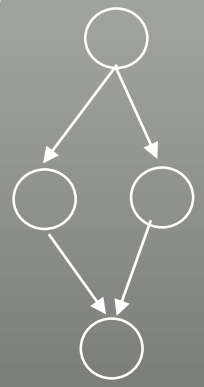
Liczba węzłów (N),

Liczba składowych spójności (P)

© Bogdan Wiszniewski, 2024

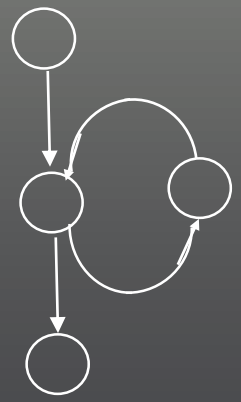
Liczba cyklomatyczna

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

© Bogdan Wiszniewski, 2024

Liczba cyklomatyczna

- **Tekst (instrukcje decyzyjne) a konstrukcja szczegółowa**

  → Brakująca lub nadmiarowa ścieżka

- **Liczba przypadków testowych ≥ M**

  (!) Kierunek przepływu sterowania nie jest brany pod uwagę, np., M(if-then-else) = M(while-do)

- **Ignorują złożoność językową**

© Bogdan Wiszniewski, 2024

Miary funkcjonalne

- **Klasy algorytmów:**

Symbol	*złożoność*	*przykład*
$\Theta(1)$	stała	tablice haszujące
$\Theta(\log n)$	logarytmiczna	wyszukiwanie binarne
$\Theta(n)$	liniowa	NWD n-cyfrowych liczb
$\Theta(n \log n)$	prawie-liniowa	zaawansowane sortowanie
$\Theta(n^c)$	wielomianowa	wyszukiwanie ścieżki
$\Theta(c^n)$	wykładnicza	generowanie liczb pierwszych

© Bogdan Wiszniewski, 2024

Źródła błędów

- **Specyfikacja wymagań:**
  - kompletność
  - spójność

© Bogdan Wiszniewski, 2024

Źródła błędów

- **Konstrukcja:**
  - poprawność
  - testowalność

© Bogdan Wiszniewski, 2024

Źródła błędów

- **Kodowanie ("przekład" algorytmu na kod programu):**
  - tekstowe,
  - niezrozumienie semantyki języka implementacji,
  - niezrozumienie semantyki algorytmu,
  - niezrozumienie wymagań.

Modele

- **Programu:**
  - Przepływ sterowania,
  - Zdarzenia,
  - Przepływ danych,
  - Tranzycje stanów
  - Model składniowy
  - Model decyzyjny
  - Model aksjomatyczny

© Bogdan Wiszniewski, 2024

Modele

- **Błędu:**
  - **Błędy w przepływie sterowania**
  - **Błędy w przepływie danych**
  - **Błędy stanu**
  - **Anomalie tekstowe**

© Bogdan Wiszniewski, 2024

Modele

- Środowiska:
  - Strumieniowe przetwarzanie sekwencyjne
  - Przetwarzanie sekwencyjne sterowane zdarzeniami
  - Przetwarzanie współbieżne
  - Przetwarzanie równoległe
  - Przetwarzanie rozproszone

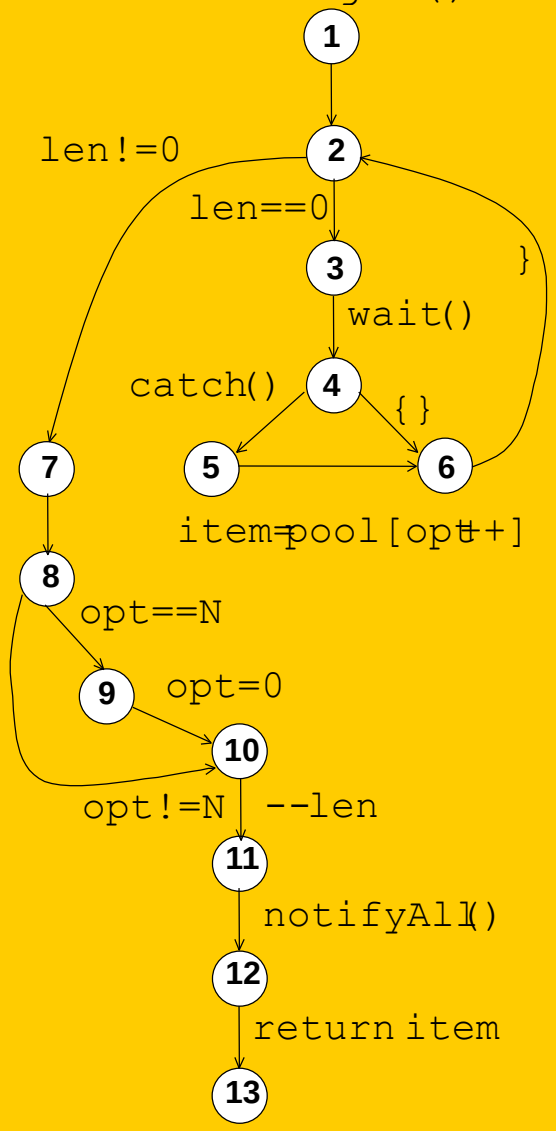
© Bogdan Wiszniewski, 2024

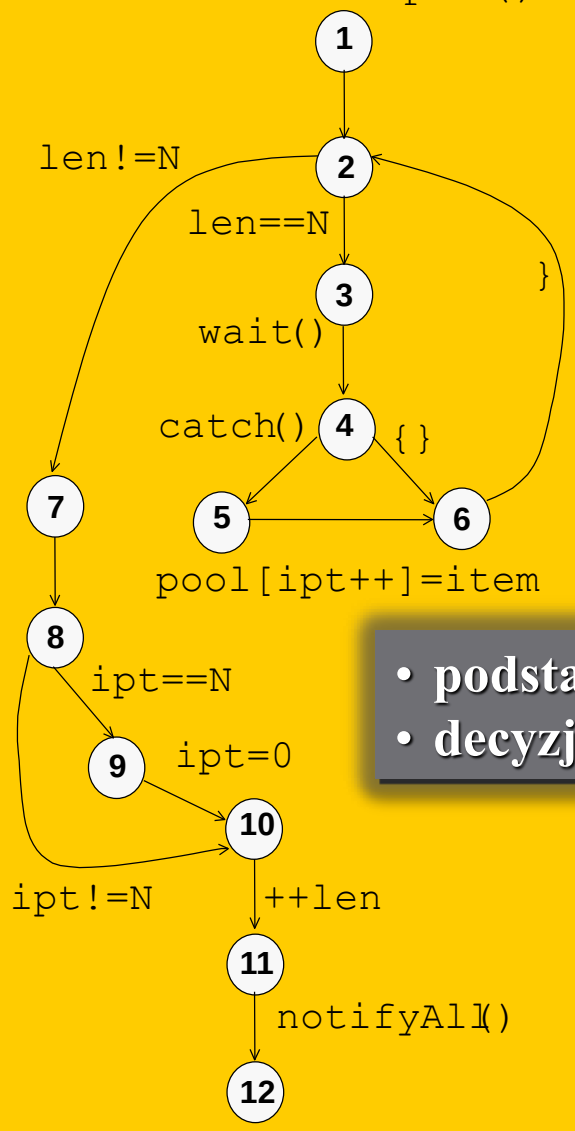
Przykładowa aplikacja

```
public class Buffer

  private int N = 10; // rozmiar buf ra
  private short // parametry bufo a
  ipt = 0,  // indeks wpisu
  opt = 0,  // indeks odczytu
  len = 0;  // liczba elementów
/** zasób dzielony**/
      private char pool[] = new char[];
/* 1*/ public synchronized char get( {
        char item
/* 2*/   while (len == 0)
/* 3*/   {try {wait(); // czekaj na producen a
/* 4*/       } catch (InterruptedException )
/* 5*/         } // obsługa wyjątk
/* 6*/   } // semafor otwart
/* 7*/   item = pool[opt++]
/* 8*/   if (opt == N)
/* 9*/       opt = 0; // modulo
/*10*/   --len; // pobrano jeden elemen t
/*11*/   notifyAll(); // bufor ma miejs e
/*12*/   return item
/*13*/ } // funkcja get(

/* 1*/ public synchronized void put(char item
/* 2*/ while (len == N)
/* 3*/   try {wait(); // czekaj na konsumen a
/* 4*/       } catch (InterruptedException e
/* 5*/         {} // obsługa wyjąt u
/* 6*/ } // semafor otwart
/* 7*/ pool[ipt++] = item
/* 8*/ if (ipt == N)
/* 9*/       ipt = 0; // modulo N
/*10*/ ++len; // zapisano jeden elemen t
/*11*/ notifyAll(); // bufor jest niepus y
/*12*/ }
/*13*/ // funkcja put(
}
```

- współdzielona tablica `pool` o `N` elementach
- metoda `Buffer.get()` do pobierania znaków
- metoda `Buffer.put()` do wpisywania znaków

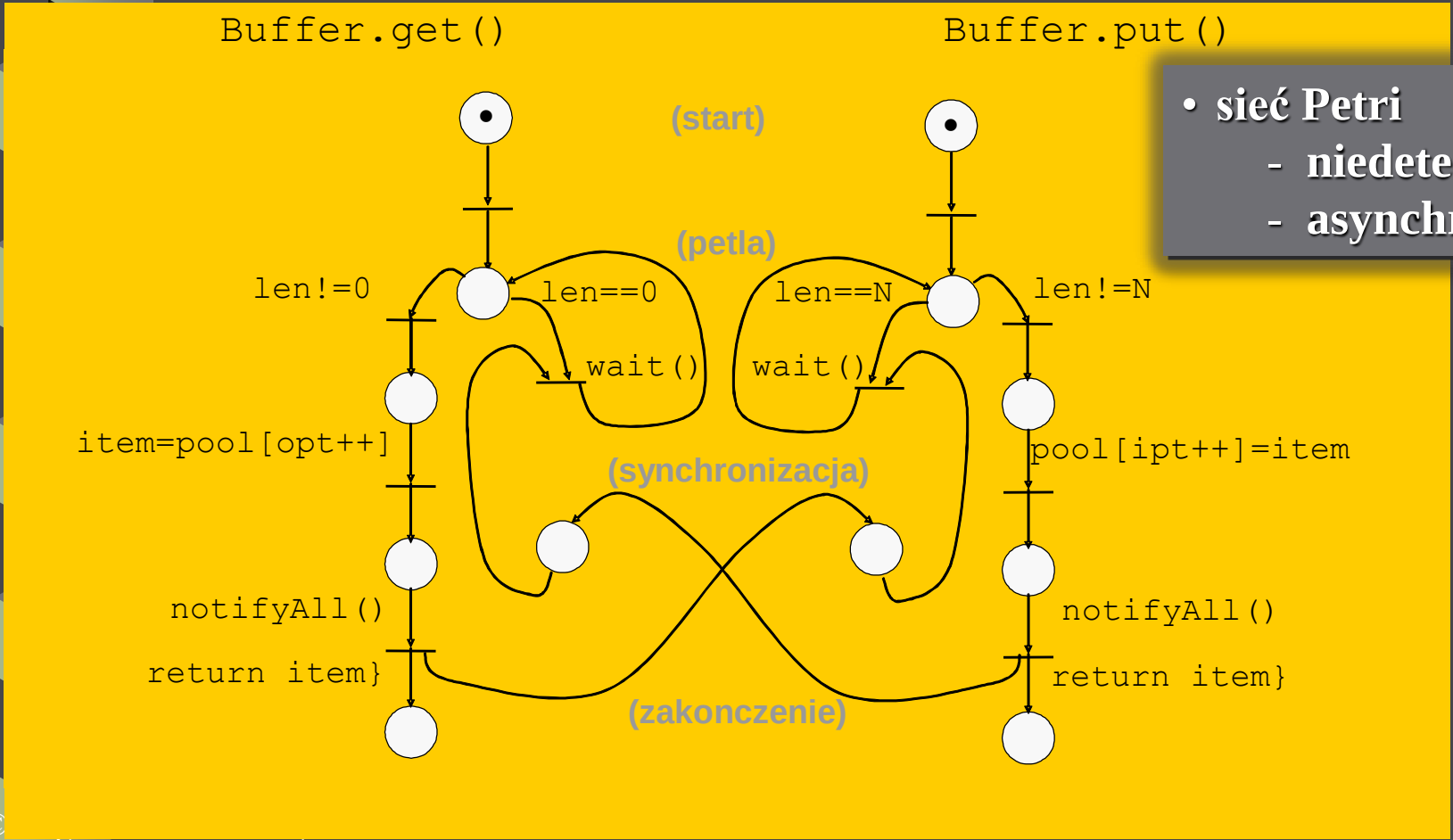
Przepływ sterowania





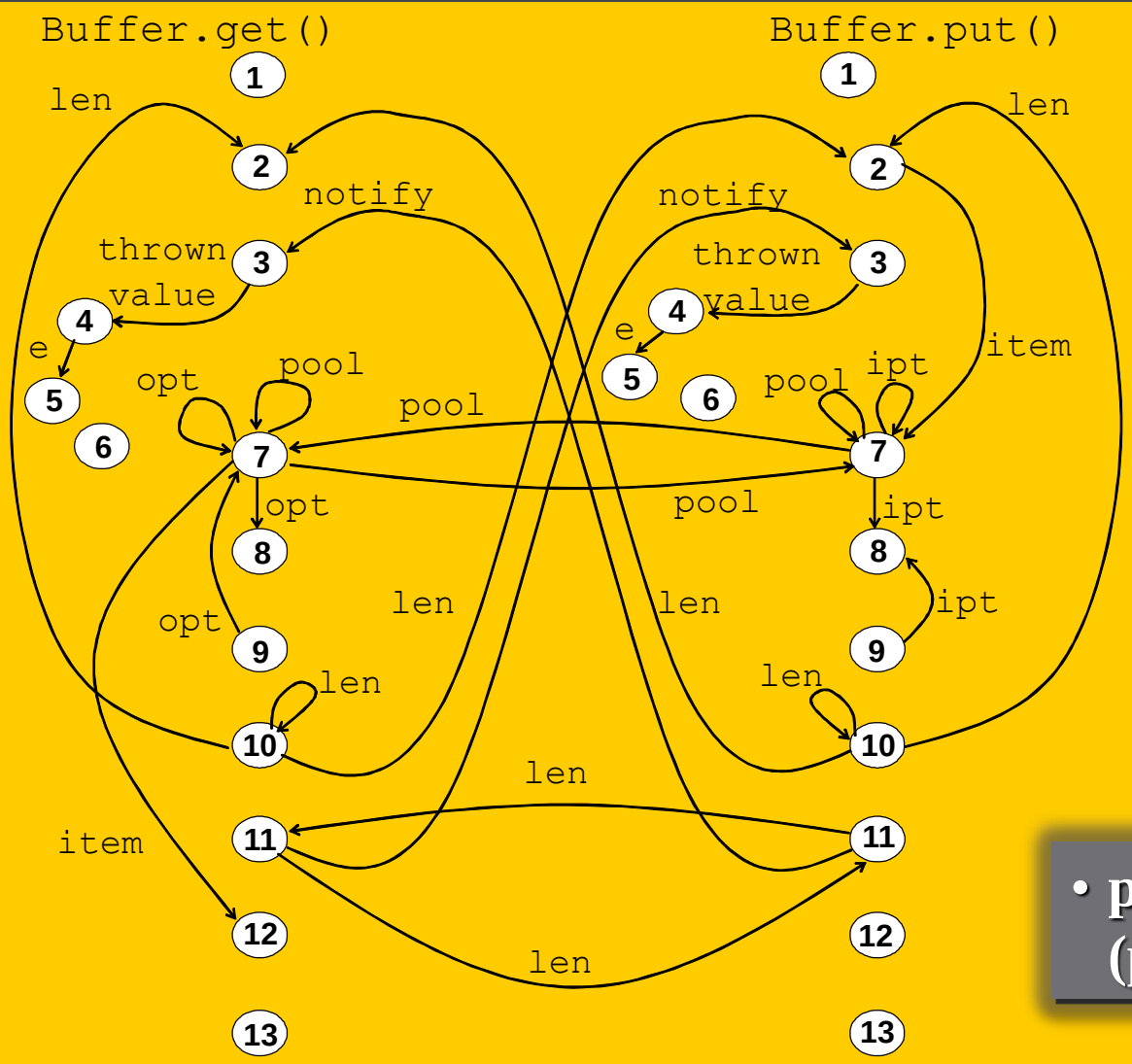
- podstawienia (w. proste)
- decyzje (rozgałęzienia)

Zdarzenia



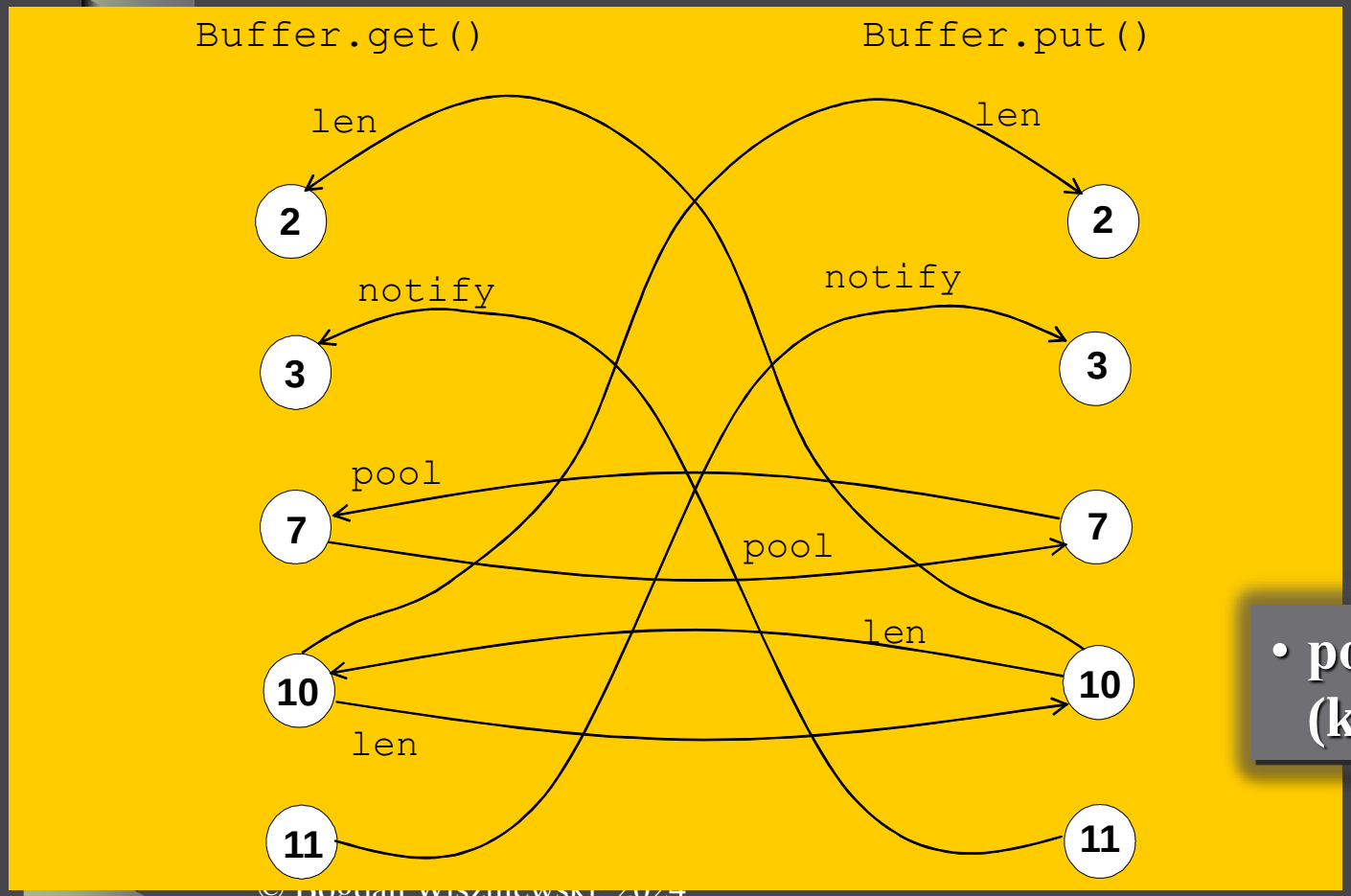
- sieć Petri
  - niedeterminizm
  - asynchronizm

Przepływ danych



- **poziom instrukcji (przetwarzania)**

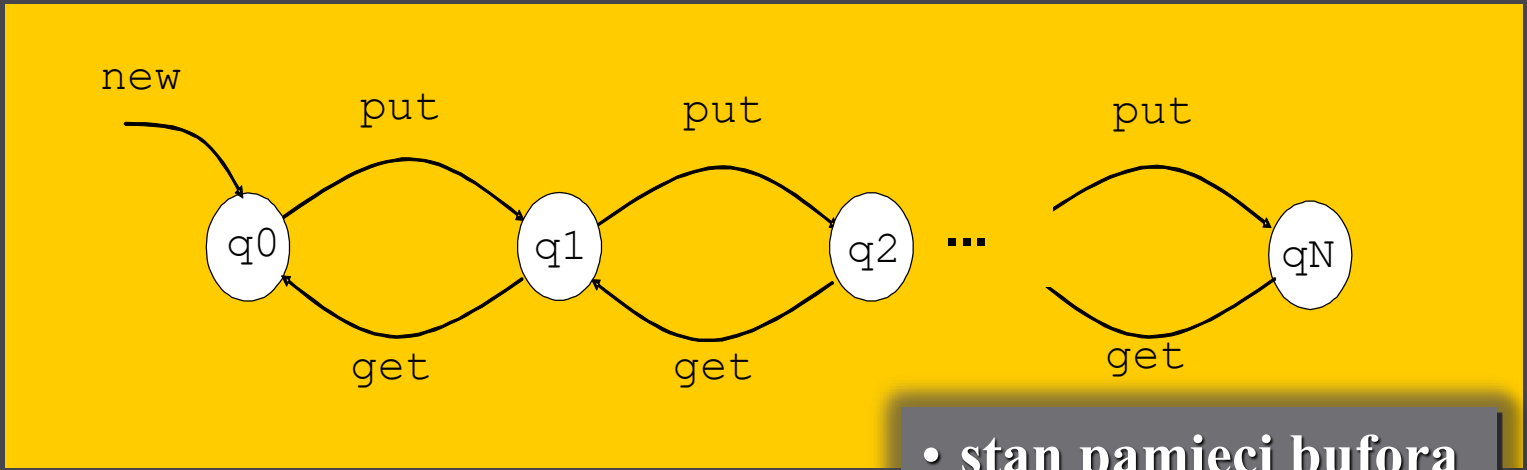
Przepływ danych



- **poziom wątków (komunikacji)**

© Bogdan Wiszniewski, 2024

Tranzycje stanu



- stan pamięci bufora

Model składniowy

	wątek konsumenta		*wątek producenta*
R1	Czekanie przy pustym buforze	W1	Czekanie przy pełnym buforze
R2	Czytanie po czekaniu	W2	Pisanie po czekaniu
R3	Czytanie bez czekania pozostawiające niepusty bufor	W3	Pisanie bez czekania pozostawiające niepełny bufor
R4	Czytanie bez czekania opróżniające bufor (pobranie ostatniego elementu)	W4	Pisanie bez czekania zapełniające bufor (zapisanie ostatniego wolnego miejsca)

wątek konsumenta → R1;R2;R3;R4;R3

Buffer → R1;W3;R2;W3;W3;R3;R4;W3^{N};W3;R3;W2

wątek producenta → W3;W3;W3;W3^{N};W3;W2

© Bogdan Wiszniewski, 2024

Model decyzyjny

- **możliwość automatycznej weryfikacji**
  - **kompletności**
  - **niesprzeczności**

			reguły decyzyjne			
			1	2	3	4
warunki	C_1	len == 0	1	0	0	0
	C_2	len == N	0	1	0	0
	C_3	opt == N	0	0	1	0
	C_4	ipt == N	0	0	0	1
akcje do wyboru	A_1	get: wait()	X	–	–	–
	A_2	put: wait()	–	X	–	–
	A_3	get: opt = 0	–	–	X	–
	A_4	put: ipt = 0	–	–	–	X
			akcje wybrane			

© Bogdan Wiszniewski, 2024

Model aksjomatyczny

	D		R
	{0,1,..,N-1}		{0,1,..,N-1}
PUT	ipt	→	ipt
GET	opt	→	opt

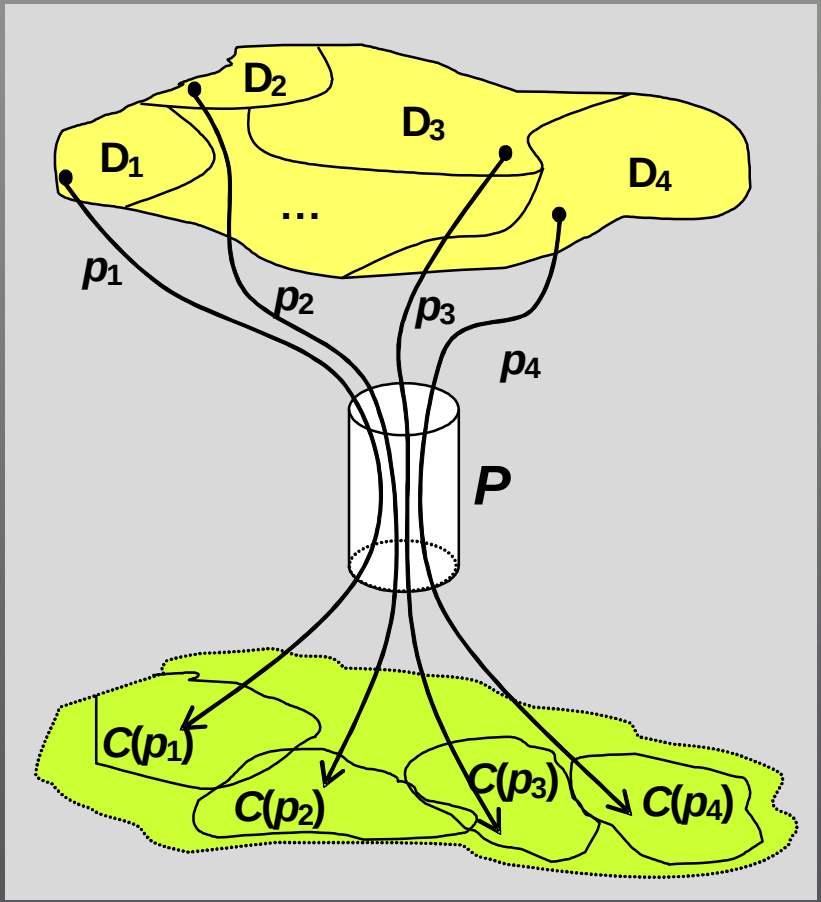
- **asercje:**
  1. $PUT^0(ipt) = GET^{len}(opt)$
  2. $GET^0(opt) = PUT^{N-len}(ipt)$

kroki dowodu:

1. $ipt = opt \Rightarrow ipt = GET^0(opt)$
2. $ipt \neq opt \wedge ipt = GET^{len-1}(GET(opt)) \Rightarrow ipt = GET^{len}(opt)$

© Bogdan Wiszniewski, 2024

Błędy przepływu sterowania



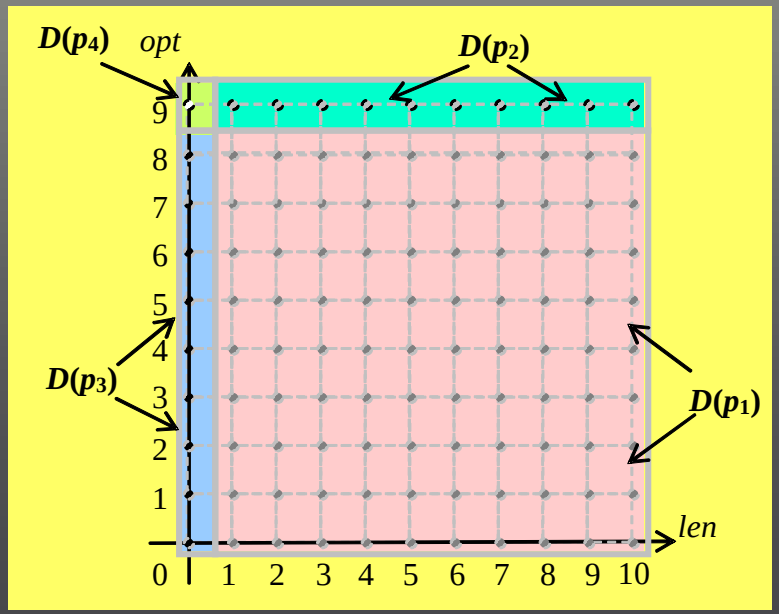
warunki wykonania ścieżek:

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

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

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

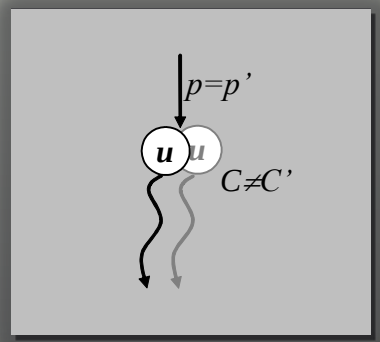
$$p_4(\bar{x}) = (len_0 = 0) \wedge (opt_0 + 1 = 10)$$



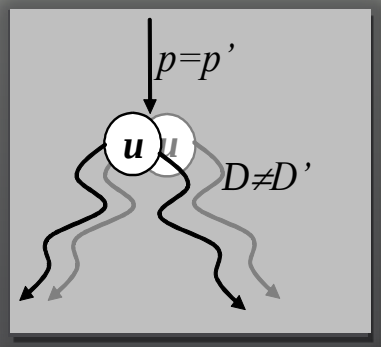
© Bogdan Wiszniewski, 2024

Błędy przepływu sterowania

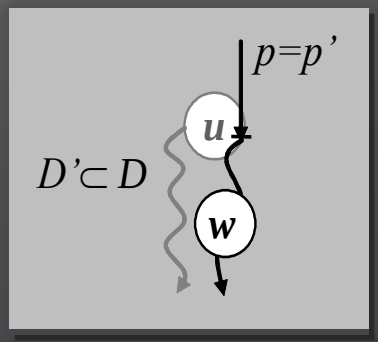
1. Błąd ‘obliczenia’ ścieżki:



2. Błąd ‘domeny’ ścieżki:



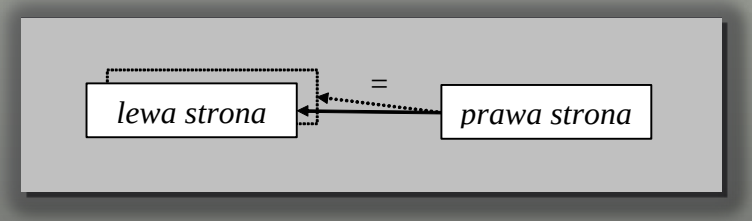
3. Błąd ‘brakującej’ ścieżki:



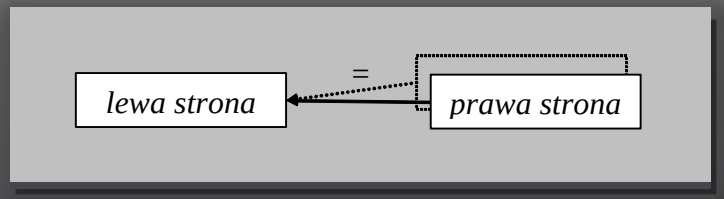
© Bogdan Wiszniewski, 2024

Błąd przepływu danych

1. Błąd ‘przypisania’:



2. Błąd ‘wyznaczenia wartości’:

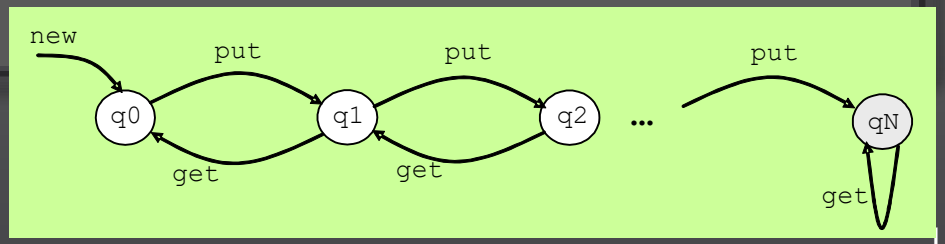


© Bogdan Wiszniewski, 2024

Błędy stanu

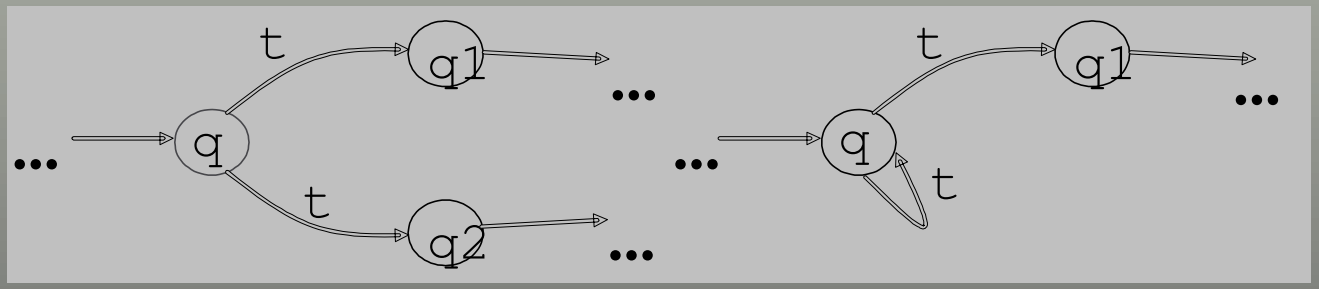
- **Zakleszczenie (ang. deadlock)**

```
public synchronized char get() {
        char item;
        while (len == 0) {
            try {
                wait(); // czekaj na producenta
            } catch (InterruptedException e) {}
        } //semafor otwarty
        item = pool[opt++];
        if (opt == N) opt = 0; // modulo N
        --len; // pobrano jeden element
/*      notifyAll(); // bufor ma wolne miejsce */
        return item;
 }
```

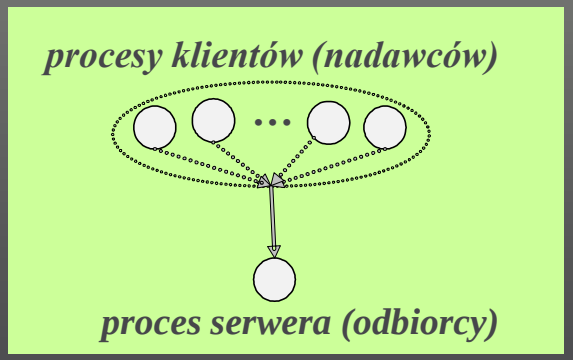


Błędy stanu

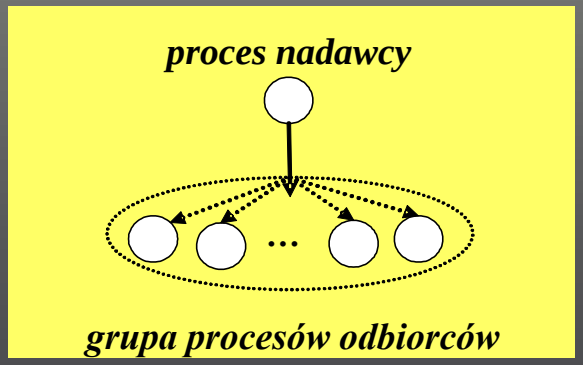
- **Wyścigi**



- przy odbiorze:



- przy nadawaniu:



Anomalie tekstowe

- **Interpretacja składni**

```
public synchronized char get() {
    char item;
    while (len == 0);
        try {
         wait(); // czekaj na producenta
      } catch (InterruptedException e) {}
    } //semafor otwarty
    item = pool[opt++];
    if (opt == N);
          opt = 0; // modulo N
    notifyAll(); // bufor ma wolne miejsce
     --len; // pobrano jeden element
    return item;
```

nadmiarowy średnik

pusta instrukcja

nadmiarowy średnik

pusta instrukcja

Anomalie tekstowe

- **Efekty uboczne**

```
int main() {
        int x,y;
        int *z;
        z=&x;
        z++=1; /* inicjacja zmiennej x */
        z=2; /* domniemana inicjacja zmiennej y */
}
```

Anomalie tekstowe

- **Niejawne konwersje typu**

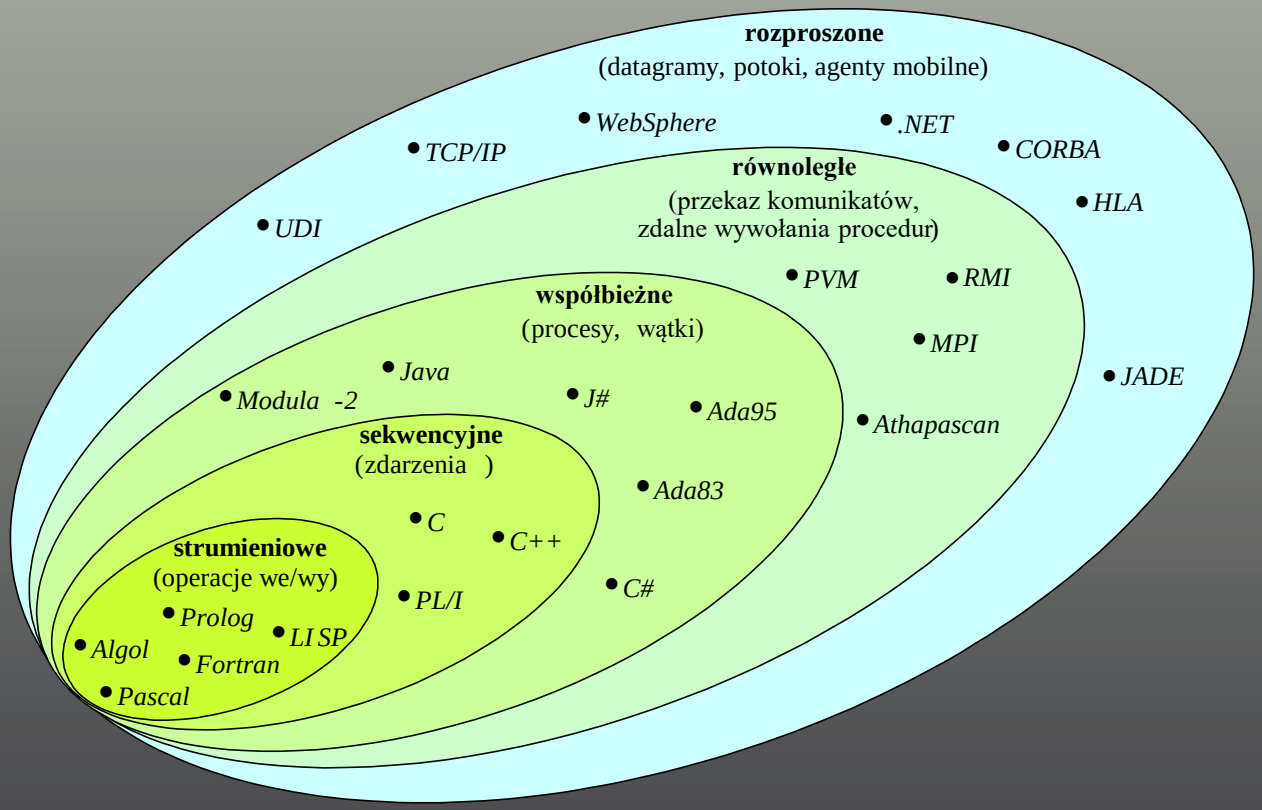
```
…
void ff(int); // funkcja z argumentem typu int
…
int ival=3.14; // zawężenie wartości 3.14 do 3, ival=3;
ff(3.14); // zawężenie wartości 3.14 do 3,
          // wywołanie ff(3);
ival=4.0; // konwersja 4.0 do 4 bez zawężenia, ival=4;
…
double fval=5; // promocja wartości 5 do typu
               // rozszerzonego 5.0, fval=5.0;
int val=1;
fval=val+3.14; // promocja 1 do typu rozszerzonego 1.0,
               // fval=1.0+3.14;
…
```

Anomalie tekstowe

- **Model programowania**
  - **dostęp niekontrolowany**
  - **dostęp zagrożony**
  - **dostęp selektywny**

C, Fortran → Modula-2, Ada83 → Java, Ada95 → …?

Modele środowiska

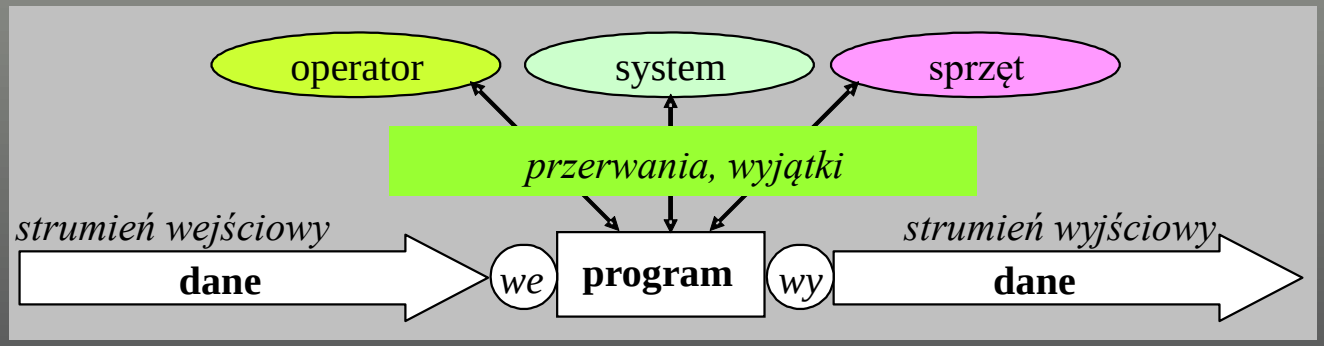


Modele środowiska

- **Strumieniowe przetwarzanie sekwencyjne**

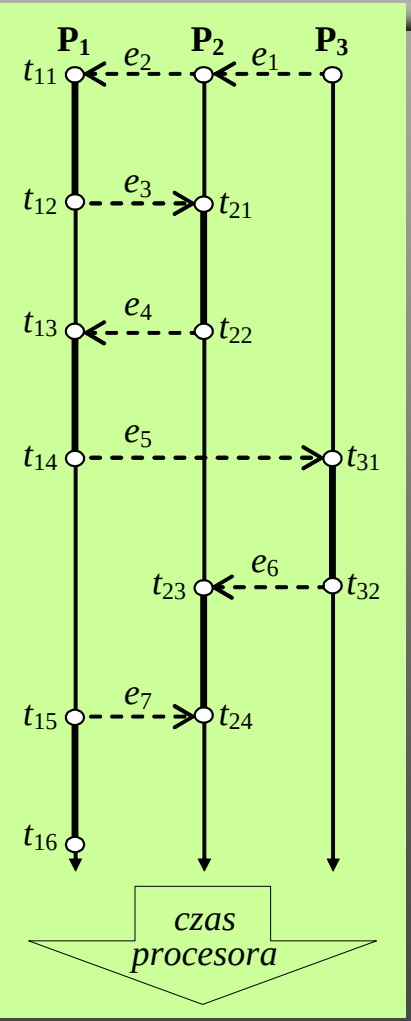
Modele środowiska

- **Przetwarzanie sekwencyjne sterowane zdarzeniami**



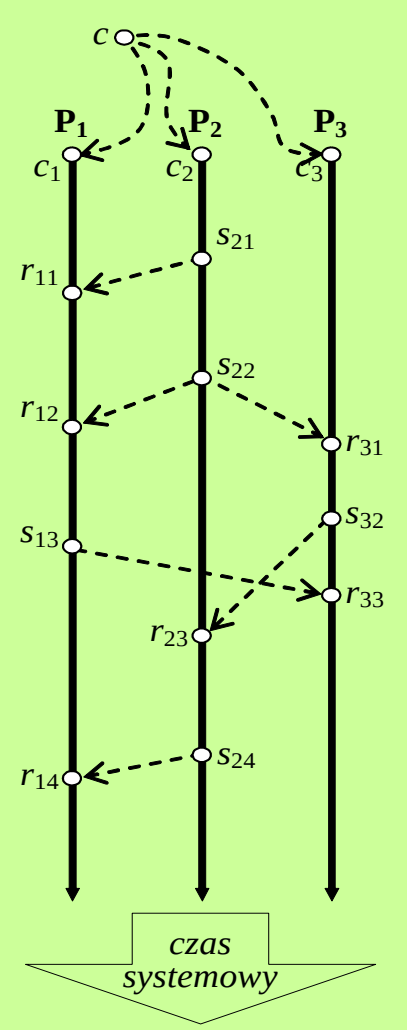
Modele środowiska

- **Przetwarzanie współbieżne**



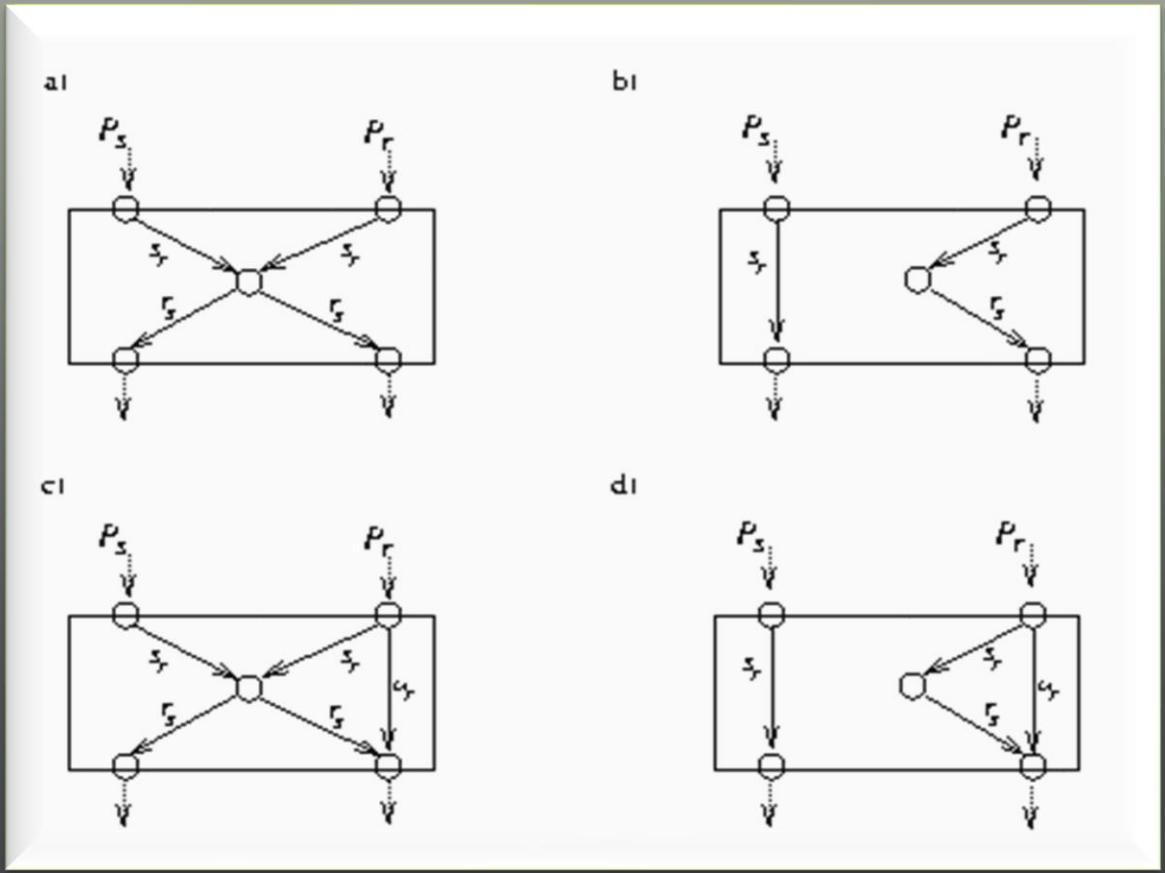
Modele środowiska

- **Przetwarzanie równoległe**



Modele środowiska

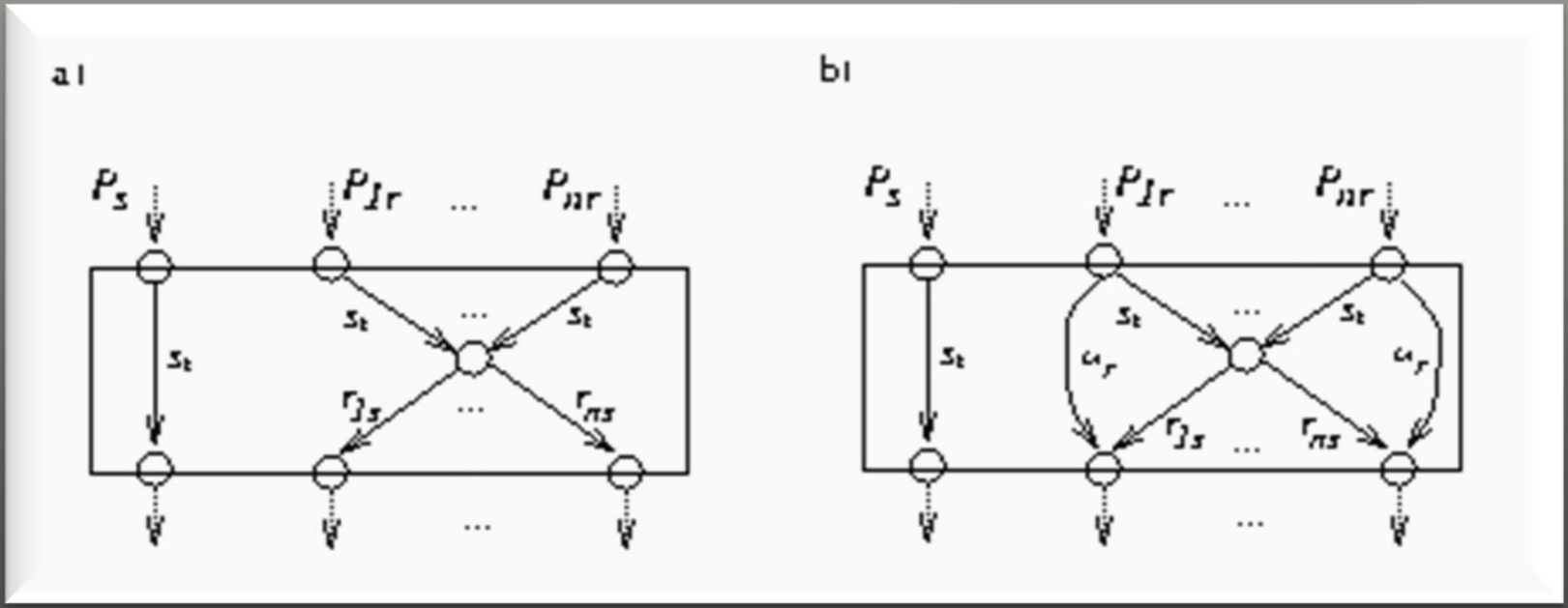
- **Zdarzenia komunikacyjne (1-1)**



© Bogdan Wiszniewski, 2024

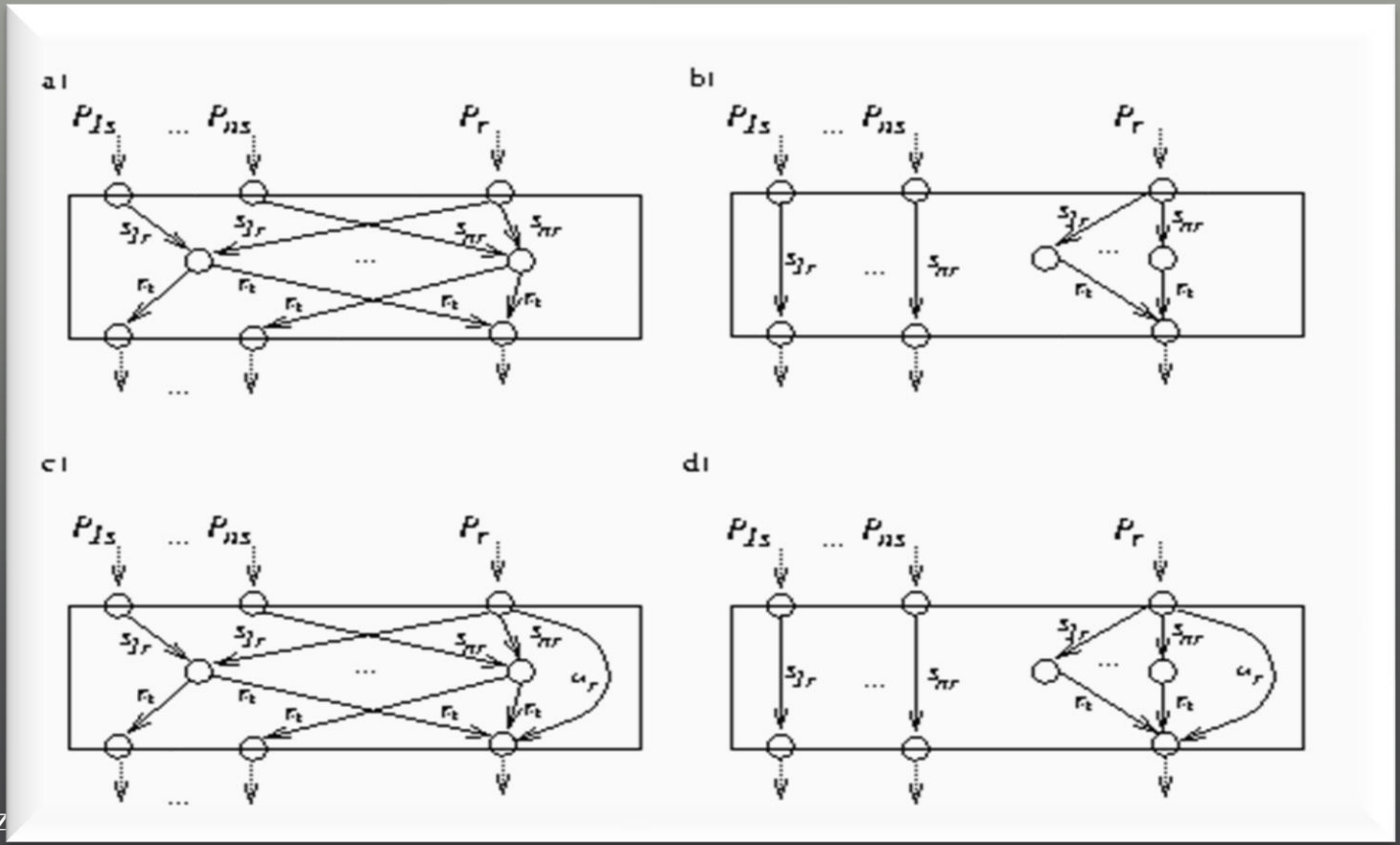
Modele środowiska

- **Zdarzenia komunikacyjne (1-n)**



Modele środowiska

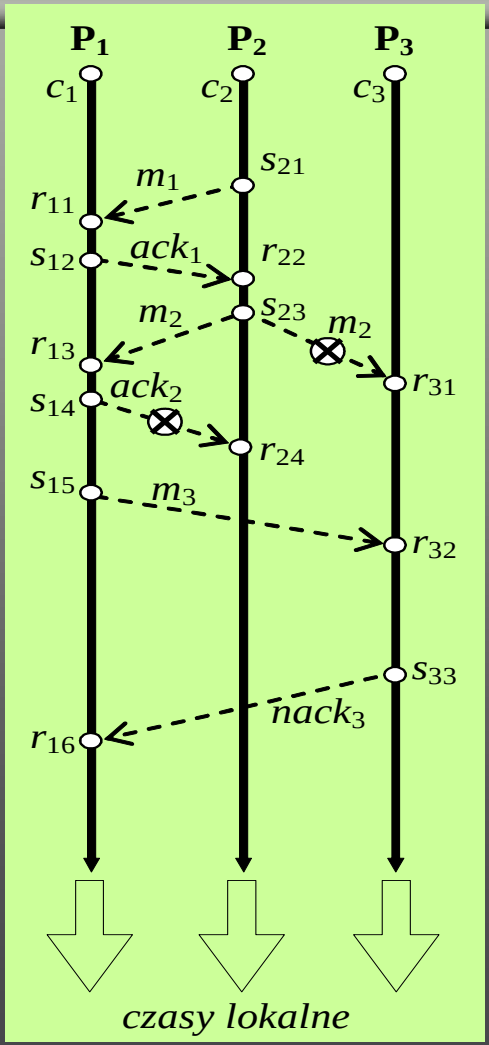
- **Zdarzenia komunikacyjne (n-1)**



© Bogdan Wisz

Modele środowiska

- **Przetwarzanie rozproszone**



Techniki analizy dynamicznej

- **Funkcjonalne ("black-box"):**
  - Program = funkcja,
  - Przypadki testowe na podstawie specyfikacji wymagań
  - Potencjalnie wszystkie błędy ale w potencjalnie nieskończonym czasie

© Bogdan Wiszniewski, 2024

Techniki analizy dynamicznej

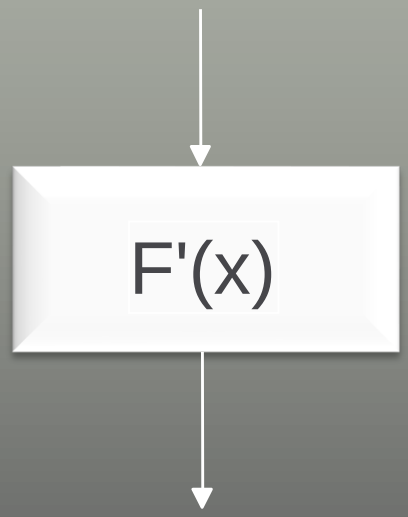
- **Strukturalne ("white-box"):**
  - Program = struktura,
  - Przypadki testowe na podstawie specyfikacji konstrukcji i/lub kodu źródłowego
  - Nie wszystkie błędy ale w skończonym czasie

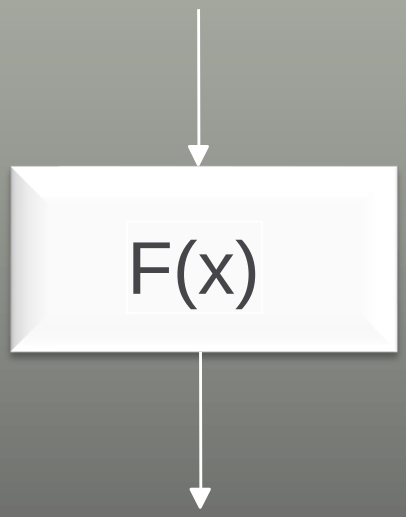
© Bogdan Wiszniewski, 2024

F. Testowanie funkcjonalne (”black-box”)

© Bogdan Wiszniewski, 2024

Strategie funkcjonalne





specyfikacja

we	wy
T1	R1
T2	R2
...	...
Tn	Rn

program

T1, T2, ..., Tn→ R1, R2, ..., Rn

© Bogdan Wiszniewski, 2024

Strategie funkcjonalne

- **Własności obliczeniowe:**

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

- **Ograniczenia:**

! Nierozstrzygalność równoważności funkcji (np. pierwotnie rekurencyjnych)

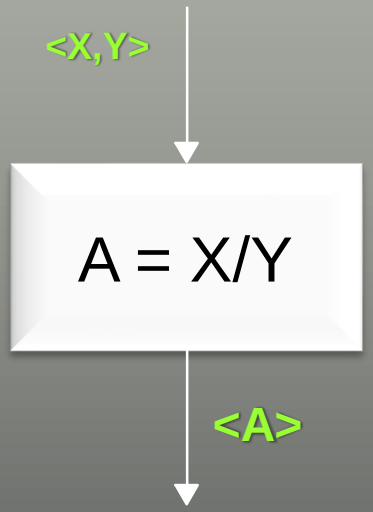
! Przybliżona arytmetyka binarna (dokładność zmiennego przecinka, zaokrąglenia, błąd nadmiaru)

Strategie funkcjonalne

- Metoda specjalnych wartości
- Wartości transcendentne
- Równoważność wielomianów
- Wartości losowe

© Bogdan Wiszniewski, 2024

Przykład metody specjalnych wartości



<...,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

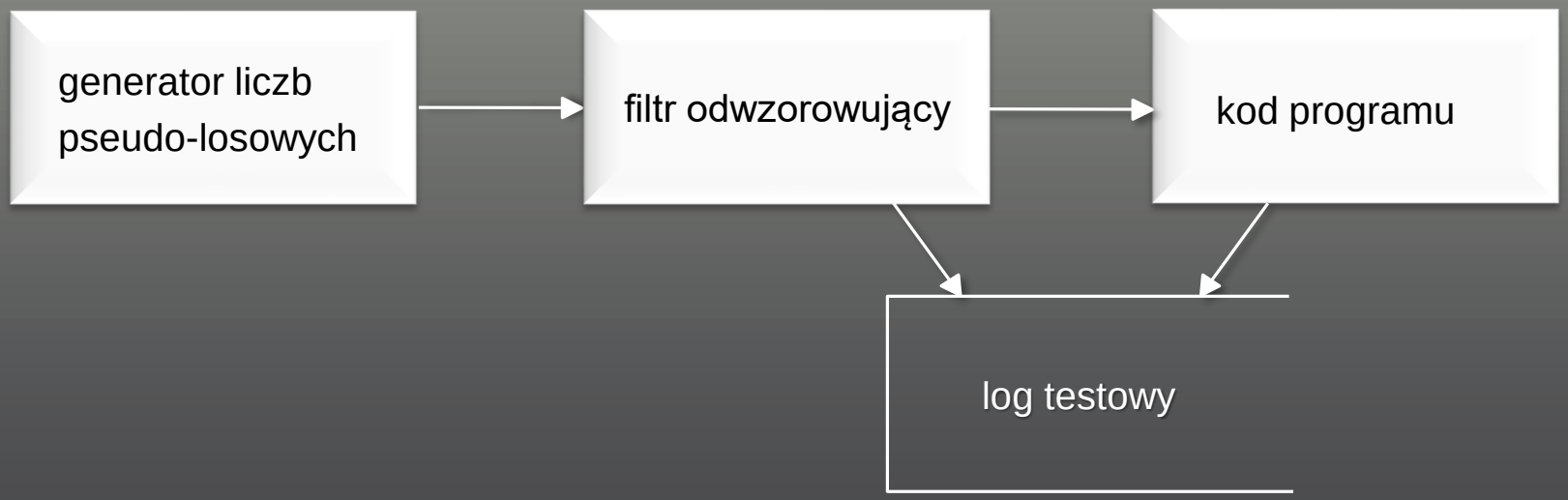
Równoważność wielomianów

(!) Arytmetyka komputerowa oblicza funkcje standardowe za pomocą wielomianów

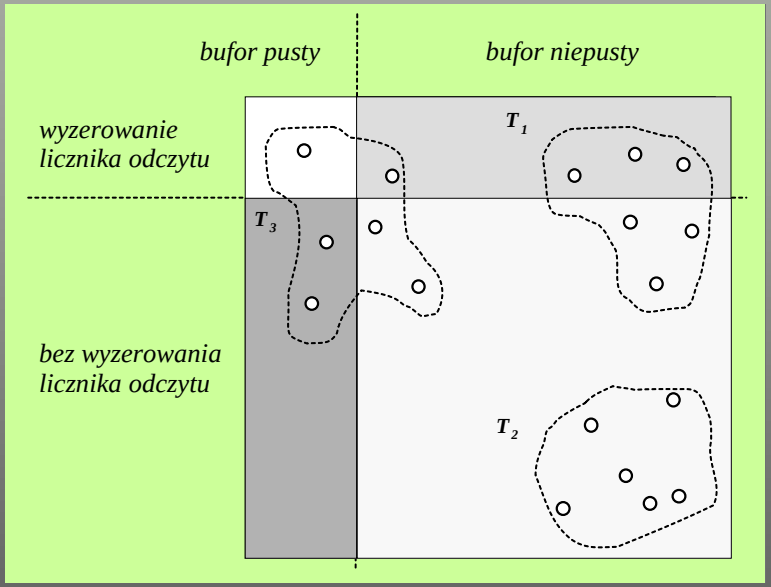
- **Klasyczna algebra wielomianów:**
  - **Klasa wielomianów cf(n,x),**
  - **Badana F, specyfikowana $F' \in CF$**
  - **$T=\{t_1, t_2, \ldots, t_{n+1}\}$**
  - **$F'(T)=F(T) \Rightarrow F'(x) \equiv F(x)$**

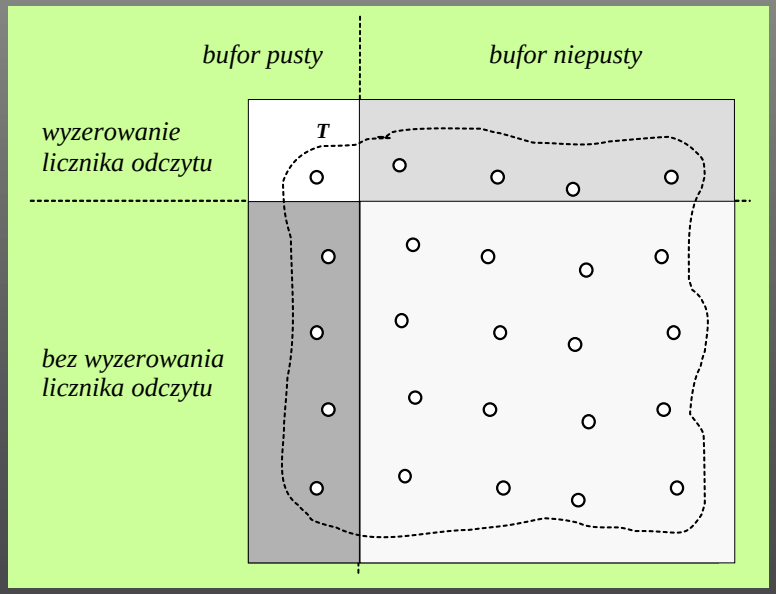
Wartości losowe

(!) Sprawdzić działanie programu dla możliwie najbardziej typowych i najczęściej spotykanych wartości wejściowych



Wartości losowe





© Bogdan Wiszniewski, 2024

G. Testowanie strukturalne (”white-box”)

Strategie strukturalne

- Model strukturalny (program, system)
  - testowanie przepływu sterowania
  - testowanie przepływu danych
  - testowanie mutacyjne

© Bogdan Wiszniewski, 2024

Strategie strukturalne

- **Ocena testu:**
  - **ilościowa (miary)** → *reguła kciuka*
  - **jakościowa (model)** → *błędy to odchyłki*
- **Zaliczenie testu:**
  - **wykonano założoną listę przypadków**
  - **te same wyniki otrzymane i przewidywane**

→ *strategia testowania*

Strategie strukturalne

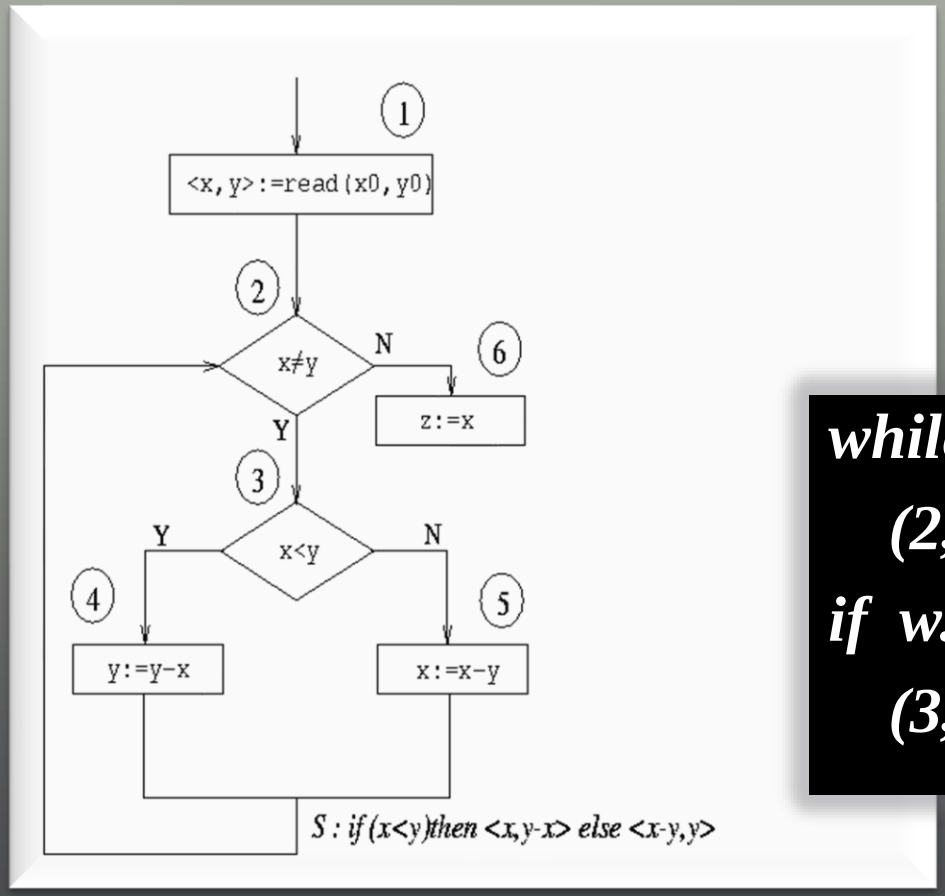
- Testowanie rozgałęzień
- Testowanie ścieżek
  - metoda obiegów "granicznych" (ang. boundary-interior)
  - testowanie domen
  - równoważność obliczeniowa ścieżek
  - wzorce pętli prostych

Strategie strukturalne

- **Testowanie przepływu danych**
  - **łańcuchy definicja-użycie**
- **Testowanie mutacyjne**
  - **anomalie tekstowe**

© Bogdan Wiszniewski, 2024

Testowanie rozgałęzień



(!) Każdy predykat na "tak" i na "nie"

while w. "2":
(2,3), (2,6)
if w. "3":
(3,4), (3,5)

Testowanie ścieżek

(!) Błędy w przepływie sterowania powodują błędne wyniki

(!) Ścieżka to dobry przypadek testowy

- **Model programu:**
  - Graf przepływu sterowania: $G(a,n,s,e)$
  - Zmienne wejściowe: $x=\langle x_1, x_2, ..., x_n\rangle$
  - Dziedzina wejściowa: $D=X_1\times X_2\times ...\times X_n$
  - Ścieżka: $p=(n_0, n_1, ..., n_k)$
  - Warunek ścieżki: $p(x)$
  - Dziedzina ścieżki: $d(p)=\{ x \mid p(x)\}$
  - Obliczenie ścieżki: $c(p): d(p) \rightarrow R$

© Bogdan Wiszniewski, 2024

Testowanie ścieżek

- **Strategie testowania ścieżek**
  - Metoda obiegów "granicznych" (ang. boundary-interior)
  - Testowanie domen
  - Równoważność obliczeniowa ścieżek
  - Wzorce pętli prostych

Metoda obiegów granicznych

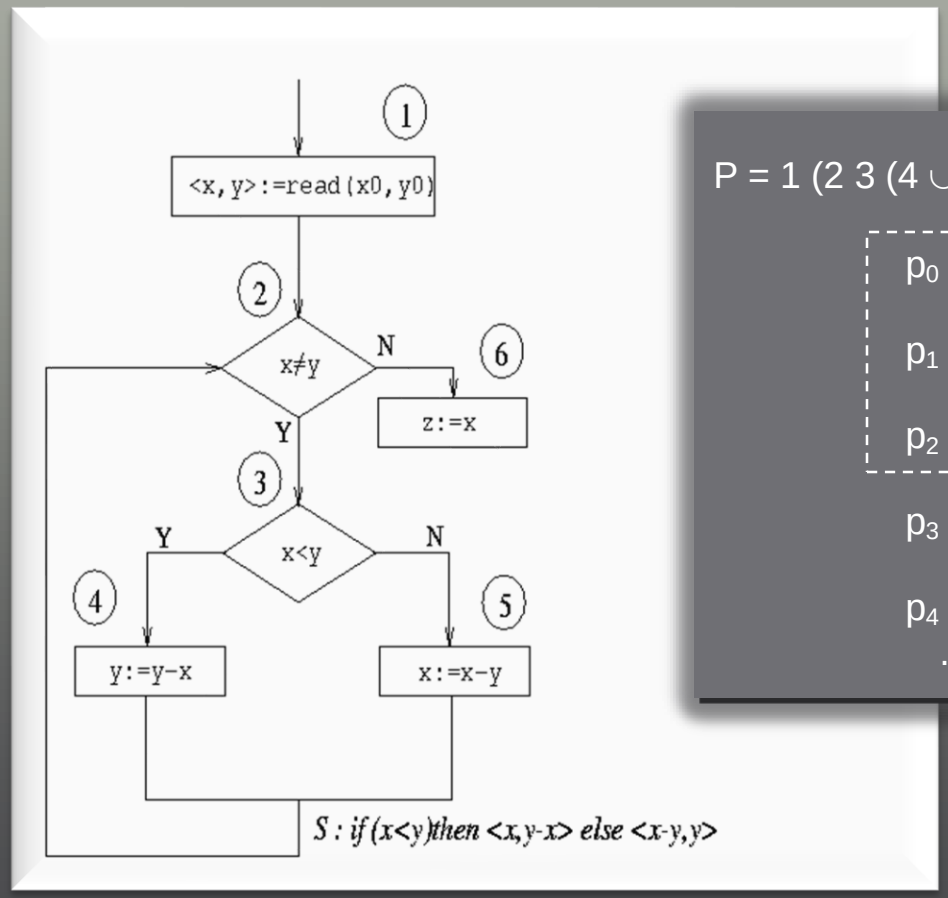
(!) Problem pętli

- **Kryterium intuicyjne:**
  - **Każda pętla ZERO i nie-zero obiegów,**
  - **Każda pętla MAKS liczbę obiegów**

→ *silniejsze od testowania rozgałęzień*

© Bogdan Wiszniewski, 2024

Przykład



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

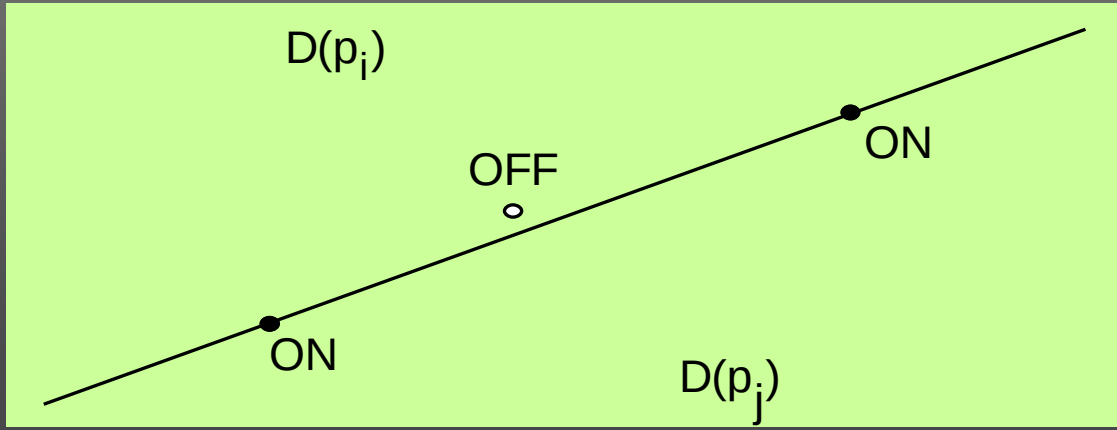
...

© Bogdan Wiszniewski, 2024

Testowanie domen

(!) Koncentracja na błędach dziedziny

- **Założenia:**
  - Predykaty p(x) są liniowymi funkcjami na X,
  - Obliczenia ścieżek c(p) są różne,
  - Nie zachodzi przypadkowa poprawność.

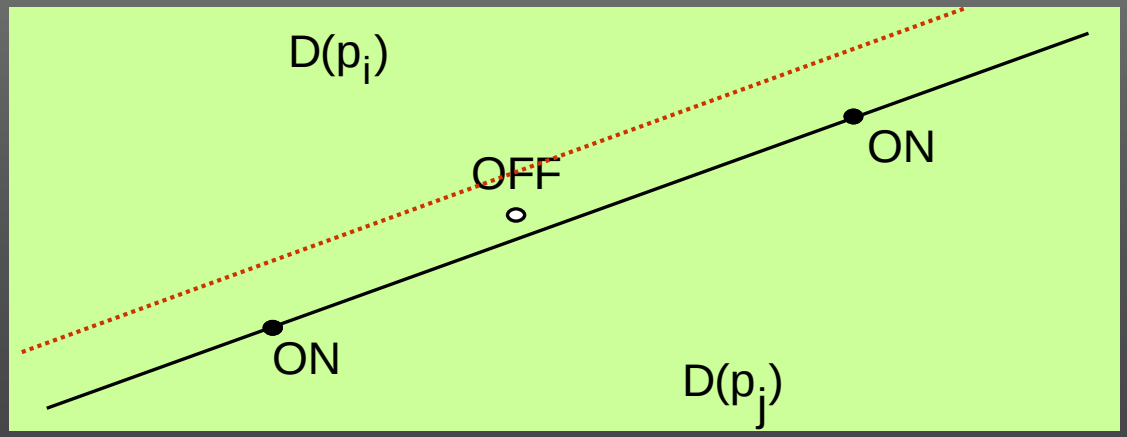


© Bogdan Wiszniewski, 2024

Testowanie domen

(!) Koncentracja na błędach dziedziny

- **Założenia:**
  - **Predykaty p(x) są liniowymi funkcjami na X,**
  - **Obliczenia ścieżek c(p) są różne,**
  - **Nie zachodzi przypadkowa poprawność.**

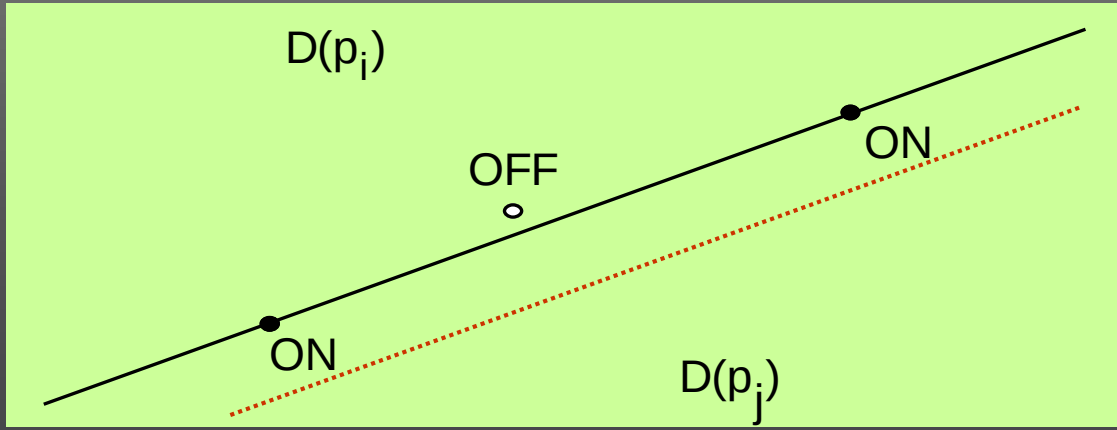


© Bogdan Wiszniewski, 2024

Testowanie domen

(!) Koncentracja na błędach dziedziny

- **Założenia:**
  - Predykaty p(x) są liniowymi funkcjami na X,
  - Obliczenia ścieżek c(p) są różne,
  - Nie zachodzi przypadkowa poprawność.

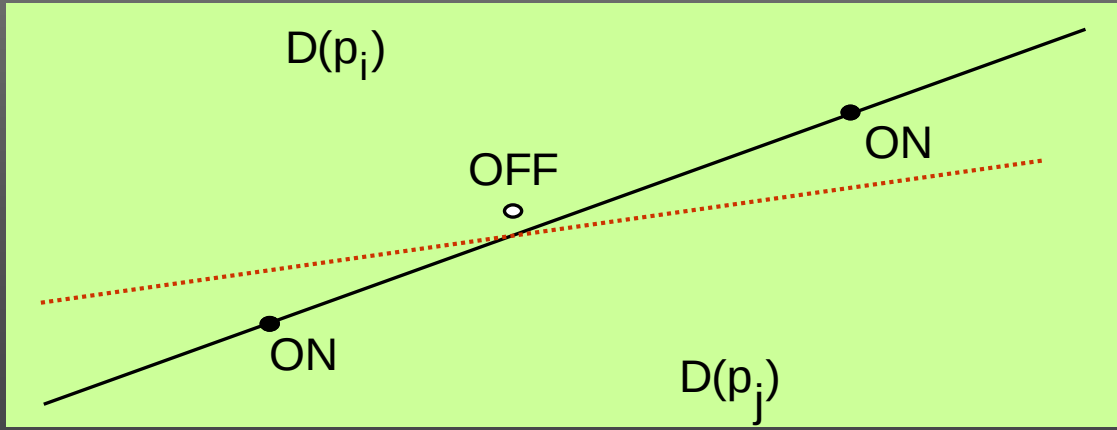


© Bogdan Wiszniewski, 2024

Testowanie domen

(!) Koncentracja na błędach dziedziny

- **Założenia:**
  - **Predykaty p(x) są liniowymi funkcjami na X,**
  - **Obliczenia ścieżek c(p) są różne,**
  - **Nie zachodzi przypadkowa poprawność.**

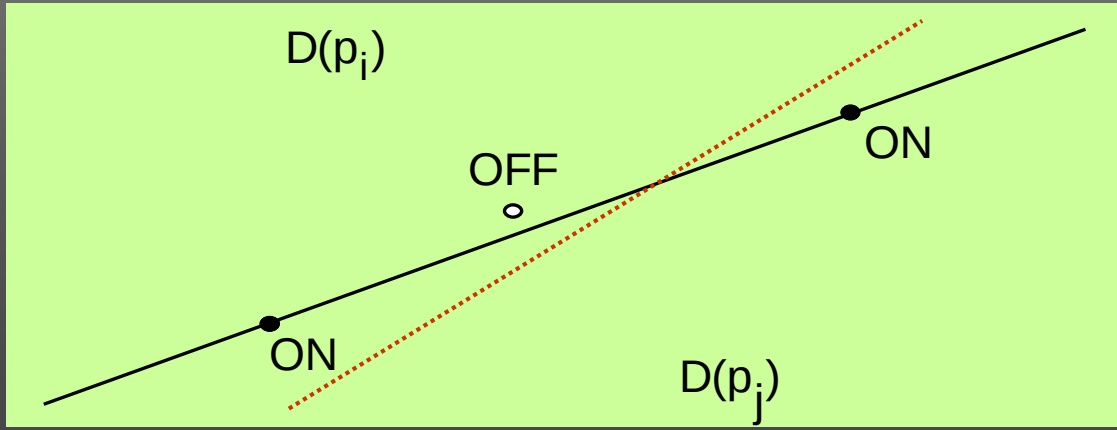


© Bogdan Wiszniewski, 2024

Testowanie domen

(!) Koncentracja na błędach dziedziny

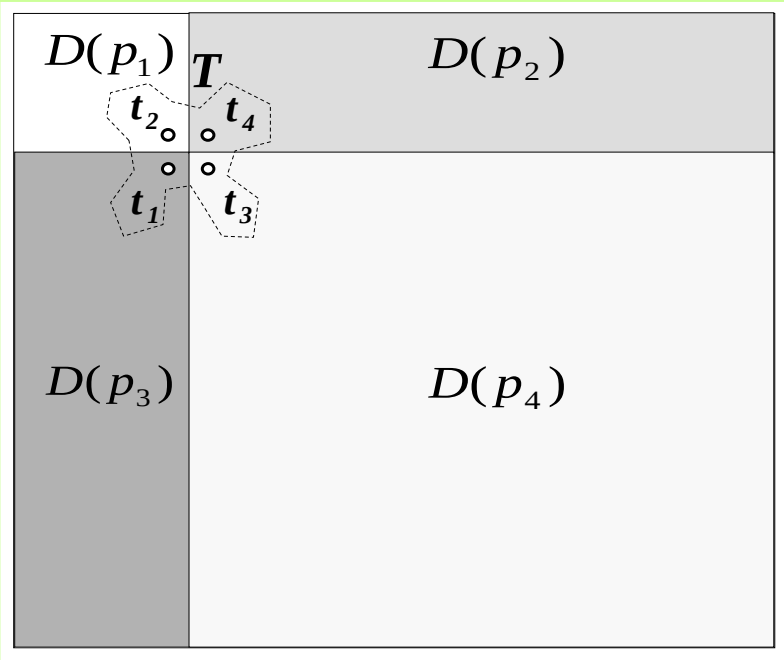
- **Założenia:**
  - **Predykaty p(x) są liniowymi funkcjami na X,**
  - **Obliczenia ścieżek c(p) są różne,**
  - **Nie zachodzi przypadkowa poprawność.**



© Bogdan Wiszniewski, 2024

Przykład

- **Testowanie domen**



© Bogdan Wiszniewski, 2024

Obliczeniowa równoważność ścieżek

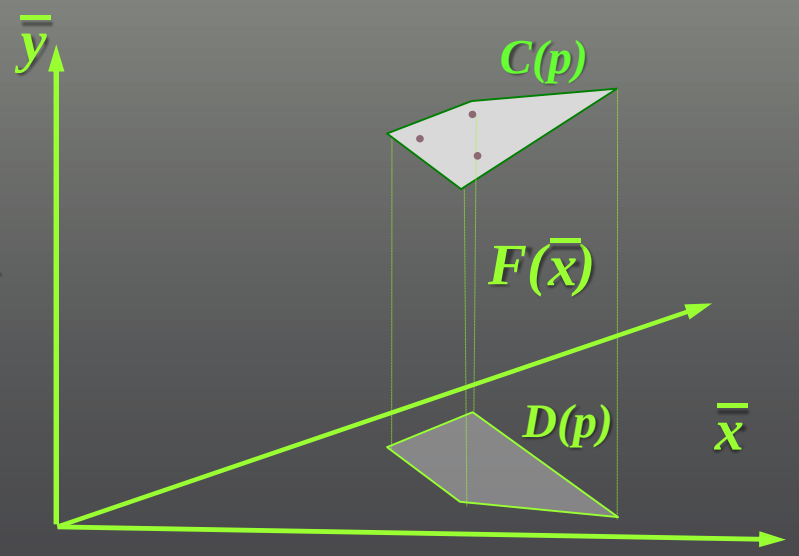
- zmienne wejściowe: $x=\langle x_1, x_2, \ldots, x_n\rangle$
- dziedzina wejściowa: $D=X_1\times X_2\times\ldots\times X_n$
- zmienne wyjściowe: $y=\langle y_1, y_2, \ldots, y_m\rangle$
- przestrzeń obliczeń: *liniowa, (n+m)-wymiarowa*

przypadek testowy:

ścieżka p

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

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



© Bogdan Wiszniewski, 2024

Przykład

- **Testowanie obliczeń:**
  - **ścieżka**

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

  - **obliczenie**

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

  - **hiperpłaszczyzna**

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

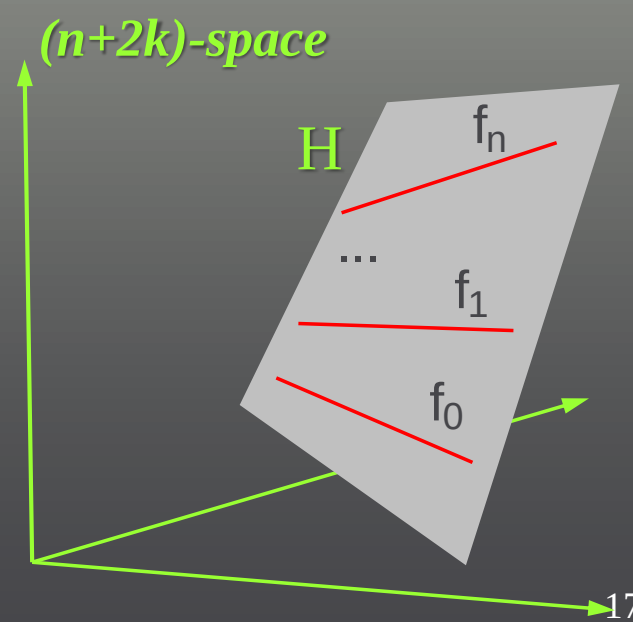
Wzorce pętli prostych

- zmienne wejściowe: $x=\langle x_1, x_2, \ldots, x_n\rangle$
- dziedzina wejściowa: $D=X_1\times X_2\times\ldots\times X_n$
- zmienne programu: $z=\langle z_1, z_2, \ldots, z_k\rangle$
- przestrzeń programu: **(n+2k)-wymiarowa**

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

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

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



- **kryterium stopu:**

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

© Bogdan Wiszniewski, 2024

Przykład

```
                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:             }
```

zmienne wejściowe:

`symbol, number, count`

→ *n=3*

zmienne programu:

`number, count`

→ *k=2*

© Bogdan Wiszniewski, 2024

Przepływ danych

- **łańcuch prosty:**

  <definition,use>

  d: x = f(5)+3y

  ...

  u: z = 2x-y

- **łańcuch użycia:**

  <all-definition,use>

  d: x = f(5)+3y

  ...

  u: z = 2x-y

  ...

  a: w = x*z

- **łańcuch żywotności:**

  <all-definition,all-use>

  d: x = f(5)+3y

  ...

  u: z = 2x-y

  ...

  a: w = x*z

  ...

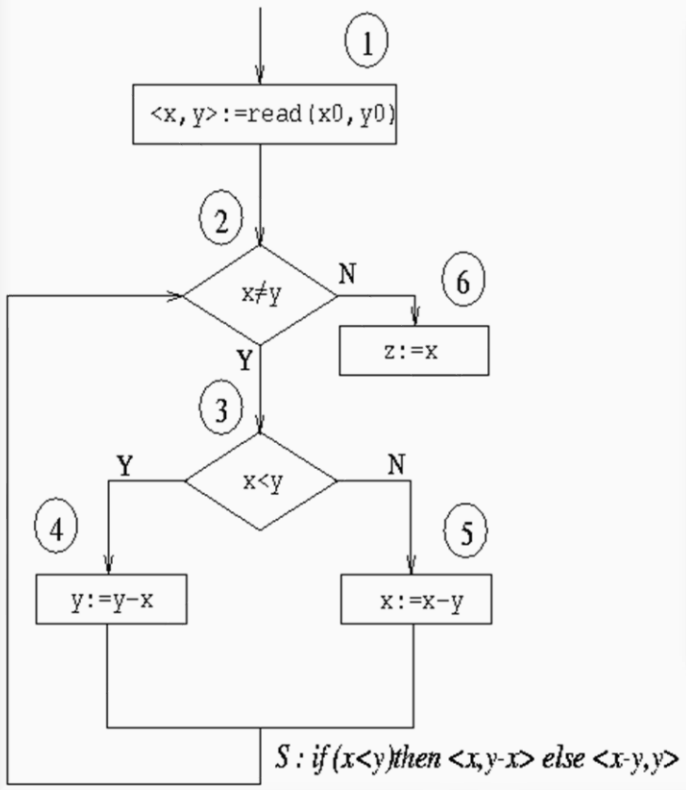
  b: y = x+w

kryterium

→ *wykonać każdy łańcuch*

© Bogdan Wiszniewski, 2024

Przykład



S : if (x<y)then <x,y-x> else <x-y,y>

łańcuchy "d-u" (proste):

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

łańcuchy "ad-u"(użycia):

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

łańcuchy "ad-au" (żywotności):

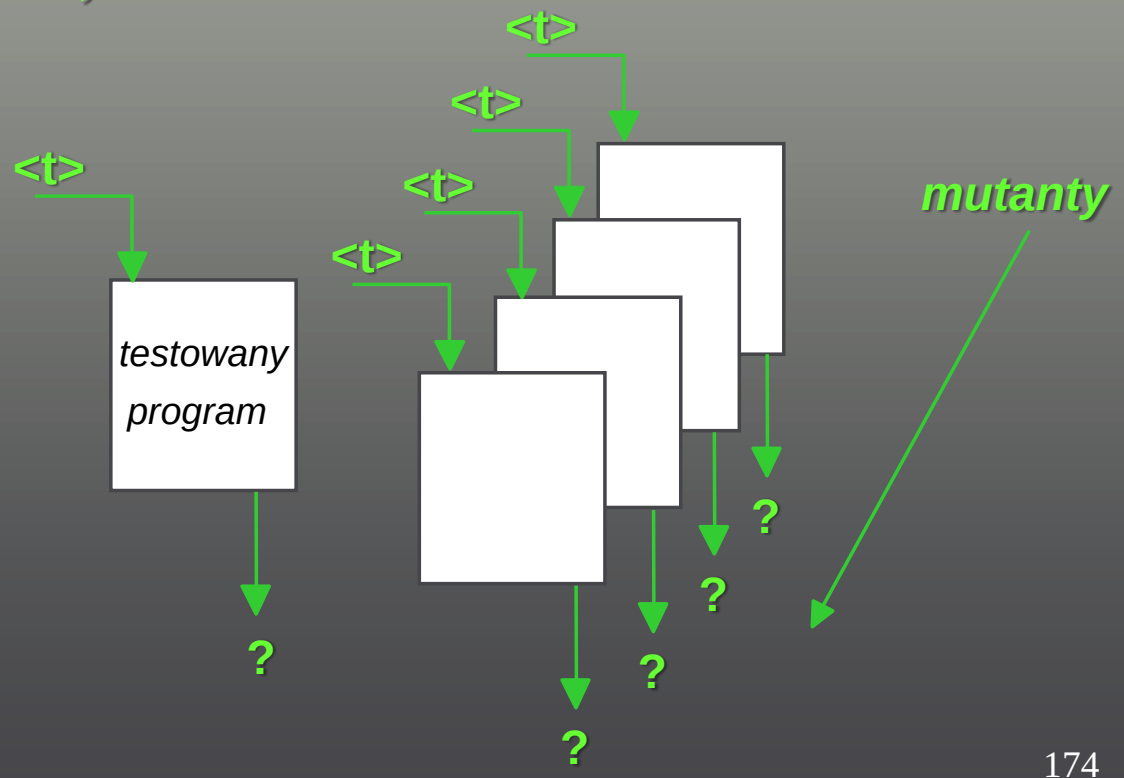
<1,4,6>, <1,5,6>, ...

© Bogdan Wiszniewski, 2024

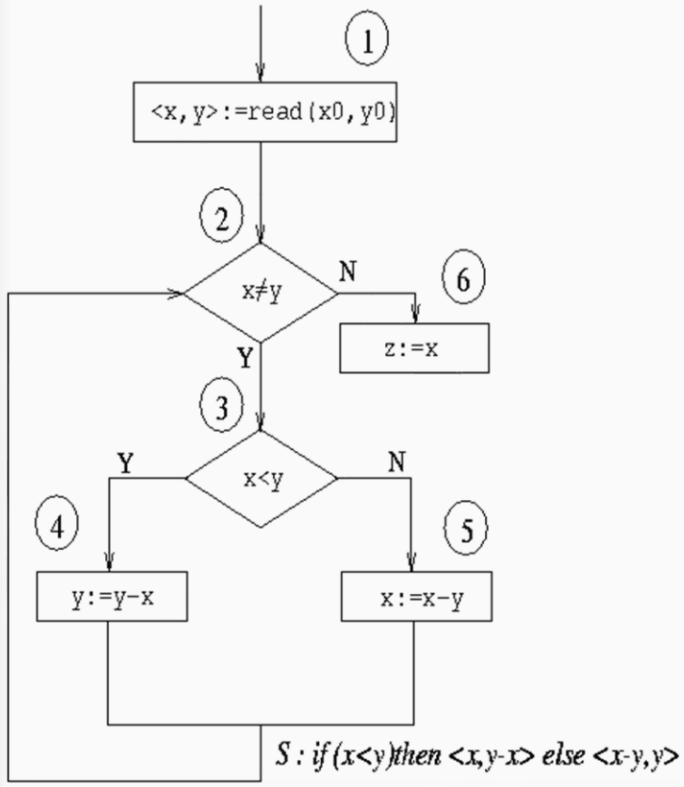
Testowanie mutacyjne

(!) błędy wynikają ze sporadycznych "przekłamań" w tekście programu

→ test harness, Monte-Carlo



Przykład



S : if (x<y)then <x,y-x> else <x-y,y>

M_1 → *while (x=y) ...*
M_2 → *if (x=y) ...*

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

© Bogdan Wiszniewski, 2024

Struktura przypadków testowych

- Dane wejściowe
- Spodziewane wyniki
- Wymagania na środowisko
- Powiązania ze scenariuszem

© Bogdan Wiszniewski, 2024

Skrypt testowy

```
/* 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;])
 >
}
```

rejestracja stanu

Skrypt testowy

```
/* 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 [])
 >
}
```

wymuszenie wartości

Atrybuty przypadków testowych

- **Reprezentatywność**
  - pojedynczy przypadek reprezentuje podzbiór
- **Wykonalność**
  - istnieją dane spełniające np. warunek wykonania ścieżki
- **Obserwowalność**
  - automat deterministyczny
- **Powtarzalność**
  - identyfikacja wszystkich zmiennych wejściowych (interpretacja warunku ścieżki)
  - kontrola uwarunkowań czasowych akcji

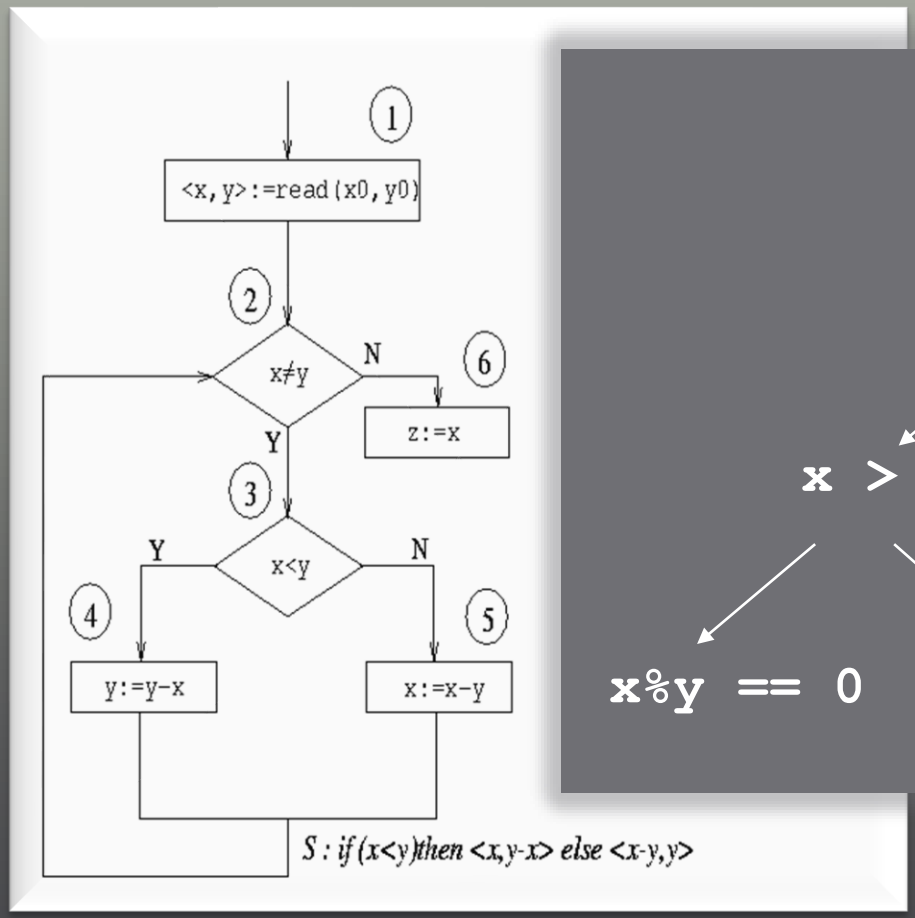
Wykonalność

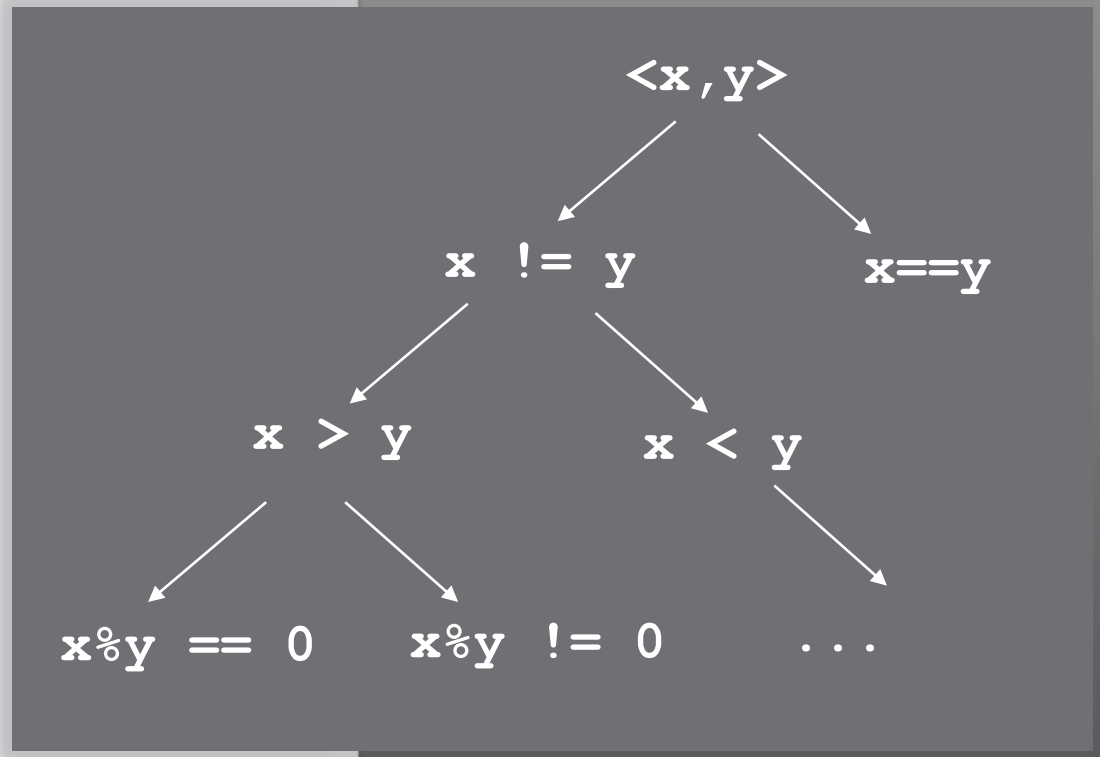
```
                int asynBCD(int number,int count){
1:  2           char symbol;
2:  3           for(;;)
3:  4           {for(;;)
4:  5           {receive(symbol);
5:  6 7         if((symbol==SPACE)||(symbol==STOP))
6:  11          break;
7:  8           count++;
8:  9 10        if(count>9)
9:   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

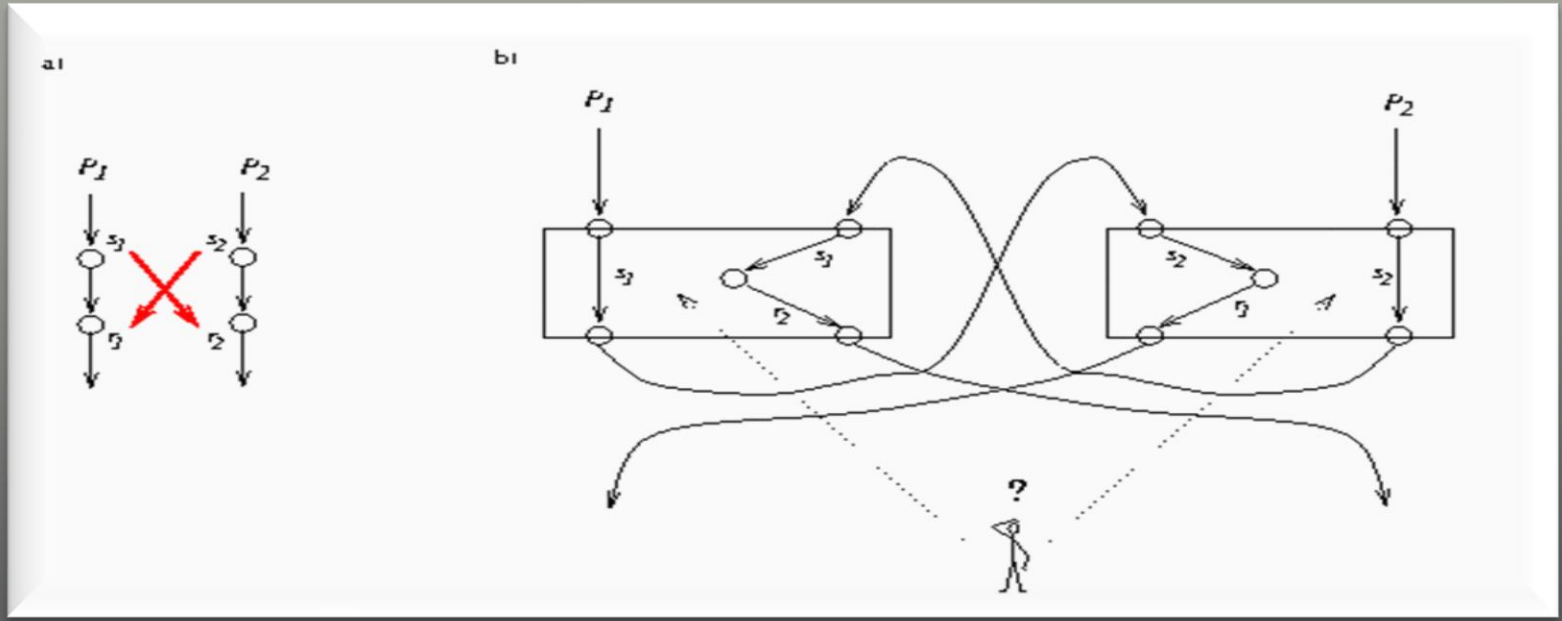
© Bogdan Wiszniewski, 2024

Reprezentatywność



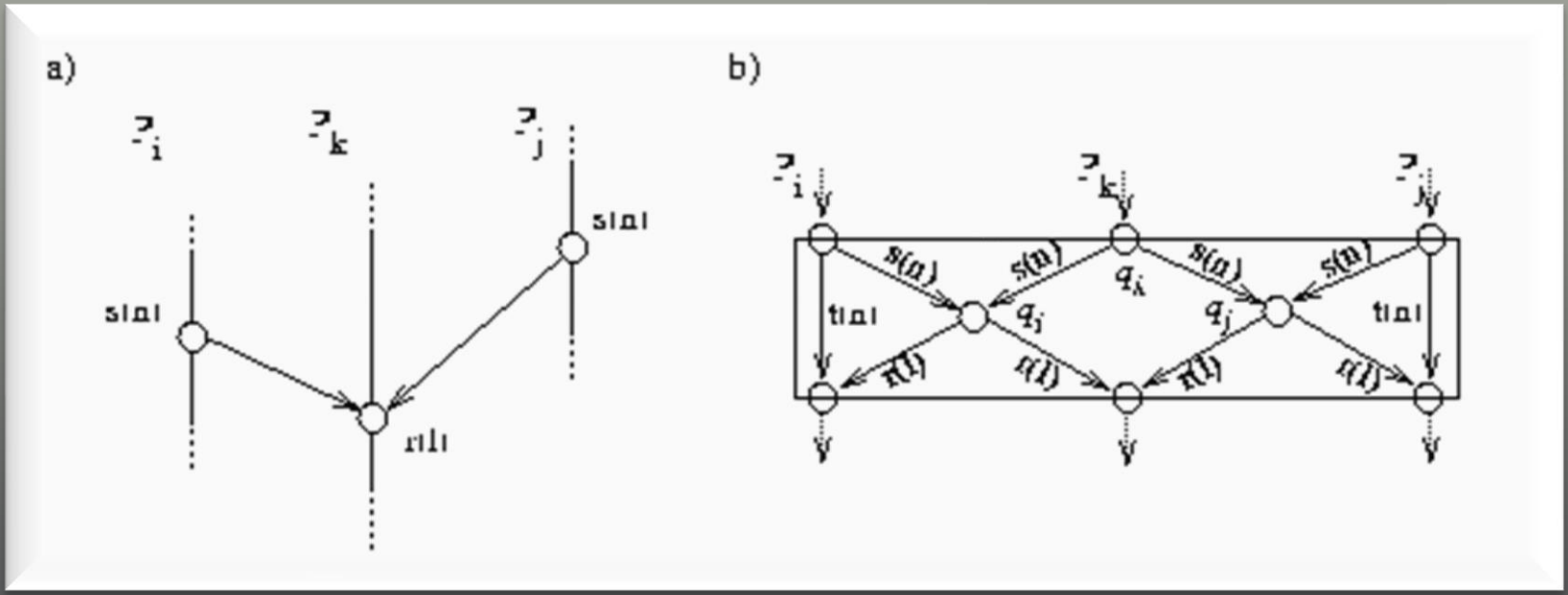


Obserwowalność



→ *błąd testowania*

Powtarzalność



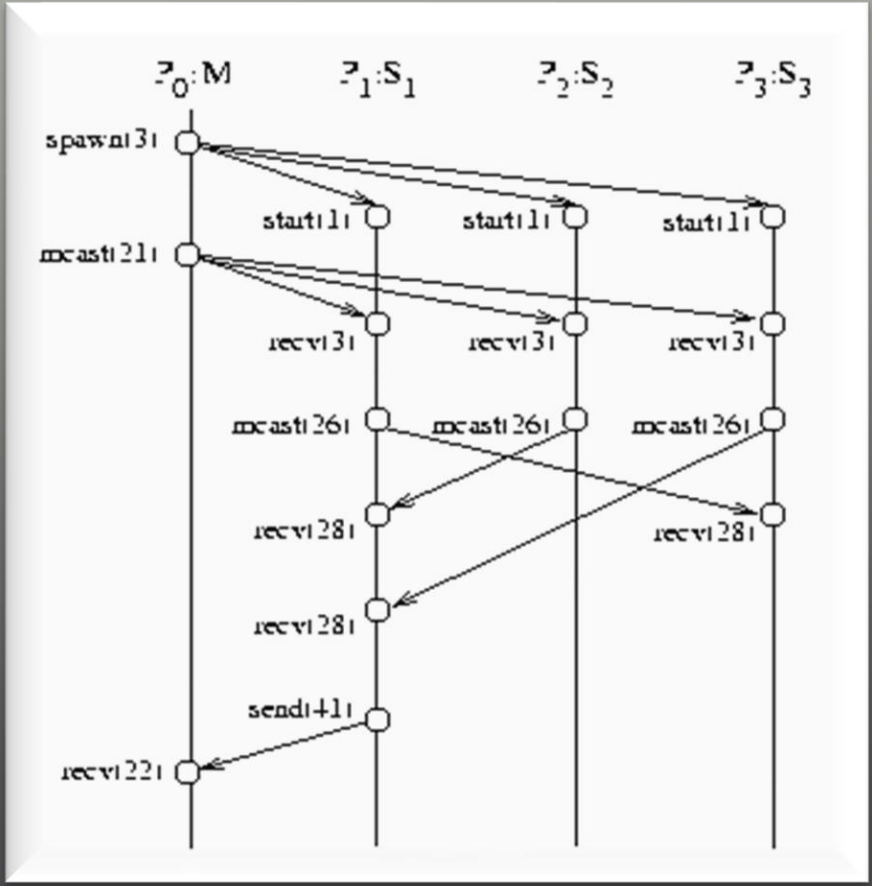
Rejestracja wyników

- **Punkt kontrolny**
  - statyczny (“wkompilowamy”)
  - dynamiczny (przerwanie)
- **Log**
  - scentralizowany
  - rozproszony
- **Analiza wyników**
  - on-line (detekcja stanu lub zdarzenia),
  - off-line (% wypełnienia testu, lokalizacja błędu)
  - replay (wizualizacja, odtwarzanie stanu)

© Bogdan Wiszniewski, 2024

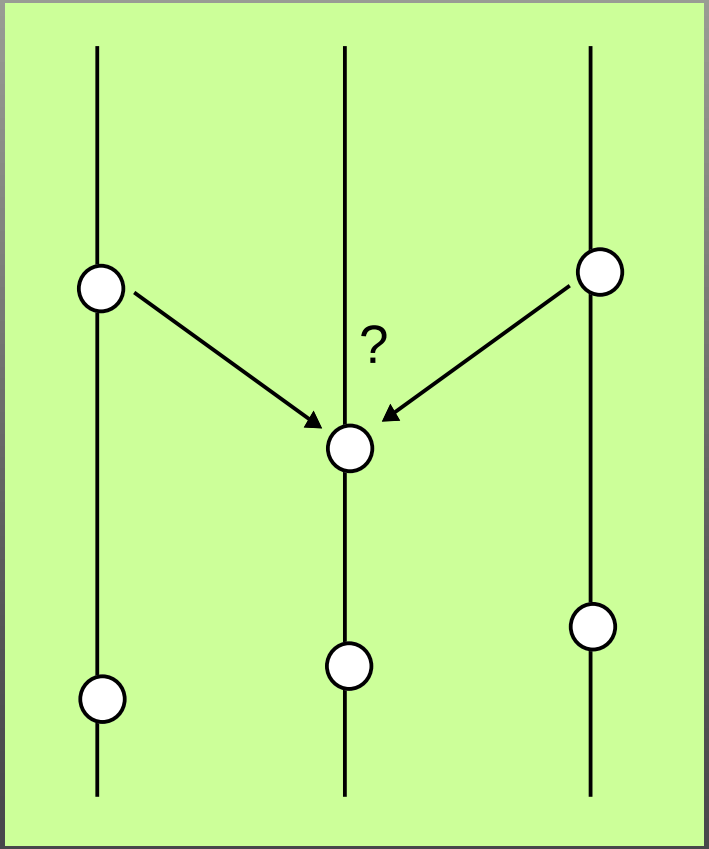
Tryby wykonania scenariuszy

- losowy
- nadzorowany
- deterministyczny



Scenariusze testowe

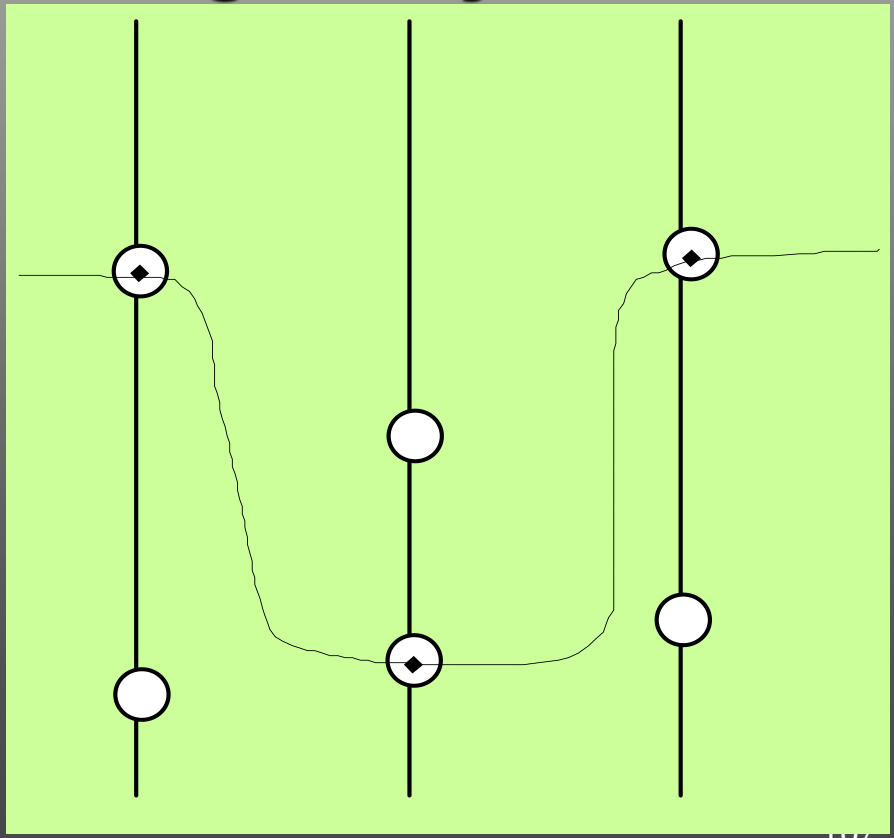
- **One-thread-One-time (OtOt):**
  wykrywanie wyścigów



© Bogdan Wiszniewski, 2024

Scenariusze testowe

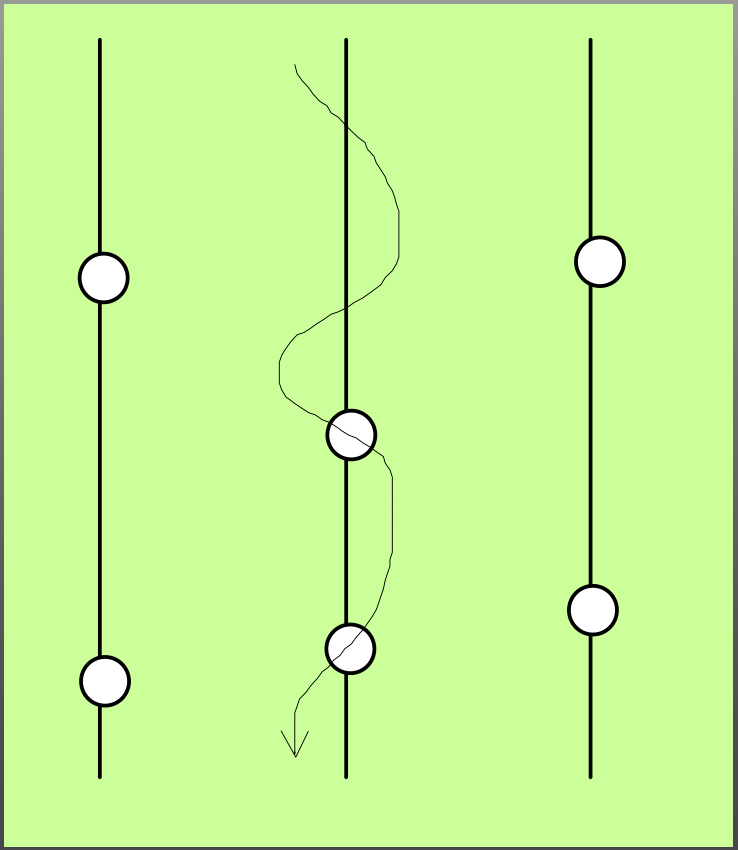
- **Many-threads-One-time (MtOt):**
  monitorowanie stanów globalnych



© Bogdan Wiszniewski, 2024

Scenariusze testowe

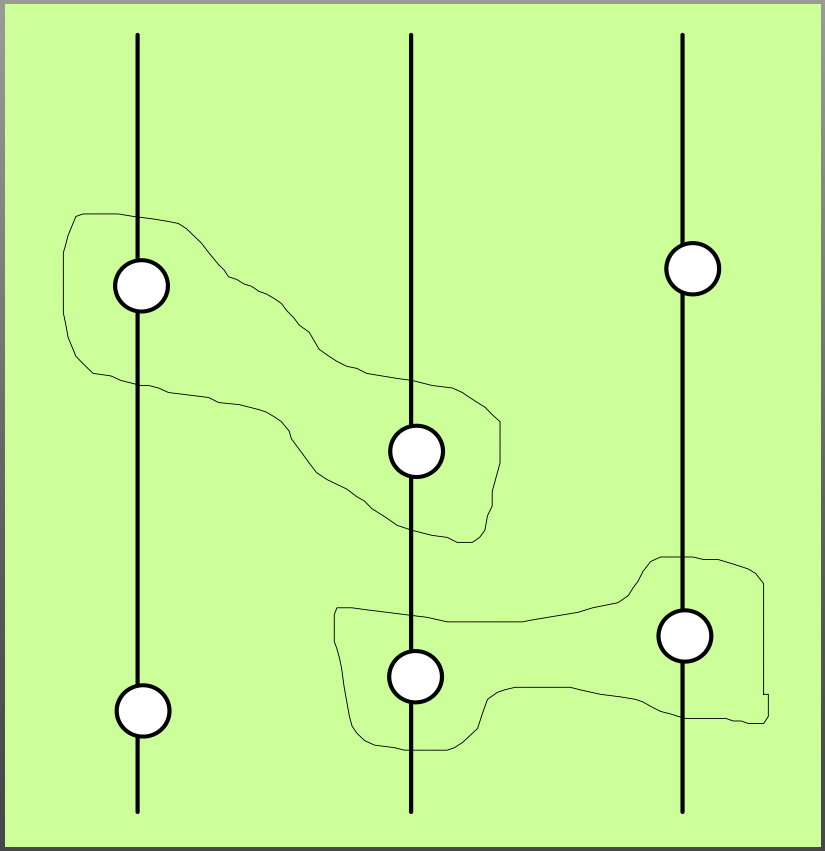
- **One-thread-Many-times (OtMt):**
  testowanie ścieżki procesu



© Bogdan Wiszniewski, 2024

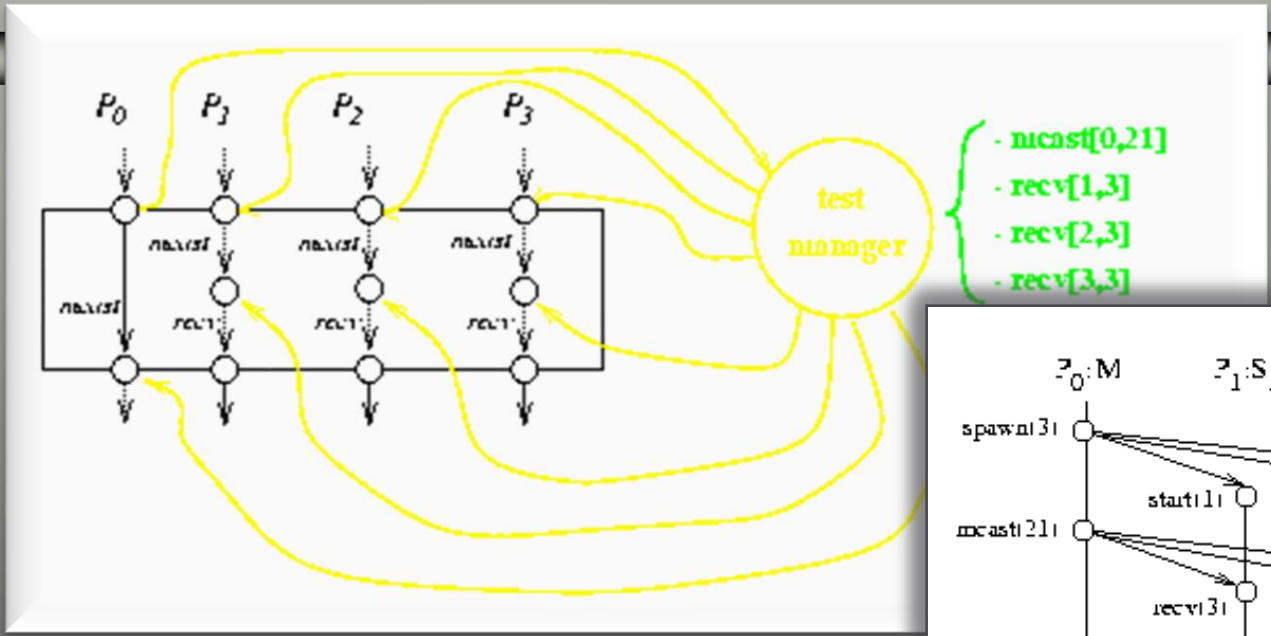
Scenariusze testowe

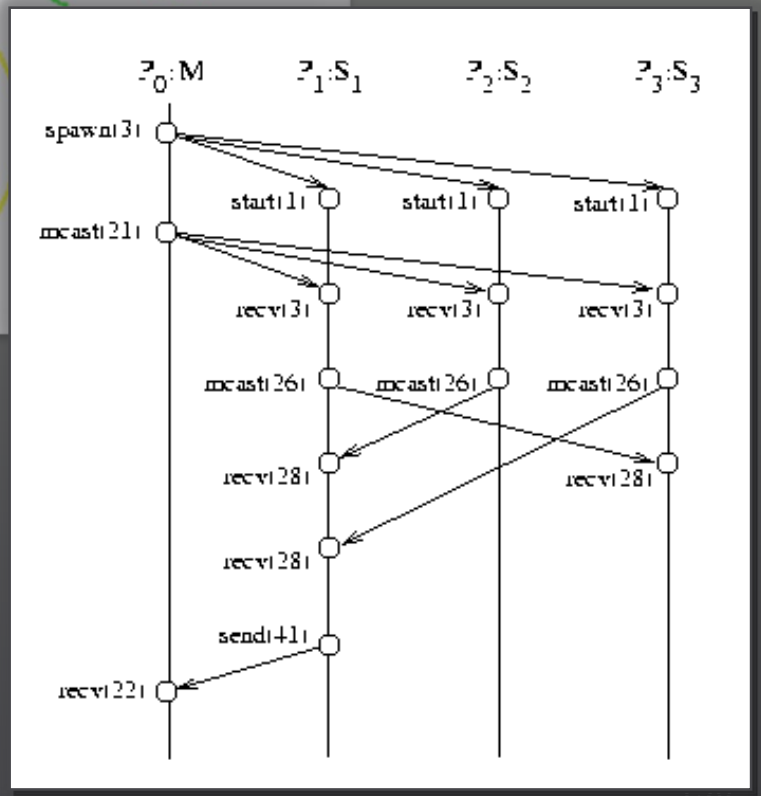
- **Many-threads-Many-times (MtMt):**
  monitorowanie zdarzeń



© Bogdan Wiszniewski, 2024

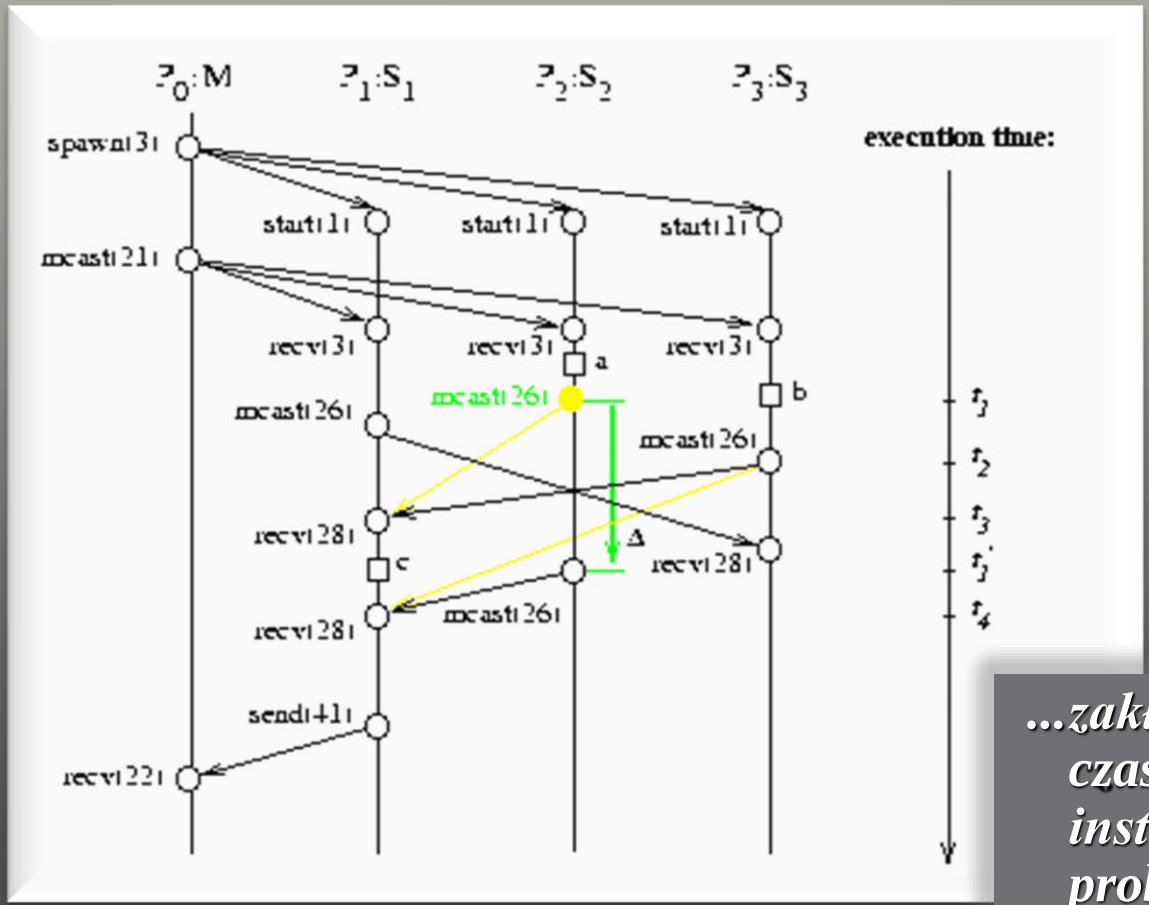
Instrumentacja kodu i środowiska





© Bogdan Wiszniewski, 2024

Efekt próbnika



...zakłócenie relacji czasowych przez instrumentację kodu (ang. probe-effect)

Struktura logu

- **Nagłówek:**
  - unikatowy identyfikator
  - komentarz
  - rekordy zapisu czynności i zdarzeń
- **Rekord:**
  - data, czas, ID przypadku testowego
  - zdarzenie, stan lokalny , kontekst
- **Zdarzenie:**
  - wykonana instrukcja, sygnał, wyjątek, zmiana wartości
- **Stan:**
  - zawartość pamięci obiektu
- **Kontekst:**
  - historia, warunek rejestracji, stan globalny

© Bogdan Wiszniewski, 2024

Co warto rejestrować?

- **Potencjalne miejsca występowania błędów:**
  - instrukcje arytmetyczne, podstawienia,
  - obliczanie warunku,
  - konwersja typu, parametry wywołania,
  - wywołania (i powroty z) funkcji/procedur,
  - zmienne dynamiczne,
  - obsługa diagnostyki systemowej,
  - pakowanie/rozpakowywanie komunikatów,
  - znakowanie komunikatów,
  - wyścigi,
  - dopasowanie akcji komunikacyjnych

© Bogdan Wiszniewski, 2024

Lokalizacja błędów

(!) Wiedzieć, że program ma błąd nie oznacza wiedzieć co go powoduje

- **Uruchamanie (ang. debugging):**
  - Post-mortem, zrzut (ang. core dump),
  - Log (ang. trace file)
  - Budowanie hipotezy
  - Eliminacja hipotez
- **Narzędzia:**
  - Wydruki kontrolne
  - Pułapka dynamiczna (ang. breakpoint trap)
  - Replay

Czy można bez testowania?

- Programiści popełniają błędy gdy tworząc program nie są w stanie zapamiętać wszystkich szczegółów potrzebnych do tego by uczynić go poprawnym
- Nie ma programów bez błędów, są tylko źle testowane
- Programy o dowiedzionej poprawności mogą zawierać błędy
- Prawidłowe zachowania możemy brać za błędne (i odwrotnie)
- Błędy ujawniają się przez cały czas życia programu

© Bogdan Wiszniewski, 2024