Functional, Control and Data Flow, and Mutation Testing: Theory and Practice

Part II - More practice than theory

Auri M. R. Vincenzi[1], Márcio E. Delamaro[2], Erika N. Höhn[3] and José C. Maldonado[3]

[1]Mestrado em Informática
Universidade Católica de Santos

[2]Faculdade de Informática
Centro Universitário Eurípides de Marília

[3]Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo

PSSE'2007 – 3rd to 6th December 2007

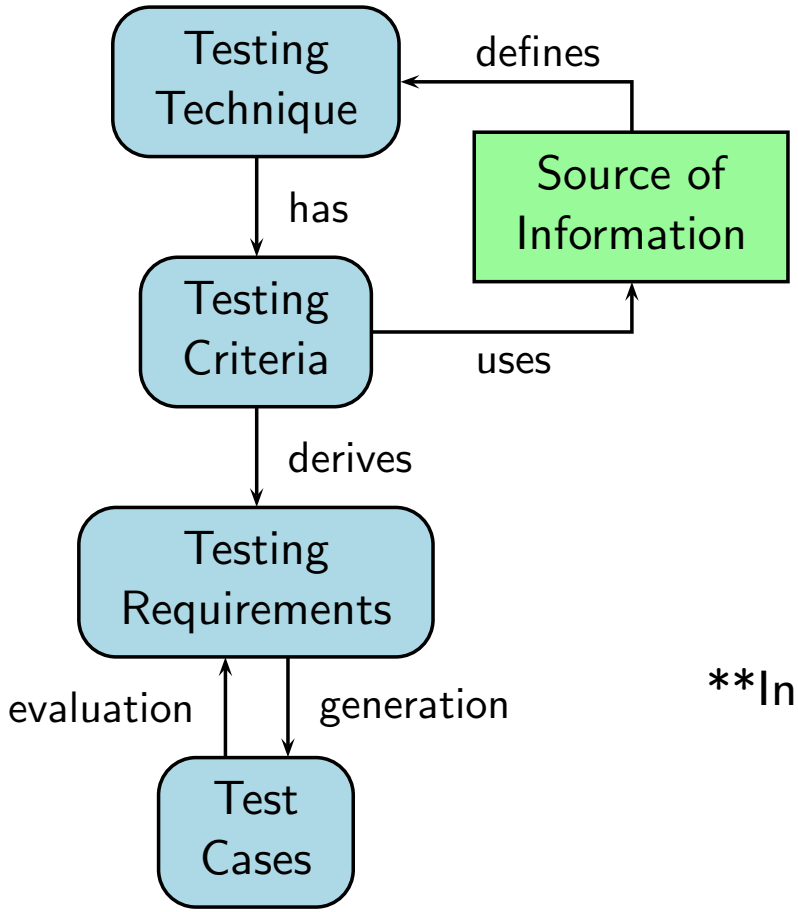
Definition (1)

Given a product P and a test set T, it is defined:

- **Adequacy criterion of test case**: predicate to evaluate T when testing P;
- **Selection method of test case**: procedure to choose test cases to test P.

Testing adequacy criterion is "a predicate that defines what properties of a program must be exercised to constitute a 'thorough' test" (Goodenough and Gerhart, 1975).

Definition (2)



**Infeasible requirements.

Minimal properties

The minimal properties which a testing criterion C should fulfill are (Maldonado, 1991):

1. To guarantee, from the control flow perspective, the coverage of all conditional deviations;
2. To require, from the data flow perspective, at least on use of all computational result; and
3. To require a finite test set.

The complexity of a testing criterion is defined as the maximum number of test cases required to satisfy it in the worst case.

Subsume relation

Establish a partial order between testing criteria.

"A criterion C_1 subsumes a criterion C_2 if for every program P and any test set T_1 C_1-adequate, T_1 is also C_2-adequate and there is a program P and a test set T_2 C_2-adequate such that T_2 is not C_1-adequate".

Evaluation aspects

These aspects are evaluated from the theoretical and experimental point of views.

Cost: the required effort to use the criterion.

Efficacy: the capacity of the criterion in detecting faults.

Strength: the probability of satisfying a given criterion C_2 after to satisfy a criterion C_1.

Overview

- The name **functional** or **black-box** testing is due to the technique considers the product under test (PUT) as a box of which are only known its inputs and outputs.
- Unlike its complement, white box testing, black box testing requires no knowledge of the internal paths, structure, or implementation of the PUT.
- Black box testing criteria generate testing requirements based on the product specification.

Functional Testing Process

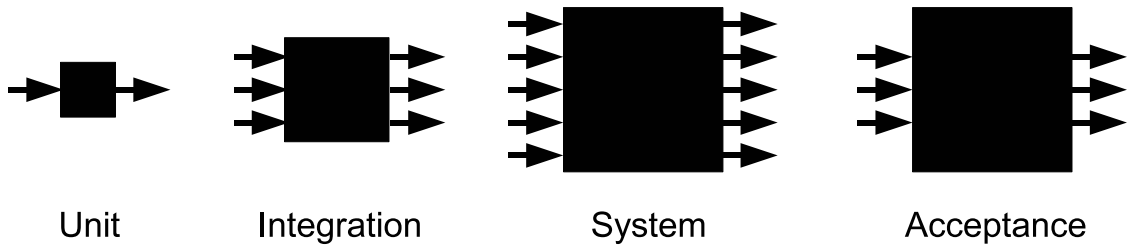
- The basic steps of application are:
  1. The requirements or specifications are analyzed.
  2. Valid inputs are chosen based on the specification to determine that the PUT processes them correctly. Invalid inputs must also be chosen to verify that the PUT detects them and handles them properly.
  3. Expected outputs for those inputs are determined.
  4. Tests are constructed with the selected inputs.
  5. The tests are run.
  6. Actual outputs are compared with the expected outputs.
  7. A determination is made as to the proper functioning of the PUT.

Some Functional Testing Criteria

- Equivalence Partition.
- Boundary Value Analysis.
- Decision Table.
- Pairwise Testing.
- State-Transition Testing.
- Domain Analysis Testing.
- Use Case Testing.
- Systematic Functional Testing.

Functional Testing *versus* Testing Phases

- Since it is an implementation independent testing technique, black box testing criteria can be applied at any of the testing phases.
- As we move up in size from module to subsystem to system the box gets larger, with more complex inputs and more complex outputs, but the approach remains the same.
- Also, as we move up in size, we are forced to the black box approach; there are simply too many paths through the PUT to perform white box testing.



Functional testing applicability.

Disadvantages of the Functional Testing

- It is dependent on very good software specification (which, in general, it is not well elaborated).
- It is not possible with black box criteria to determine whether essential parts of the software have been executed.
- To find every defect using black box testing, every possible combination of input data is required, both valid and invalid. This **exhaustive testing** is almost always impossible. What is the probability to select a test case to execute this "feature"?

```
1 if (name=="Lee" && employeeNumber=="1234" &&
2     employmentStatus=="RecentlyTerminatedForCause") {
3     send Lee a check for $1,000,000;
4 }
```

  - A worse case: how to test a compiler program? Is it possible to execute it with all possible inputs? How many programs can be written for a given compiler? (Myers, 1979).

Advantages of the Functional Testing

- It can be used at any testing phase.
- It is programming-paradigm independent.
- Systematic black box testing directs the tester to choose subsets of tests that are both efficient and effective in finding defects. Better than random testing (Copeland, 2004).
- It is effective to determine some kinds of faults:
  - Missing functionalities, for instance.

Other Examples

- Others examples of the Equivalence Partition criterion can be found elsewhere (Beizer, 1995; Copeland, 2004; Myers et al., 2004).

Identifier Program (1)

Program specification (extracted from (Maldonado et al., 2004)):

The program determines if a given identifier is valid or not in a variant of Pascal language, called Silly Pascal. A valid identifier must begin with a letter and contain only letters or digits. Moreover, it has at least one and no more than six character length.

Samples of identifiers:

abc12 (valid);
cont*1 (invalid); 1soma (invalid); a123456 (invalid)

Identifier Program (2)

Equivalence classes for the `Identifier` program:

Input conditions	Valid Classes	Invalid Classes	
Identifier's size t	$1 \leq t \leq 6$ (1)	$t < 1$ (2)	$t > 6$ (3)
First character c is a letter	Yes (4)	No (5)	
Only contains valid characters	Yes (6)	No (7)	

Test set example:

$T_{\text{Equivalence Partition}} = \{$(a1, Valid), ("", Invalid), (A1b2C3d, Invalid), (2B3, Invalid), (Z#12, Invalid)$\}$

Identifier Program (3)

Go to practice:

- Testing `Identifier` without JUnit.
- Testing `Identifier` with JUnit.

Automation Support

- JUnit is not the only tool to support the documentation and automatic execution of test cases.
- Considering the most popular programming languages, there are others similar tools, such as:
  - TestNG (http://testng.org/) for Java.
  - DUnit (http://dunit.sourceforge.net/) for Delphi.
  - cUnit (http://sourceforge.net/projects/cut/) for C.
  - Jeté (http://jete.sourceforge.net/) integration testing for Java.
  - And others. An extensive list can be found elsewhere http://www.testingfaqs.org/ and http://www.opensourcetesting.org/.

Overview

- Structural or White Box Testing is the opposite of Functional Testing.
  - Testing is based on the internal paths, structure, and implementation of the product under test (PUT).
  - Structural testing generally requires detailed programming skills.

Structural Testing Process

- The basic steps of application are:
  1. The PUT's implementation is analyzed.
  2. Paths through the PUT are identified.
  3. Inputs are chosen to cause the PUT to execute selected paths. This is called path sensitization.
  4. Expected results for those inputs are determined.
  5. The tests are run.
  6. Actual outputs are compared with the expected outputs.
  7. A determination is made as to the proper functioning of the PUT.

Structural Testing Criteria

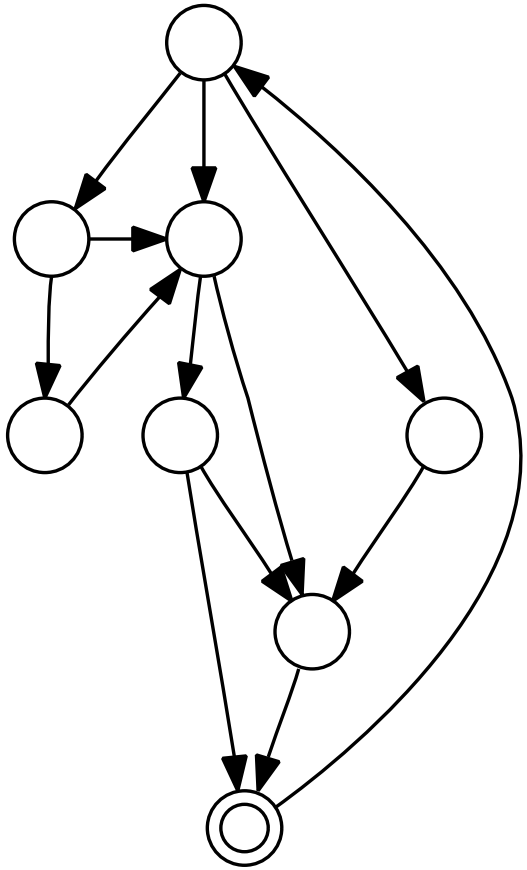
- Control Flow Based Testing Criteria.
  - All-Nodes, All-Edges, All-Paths
- Data Flow Based Testing Criteria.
  - All-Defs, All-P-Uses, All-C-Uses, All-Uses, All-Du-Paths, All-Pot-Uses, All-Pot-Du-Paths, All-Pot-Uses/Du.

Structural Testing *versus* Testing Phases (1)

- White box testing can be applied at all levels of system development-unit, integration, and system.
- White box testing may involve:
  - Paths that are tested are within a module (unit testing).
  - Paths between modules within subsystems.
  - Paths between subsystems within systems.
  - Paths between entire systems.

Structural Testing *versus* Testing Phases (2)

Different paths within the PUT.



Disadvantages of the Structural Testing

- The number of execution paths may be so large that they cannot all be tested.
- The test cases chosen may not detect data sensitivity errors. For instance:

  ```
  y = x * 2; // should read y = x ** 2
  ```

  will pass for test cases `x = 0, y = 0` and `x = 2, y = 4`.
- White box testing assumes the control flow is correct (or very close to correct). Since the tests are based on the existing paths; nonexistent paths cannot be usually discovered through white box testing.
- In general, it is necessary to determine infeasible paths. Impossible to be fully automated.
- The tester must have the programming skills to understand and evaluate the software under test.

Advantages of the Structural Testing

- Efficacy on determine logical or programming faults in the PUT, specially at the unit level.
- It is possible to determine whether critical or essential paths were executed by the test set.
  - Minimal testing requirement: to ensure that every statement in the software was executed at least once by the test set before the software to be released.

Definition (1)

- Control flow based testing is one of the approaches within structural testing technique.
- Identifies the execution paths inside a module (testing requirements) and then creates and executes test cases to cover those paths.
- Definition:
  - **Path**: A sequence of statements that begins at an entry and ends at an exit.

Drawbacks (1)

- Control flow paths has a number of significant drawbacks:
  - The number of paths could be huge and thus untestable within a reasonable amount of time. Every decision doubles the number of paths and every loop multiplies the paths by the number of iterations through the loop. For example:

```
1 for (i=1; i<=1000; i++)
2    for (j=1; j<=1000; j++)
3       for (k=1; k<=1000; k++)
4          doSomethingWith(i,j,k);
```

    executes `doSomethingWith()` one billion times (1000 x 1000 x 1000). Each unique path deserves to be tested.
  - Paths called for in the specification may simply be missing in the module:

```
1 if (a>0) doIsGreater();
2 if (a==0) doIsEqual();
3   // missing statement - if (a<0) doIsLess();
```

Drawbacks (2)

- More problems...
  - Defects may exist in processing statements within the module even through the control flow itself is correct.

```
1 // actual (but incorrect) code
2 a=a+1;
3 // correct code
4 // a=a-1;
```

  - The module may execute correctly for almost all data values but fail for a few:

```
1 int blech (int a, int b) {
2    return a/b;
3 }
```

    fails if b has the value 0 but executes correctly if b is not 0.

- Even though control flow testing has a number of drawbacks, it is still a vital tool in the tester's toolbox. And it is complementary to its counterpart (Black Box Testing).

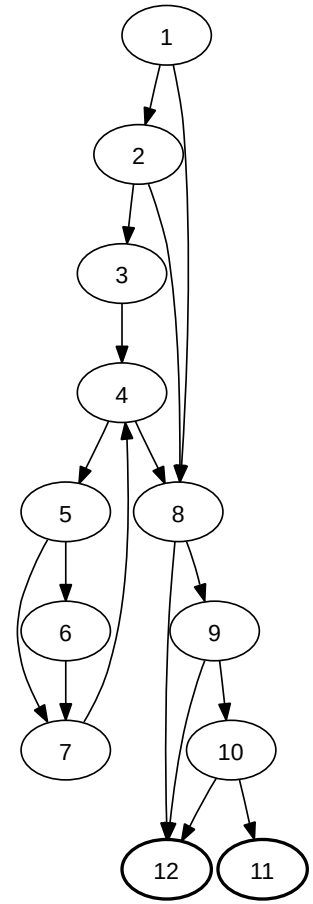
Control Flow Graph (1)

- Abstraction of a PUT, representing its control flow.
- Directed graph composed by nodes and edges.
- Each **node** represents one or more statements that are all executed in sequence, i.e., once the first statement within a node is executed all the others in the same node are also executed. There is no control flow deviation to any of this statements, except to the first one.
- An **edge** represents the control flow existent between nodes.

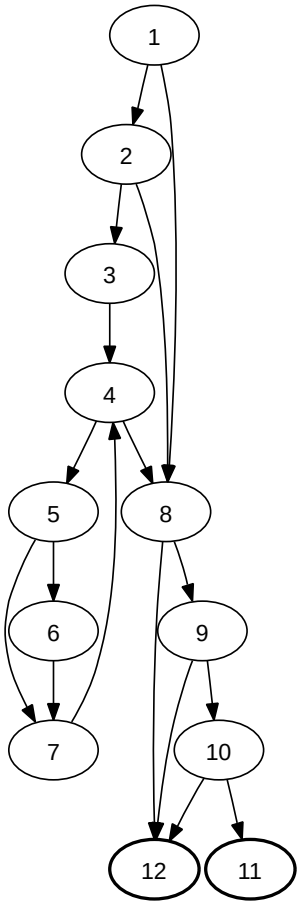
Control Flow Graph (2)

Example:

```
public boolean validateIdentifier(String s) {
        char achar;
/*01*/  boolean valid_id = false;
/*01*/  if (s.length() > 0) {
/*02*/      achar = s.charAt(0);
/*02*/      valid_id = valid_s(achar);
/*02*/      if (s.length() > 1) {
/*03*/          achar = s.charAt(1);
/*03*/          int i = 1;
/*04*/          while (i < s.length() - 1) {
/*05*/              achar = s.charAt(i);
/*05*/              if (!valid_f(achar))
/*06*/                  valid_id = false;
/*07*/              i++;
                  }
              }
          }
                  /*08*/              /*09*/                  /*10*/
          if (valid_id && (s.length() >= 1) && (s.length() < 6))
/*11*/      return true;
          else
/*12*/      return false;
}
```



Control Flow Graph (3)



- All-Nodes criterion
  - (`1#@`, Invalid) – passes through nodes (1,2,3,4,5,6,7,4,8,12)
  - (`i`, Valid) – passes through nodes (1,2,8,9,10,11)
- $T_{\text{All-Nodes}} = \{$(`1#@`, Invalid),(`i`, Valid)$\}$ is All-Nodes-adequate.
- But such a test set is not All-Edges-adequate: edges $(1, 8)$, $(5, 7)$, $(9, 12)$ and $(10, 12)$ are not executed by any test case in $T_{\text{All-Nodes}}$.
- Edge $(9, 12)$ is infeasible.
- $T_{\text{All-Edges}} = T_{\text{All-Nodes}} \cup \{$(`A1b2C3d`, Invalid))$\}$ is All-Edges-adequate.

Identifier Program

Go to practice: evaluating the coverage of $T_{\text{Equivalence Partition}}$ with respect to All-Nodes and All-Edges criteria.

- $T_{\text{Equivalence Partition}}$ = {(a1, Valid), ("", Invalid), (A1b2C3d, Invalid), (2B3, Invalid), (Z#12, Invalid)}
- Testing `Identifier` with Emma.
  - All-Nodes criterion.
- Testing `Identifier` with JaBUTi.
  - All-Edges criterion (All-Edges$_{ei}$ criterion).

Others Similar Tools

- Emma is not the only tool to support the application of control flow based testing criteria.
- Considering the most popular programming languages, there are others similar tools, such as:
  - Cobertura (http://cobertura.sourceforge.net/).
  - EclEmma (http://www.eclemma.org/) Emma plug-in for Eclipse.
  - TCAT (http://www.soft.com/TestWorks) for C/C++ and Java.
  - JavaCov (http://www.alvicom.hu/) for Java.
  - And others. An extensive list can be found elsewhere http://www.testingfaqs.org/ and http://www.opensourcetesting.org/.

Definition (1)

- Data flow testing is another approach within structural testing technique.
- It complements the control flow testing criteria.
- Aims at to detecting faults related with the definition and use of variables in a program, i.e., its target is the flow of data instead of the flow of control of a program.

Definition (2)

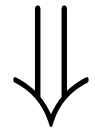
- Data flow testing is a powerful approach to detect improper use of data values due to coding mistake.
- Rapps and Weyuker (1982) disseminated this approach. They say:

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

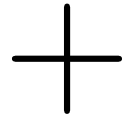
Definition (3)

Principles of the Criteria Definition (Martins, 2003)

Program = sequence of actions on variables



Control Flow



Data Flow

(Information about statements defining and using variables)

Definition (4)

Types of Uses:

- There are two kinds of variable **use**:
  - Use in computations, called **computational use** (c-use). For instance: `a = b * 1`.
  - Use in conditions, called **predicative use** (p-use). For instance: `if (a >= b)`.
- In both uses it is equally important that the variable has been assigned a value (**defined**) before it is used.
  - The **definition** of a variable occurs when a value is assigned to it. For example: `a = 10` and `b = 5`.

Definition-Use Graph (1)

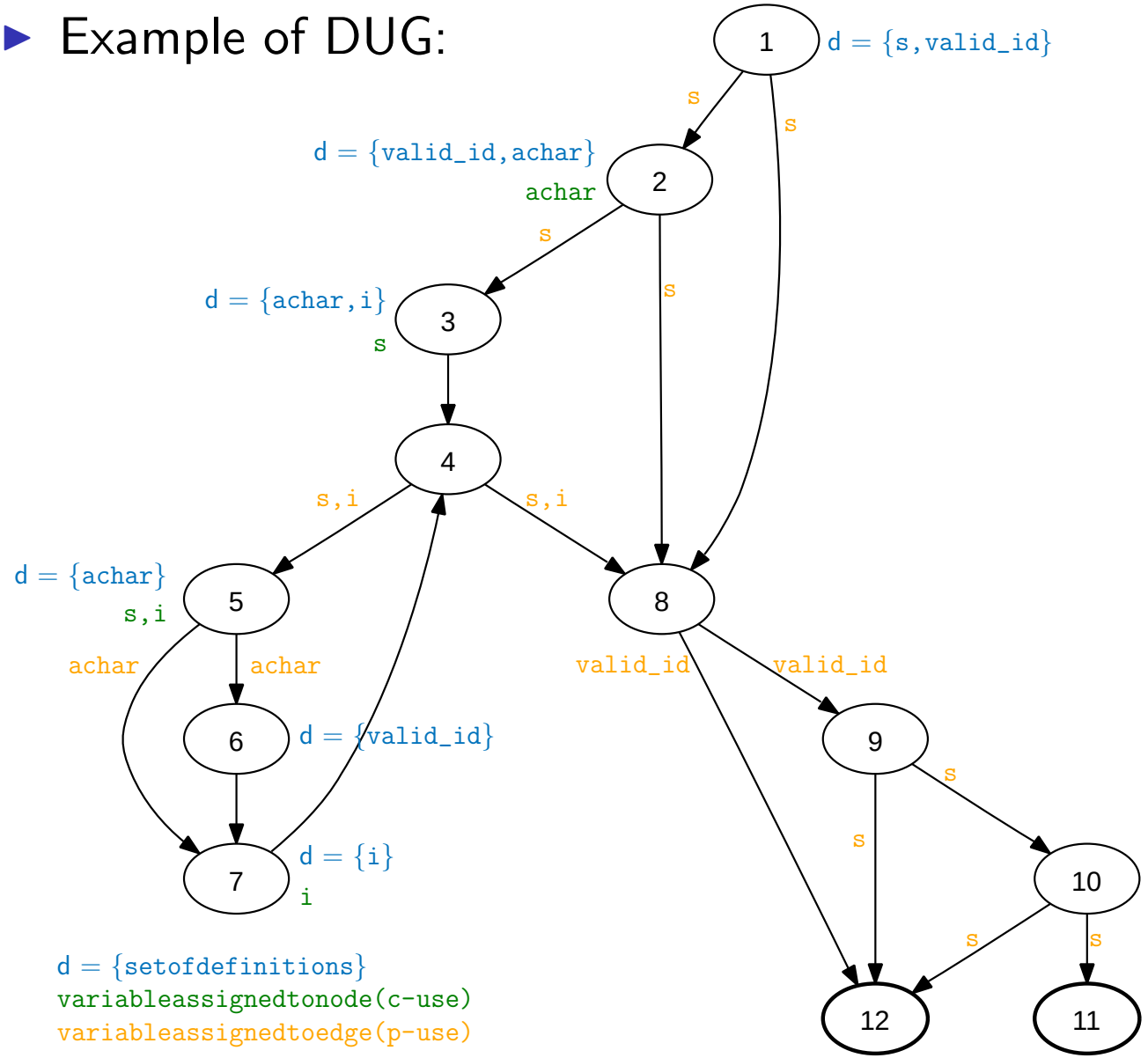
- To represent the different status of a variable in a program, a abstract representation, called Definition-Use Graph (DUG) (Rapps and Weyuker, 1982), was developed.
- It is an extension of the Control Flow Graph.
- It includes information about variables definition, uses and destructions on each node.
- Static and Dynamic Analysis of DUG:
  - Static: examine the diagram (formally through inspections or informally through look-sees).
  - Dynamic: construct and execute test cases.

Definition-Use Graph (2)

```
public boolean validateIdentifier(String s) {
       char achar;
/*01*/ boolean valid_id = false;
/*01*/ if (s.length() > 0) {
/*02*/    achar = s.charAt(0);
/*02*/    valid_id = valid_s(achar);
/*02*/    if (s.length() > 1) {
/*03*/       achar = s.charAt(1);
/*03*/       int i = 1;
/*04*/       while (i < s.length() - 1) {
/*05*/          achar = s.charAt(i);
/*05*/          if (!valid_f(achar))
/*06*/             valid_id = false;
/*07*/          i++;
             }
          }
       }
             /*08*/          /*09*/                /*10*/
       if (valid_id && (s.length() >= 1) && (s.length() < 6))
/*11*/    return true;
       else
/*12*/    return false;
}
```

Definition-Use Graph (3)

- Example of DUG:



Definition-Use Graph (4)

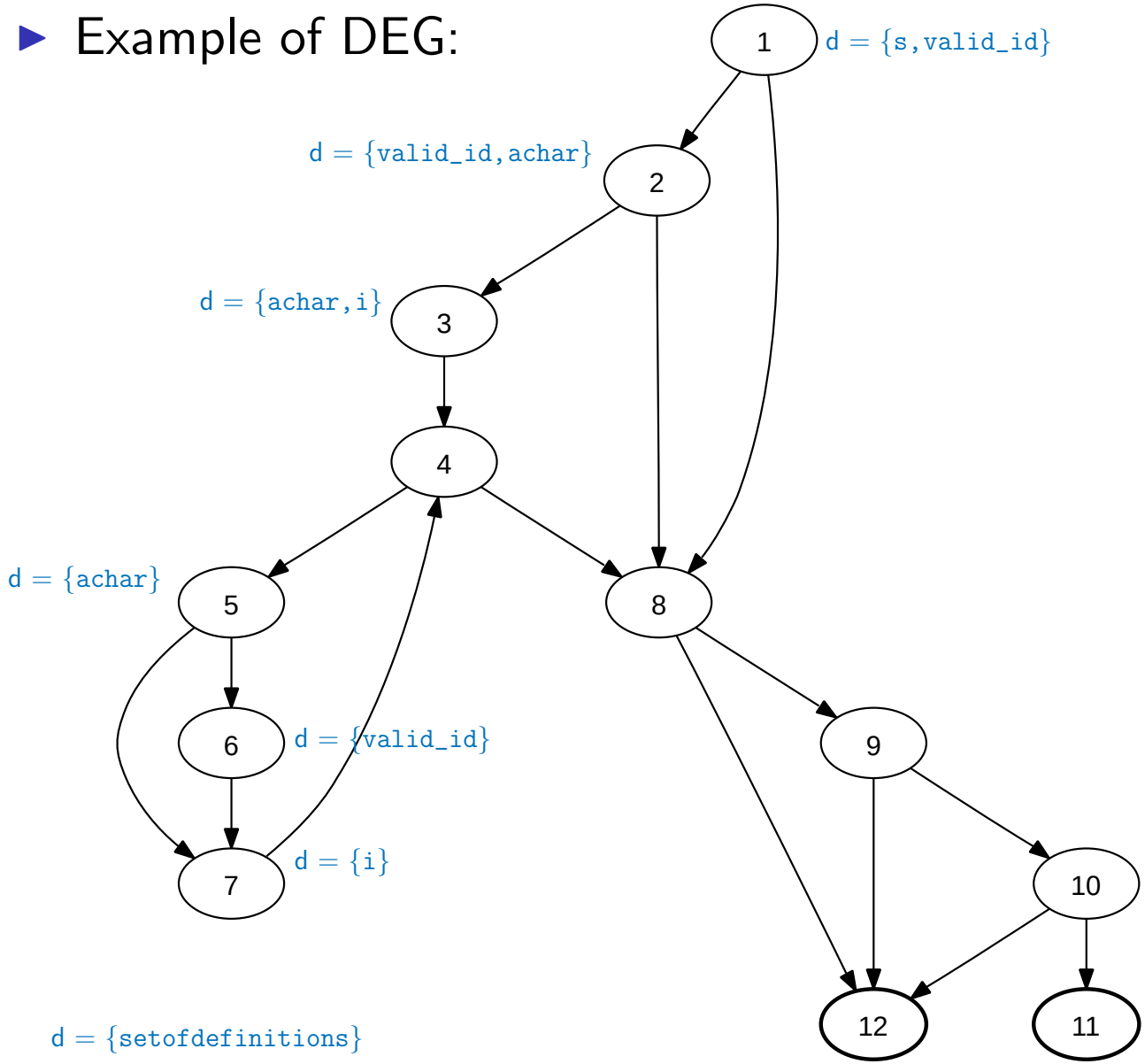
- Example of **data flow associations**:
  - $\langle$`s`, 1, 3$\rangle$, $\langle$`valid_id`, 1, (8, 9,)$\rangle$, and $\langle$`valid_id`, 1, (8, 12,)$\rangle$.
  - $\langle$`valid_id`, 1, (8, 9,)$\rangle$ is infeasible.
- Example of **definition-clear path**:
  - Path (1,8,12) is a def-clear path with respect to `valid_id` defined at node 1: covers $\langle$`valid_id`, 1, (8, 12,)$\rangle$.
  - Path (1,2,8,12) is not a def-clear path with respect to `valid_id` defined at node 1, because `valid_id` is redefined at node 2.

Rapps & Weyuker family criteria

- The most basic data flow based criteria is the All-Defs criterion which is part of the Rapps & Weyuker family criteria (Rapps and Weyuker, 1985).
- Among the others criteria of such a family the most used and investigated criterion is the All-Uses criterion.
  - **All-Defs**: requires that a data flow association for each variable definition to be exercised, at least once, by a def-clear path with respect to a c-use or p-use.
  - **All-Uses**: requires that all data flow associations between a variable definition and all its subsequence uses (c-uses and p-uses) to be exercised by at least one def-clear path.

Definition Graph (1)

- Example of DEG:



Potential Uses Criteria (1)

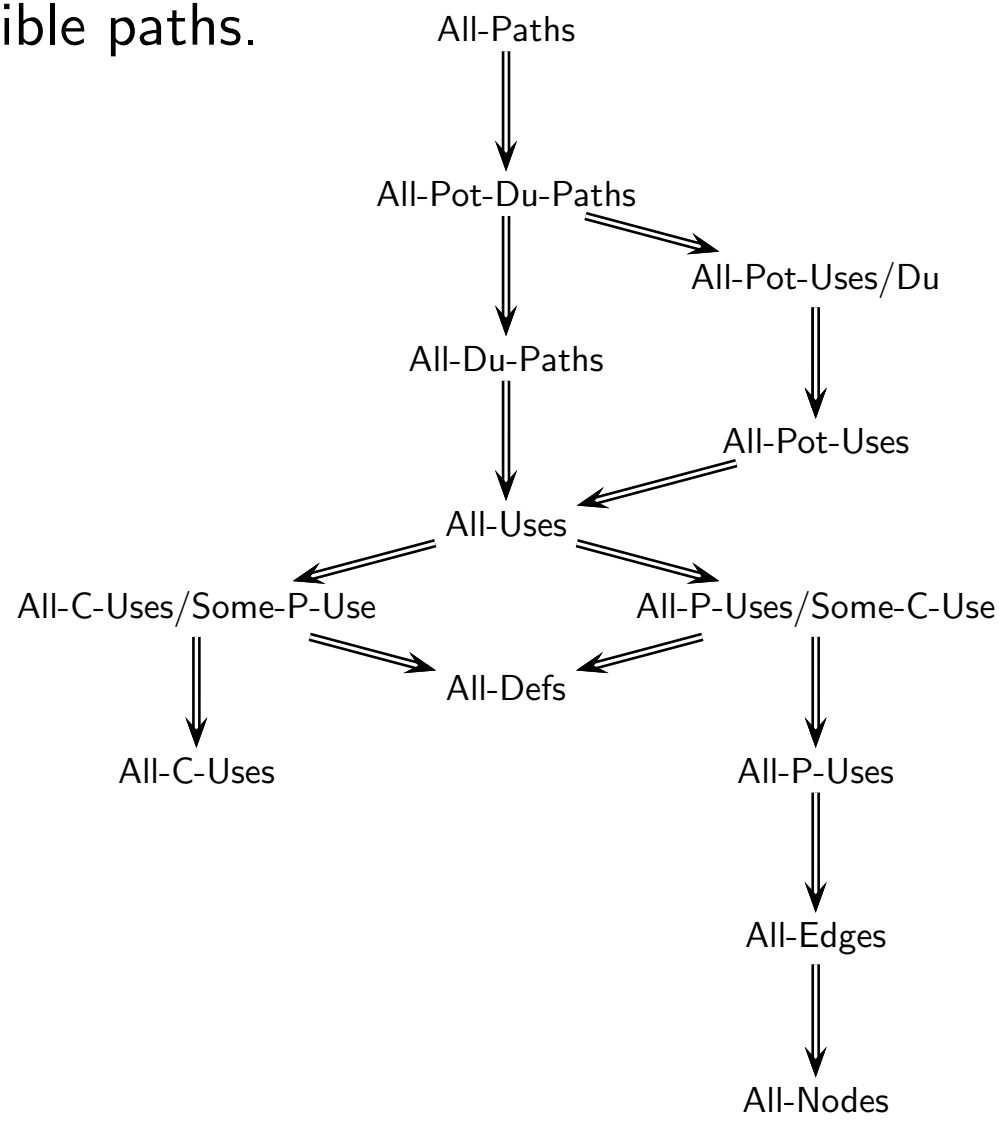
- Based on the concept of **potential-association**:
  - Associations are established without a explicit use.
- They requires only the Definition Graph (Def-Graph): CFG + Definitions.

Potential Uses Criteria (2)

- The most basic Potential Uses criteria (Maldonado, 1991) is All-Pot-Uses.
  - **All-Pot-Uses**: requires for each node i containing a definition of a variable x that to all node and edge that can be reached from i by a def-clear path with respect to x to be exercised.
- **Potential uses associations** $\langle \mathtt{s}, 1, 6 \rangle$, $\langle \mathtt{achar}, 3, (8, 9) \rangle$, and $\langle \mathtt{achar}, 3, (8, 12) \rangle$, for instance, are required by the All-Pot-Uses criterion but are not required by the other data flow based criterion.

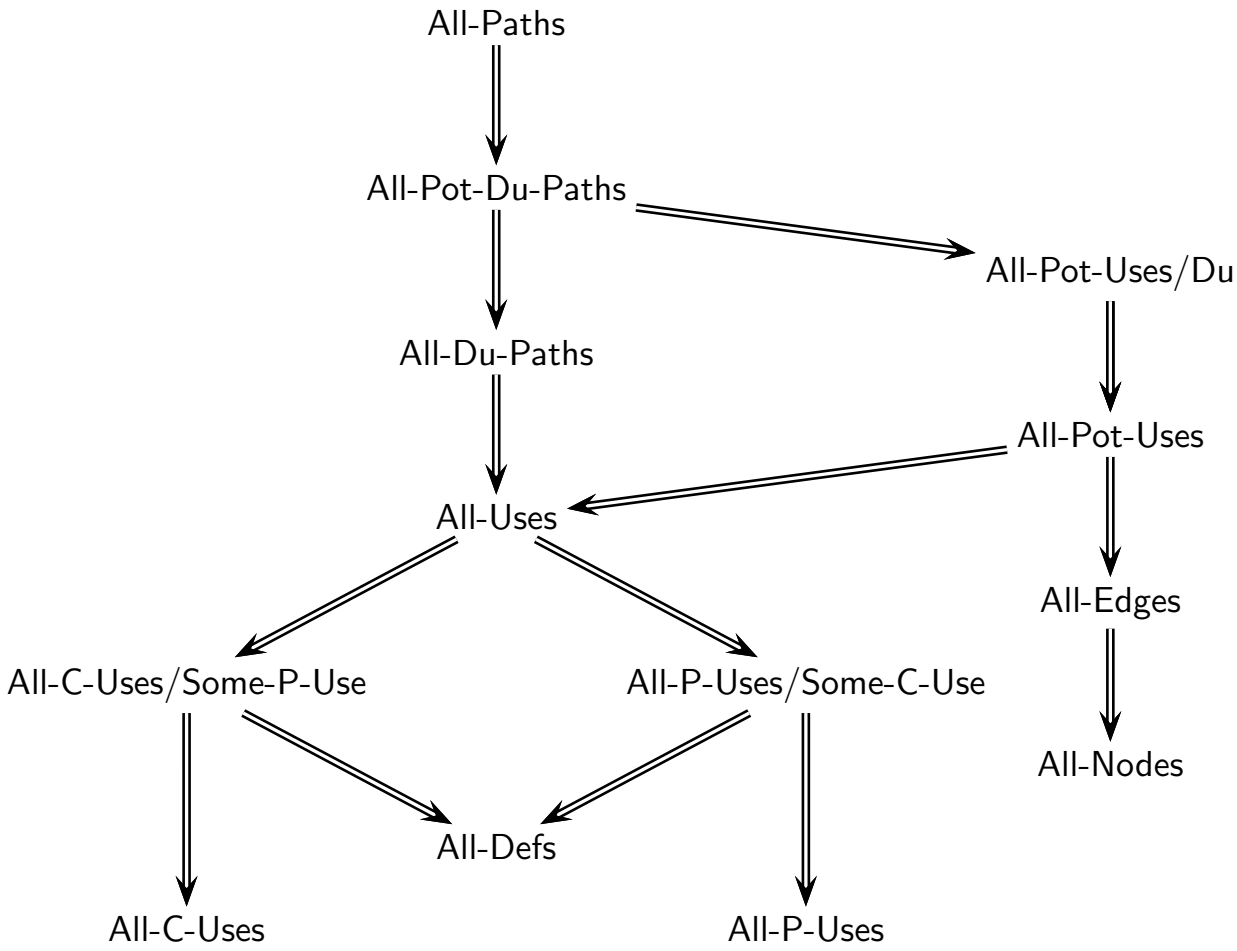
Subsume Relation (1)

- Relation between control and data considering only feasible paths.



Subsume Relation (2)

- Relation between control and data considering infeasible paths.



Identifier Program

Go to practice: evaluating the coverage of $T_{\text{All-Nodes}}$ and $T_{\text{All-Edges}}$ with respect to All-Uses and All-Pot-Uses$_{ei}$ criteria.

- $T_{\text{All-Nodes}_{ei}} = T_{\text{Equivalence Partition}} \cup \{(\texttt{""}, \text{Invalid}), (\texttt{a1}, \text{Valid})\}$.
- $T_{\text{All-Edges}_{ei}} = T_{\text{All-Nodes}_{ei}} \cup \{(\texttt{c}, \text{Valid}), (\texttt{\textbackslash\{}, \text{Invalid}), (\texttt{a\textbackslash\{b}, \text{Invalid})\}$.

- Testing `Identifier` with JaBUTi.
  - All-Uses$_{ei}$ and All-Pot-Uses$_{ei}$ criteria.

Others Similar Tools

- There are few tools which supports the automation of data flow based testing criteria.
- Many of them are prototype.
  - *xSuds* (http://xsuds.argreenhouse.com/) for C and C++.
  - JaBUTi (http://incubadora.fapesp.br/projects/jabuti/ for Java.
  - Poke-Tool (http://incubadora.fapesp.br/projects/poketoo for C, Fortran, and Cobol.
  - Coverlipse (http://coverlipse.sourceforge.net/) for Java.
  - You may find others by searching elsewhere http://www.testingfaqs.org/, http://java-source.net/, http://www.opensourcetesting.org/, and other repositories.

Auri M. R. Vincenzi, Márcio E. Delamaro, Erika N. Höhn and José C. Maldonado

Fault-based Testing Criteria

- Error Seeding (Budd, 1981)
- Mutation Analysis (DeMillo et al., 1978)
- Interface Mutation (Delamaro et al., 2001)

Definition (1)

- The basic idea behind the technique is **the competent programmer hypothesis**: a good programmer writes correct or close-to-correct programs.
  - Assuming this hypothesis is valid: errors are introduced in a program through small syntactic deviations (faults) that lead its execution to an incorrect behavior.
  - Mutation Testing applying small changes to the PUT.
  - The tester to construct test cases that show that such modifications create incorrect products (Agrawal et al., 1989).

Definition (2)

- A second hypothesis explored by the Mutation Testing is the **coupling effect** (DeMillo et al., 1978).
  - Complex errors are composition of simple ones.
  - Test sets which reveal simples faults are also able to reveal complex faults (Budd, 1980).
  - A single mutation is applied in the program P under test, i.e., each mutant has a single syntactic transformation, relative to the original program.

Definition (3)

Given a product under testing P and a test set T to be evaluated. Steps of application:

1. Original product execution
   - P is executed against T.
   - If a failure occurs, the test is over.
   - If no failure occurs, P still can have hidden faults that T is not able to reveal.
2. Mutant generation
   - P is submitted to a set of **mutation operators** which transform P into $P_1, P_2, \ldots, P_n$ called **mutants** of P.

Mutation operators are rules that model the most frequent faults or syntactic deviation related to a given programming language.

Definition (4)

4. Mutant execution
   - The mutants are executed against the same test set T.
   - **Dead mutants** - results are different from P.
   - **Live mutants** - results are the same as P.
   - The ideal situation would be all the mutants dead: T is adequate for testing P.
5. Live mutant analysis
   - Live mutants are analyzed for identifying possible equivalence with P or a weakness of the test set T.

Definition (5)

Live mutant analysis

- Equivalent mutant
  - A mutant M is said equivalent to a product P if for all input data $d \in D$ we observe that $M(d) = P(d)$.
- Fault-reveling mutant
  - A mutant is said fault-revealing if for any test case t such that $P(t) \neq M(t)$ we can conclude that $P(t)$ is not according to the expected result, i.e., the presence of a fault is revealed.

Definition (6)

- Mutation score
  - Objective measure to evaluate the test set adequacy against Mutation Testing.

$$ms(P,T) = \frac{DM(P,T)}{M(P) - EM(P)}$$

where

$DM(P,T)$: number of mutants killed by the test set T;

$M(P)$: total number of mutants;

$EM(P)$: number of mutants equivalent to P.

Disadvantage

- The main problem with this criterion is the high number of generated mutants.
  - Mutants have to be compiled and executed.
  - Live mutants have to be analyzed for possible equivalence.
- Requires good knowledge about the product implementation to ease the task of analyzing live mutants.

Advantage

- It is easier to be extended to any "executable" product at specification or implementation level.
- It is one of the most effective testing criteria in detecting faults.

Identifier Program

Go to practice: evaluating the coverage of $T_{\text{All-Pot-Uses}_{ei}}$.

- $T_{\text{All-Pot-Uses}_{ei}} = T_{\text{All-Edges}_{ei}} \cup \{(\%\%,\text{Invalid}), (\%\%\%a,\text{Invalid})\}$.
- Testing `Identifier` (C version) with *PROTEUM/IM* 2.0.
  - Using a subset of the Unit Mutation Operators (Sufficient Set).

Operator	Description
u-SSDL	Removes a statement from the program.
u-ORRN	Replaces a relational operator.
u-VTWD	Replaces the reference to a scalar by its predecessor and successor.
u-Ccsr	Replaces the reference to a scalar by a constant.
u-SWDD	Replaces a `while` by a `do-while`.
u-SMTC	Breaks a loop execution after two executions.
u-OLBN	replaces a logical operator by a bitwise operator.
u-Cccr	Replace a constant by another.
u-VDTR	Forces each reference to a scalar to be: negative, positive and zero.

Others Similar Tools

- There are some tools which supports the automation of Mutation Testing criteria.
  - *PROTEUM/IM* 2.0 (Delamaro et al., 2000) for C (unit and integration level).
  - MuJava (http://ise.gmu.edu/~ofut/mujava/) for Java.
  - Jester (http://jester.sourceforge.net/) for Java.
  - Pester (http://jester.sourceforge.net/) for Python.
  - Nester (http://nester.sourceforge.net/) for C#
  - Mothra (http://www.ise.gmu.edu/~ofut/rsrch/mut.html) for Fortran.
  - Insure++ (http://www.parasoft.com/) for C and C++.
  - Other mutation tools can be found elsewhere http://www.mutationtest.net/.

Experimental Software Engineering

- World-wide View
  - **CeBase** - NSF Center for Empirically Based Software Engineering
    http://www.cebase.org/
  - **ISERN** - International Software Engineering Research Network
    http://www.iese.fhg.de/ISERN/
  - **ESELAN** - Experimental Software Engineering Latin-American Network
    http://listas.cos.ufrj.br/mailman/listinfo/eselan-

Experimental studies

- Primary studies
  - Survey
    - Obtaining qualitative and quantitative data: interviews or questionnaires.
    - Do not have control of the running or of the measures.
  - Study Case
    - Detailed information is collected during a continuous period of time.
    - As a study observational, the control's level is less than a controlled experiment.
  - Controlled Experiment
    - High level of control.
    - Performed when a control of the situation is necessary, manipulating the behavior of study directly and systematically.
- Secondary studies
  - Systematic Reviews.
    - Collect, summarize and analyze data obtained in primary studies.

An Experimental Methodology for Software Processes Validation

Incremental evaluation of a new process

- Approach for evolving processes, from the early concept phase to the tailoring and use of the process on an industrial project;
- Building-up of a body of evidence concerning process effectiveness using different types of studies to address different questions of interest;
- Approach can be helpful for a responsible interaction with industry.

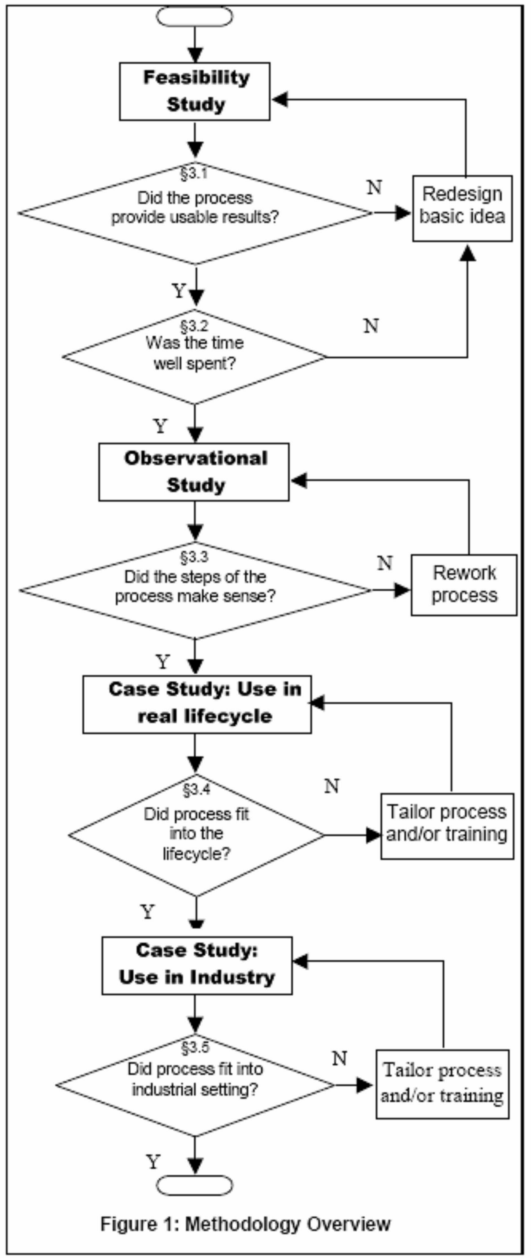
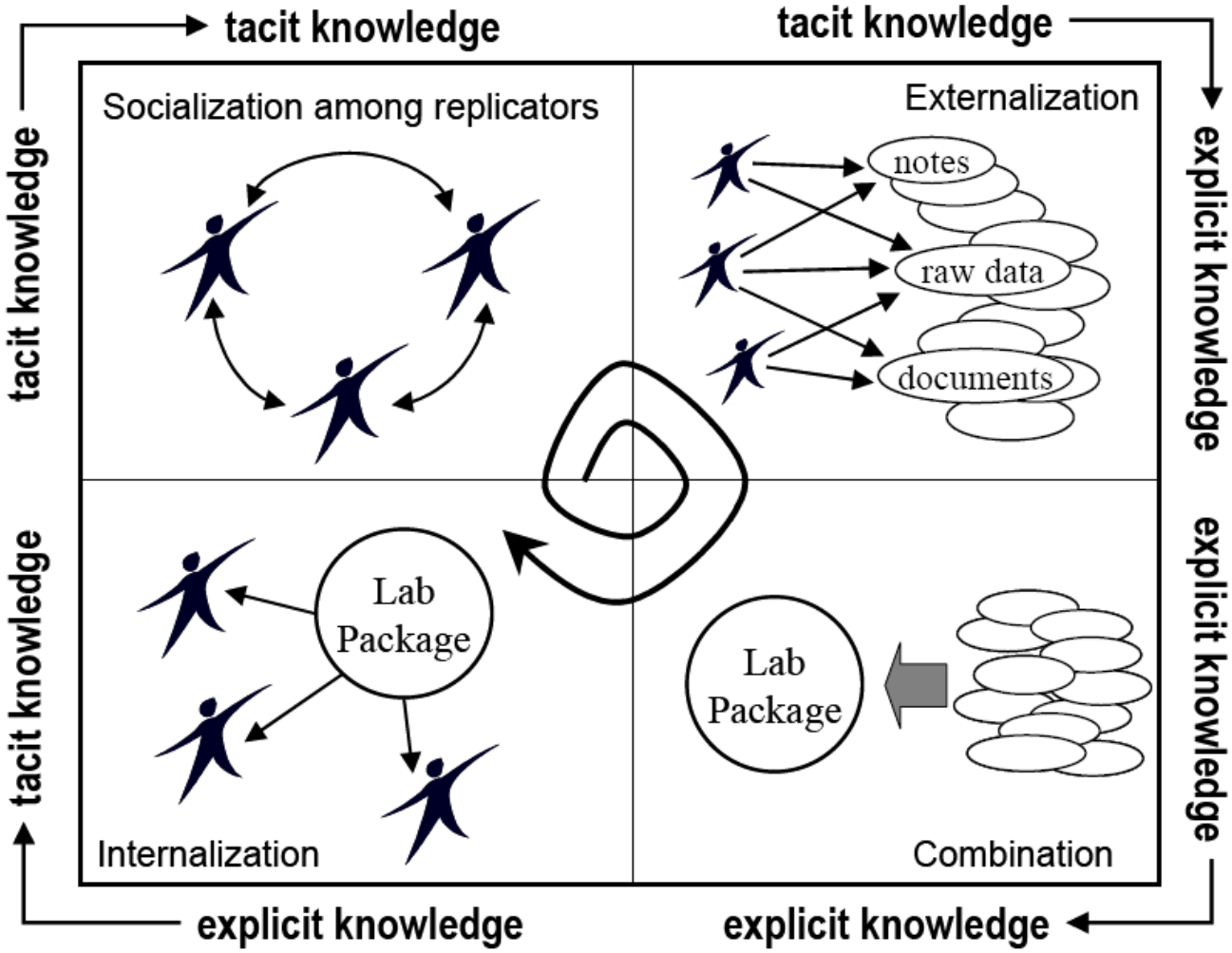


Figure 1: Methodology Overview

Experimental Studies in Software Engineering: Lessons Learned

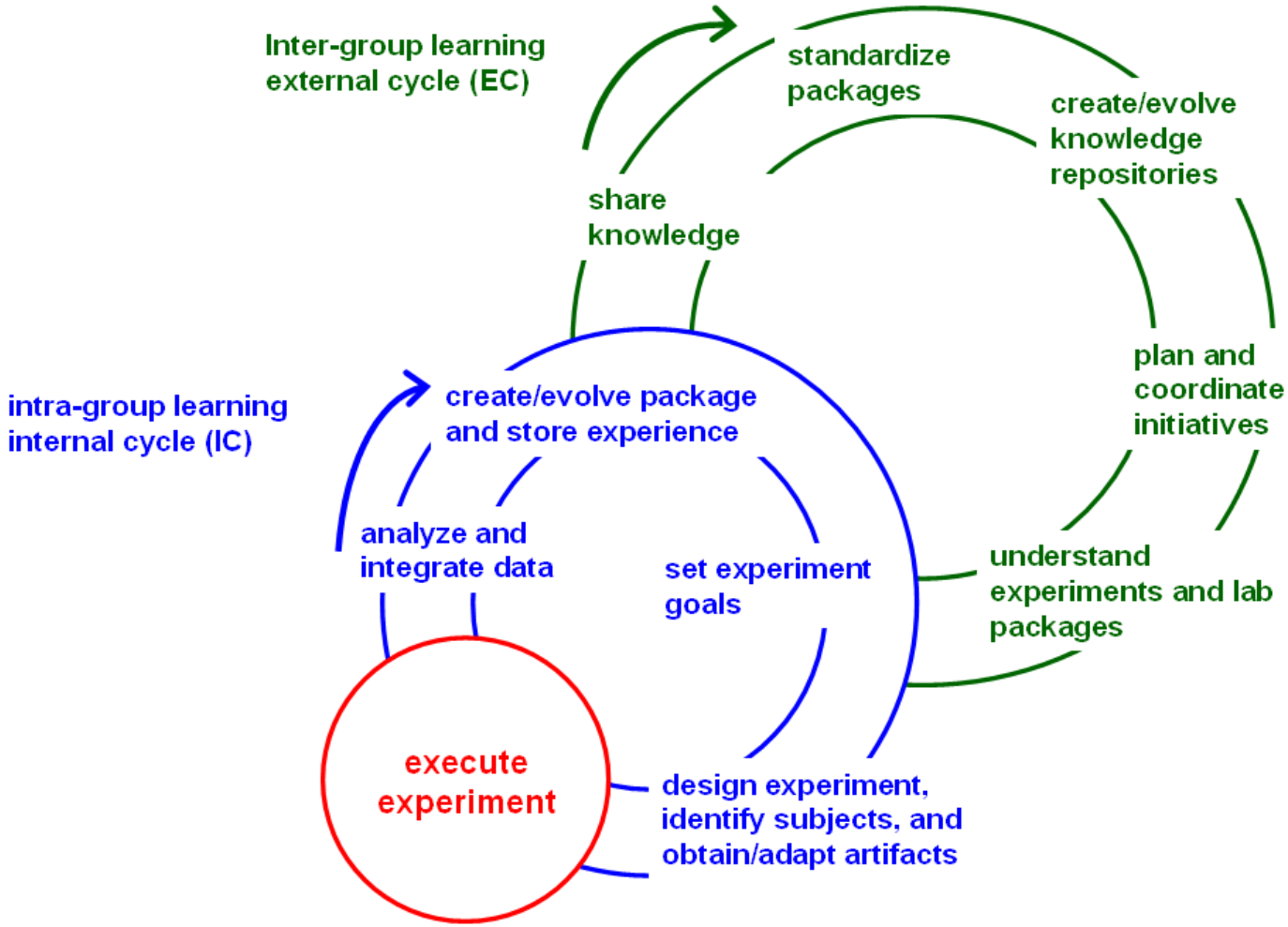
- Software engineering experimentation is hard and experimenters work better when supported by peers;
- Cooperation is key to tackle the complexity of the problems;
- Many replications are needed to truly understand the variables involved in any software engineering experiment;
- Know-how transfer and support is highly desirable when someone is starting with experimentation. In our case, Prof. Victor R. Basili was key to provide us with an "experimentation culture";
- Full cooperation requires trust among the partners;
- Close cooperation and knowledge sharing has a key role to play in software engineering experimentation.

Knowledge Sharing in Experimental Software Engineering



Experimentation Knowledge Sharing Model (Shull et al., 2004)

Framework for Improving the Replication of Experiments



FIRE - Framework for Improving the Replication of Experiments (Mendonça et al., 2007)

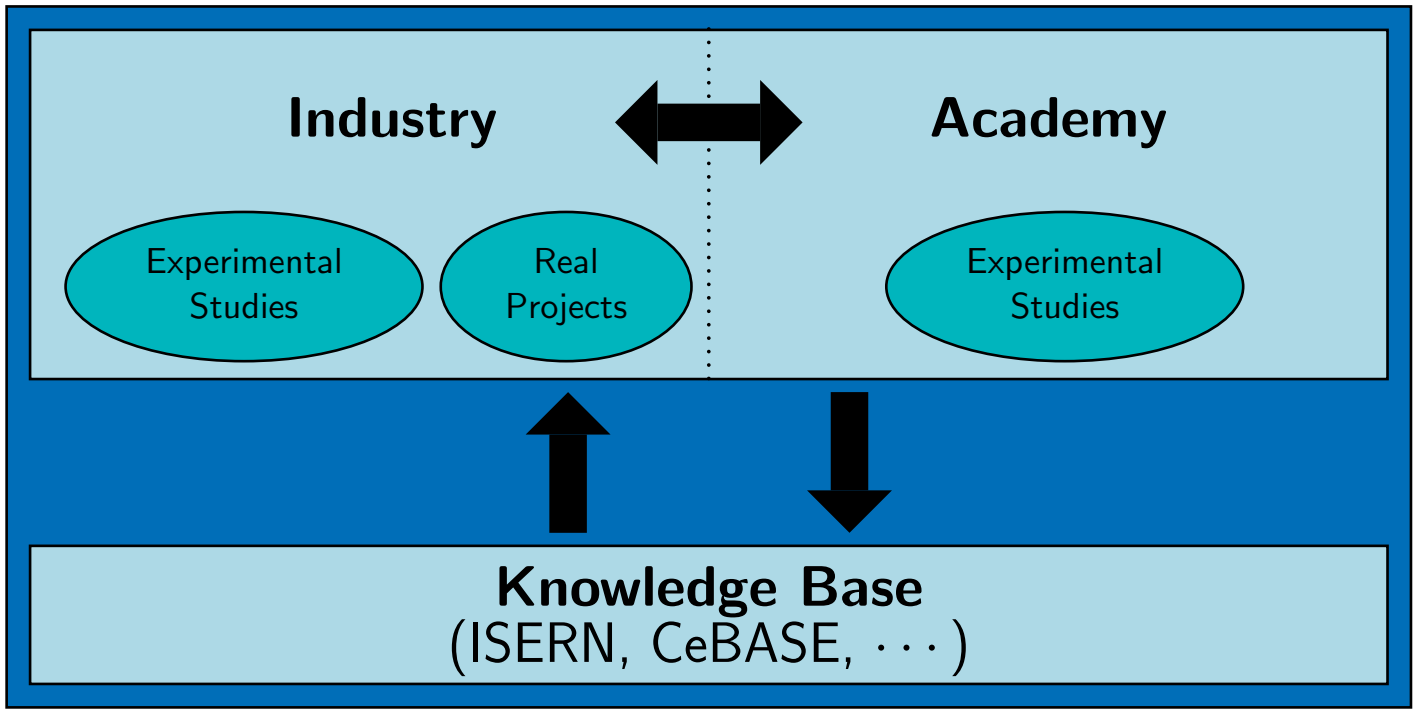
Human Resource

- Academy have not correspond to the industry expectations. Some initiatives are:
  - George Mason University.
  - Kansas State University.
  - Purdue University.
  - ICMC-USP.
  - UFPE/Motorola.
  - ...
- In Brazil:
  - More than 1,000 major courses:
    - few of them provide regular subjects on software testing.
    - initiative of ICMC-USP is to integrate software testing with basic programming concepts.

Contributions

- Experimental Studies.
  - Potential-Uses criteria.
  - Pair-Wise Data Flow.
  - Interface Mutation criterion.
  - Mutation testing $\times$ Reactive Systems.
- Testing Tools
  - *Proteum*, *PROTEUM/IM*, *Proteum/RS*.
  - *PokeTool*.
  - *MGASET*.
  - JaBUTi, JaBUTi/MA, JaBUTi/AJ, $\cdots$.
  - *CATSDL*
- Experimental Studies

Increase the Cooperation with Industry



Summary (1)

- Functional, structural and fault-based testing criteria are useful to:
  - systematize the testing activity;
  - generate test cases;
  - evaluate the quality of test set; and
  - help the tester do decide when to stop testing.
- They are complementary and should be used together to maximize their benefits and the fault-detection capability.

Summary (2)

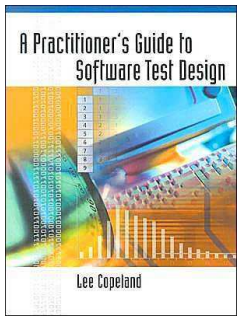
- Experimental Software Engineering plays an important role in this scenario by:
  - providing a knowledge base about different aspects of V&V techniques, including the testing criteria.
  - establishing a relationship between different V&V techniques.
  - allowing the definition of incremental testing strategies, combining different testing criteria.

Summary (3)

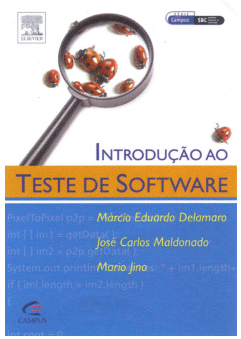
- The support automation for a testing criterion is fundamental for:
  - easing the adoption of the criterion;
  - reducing the human intervention during the testing activity;
  - carrying out experimental studies.
  - teaching and training purpose.
  - technology transference from the academy to the industry.

Recommended Reading

Interested readers can take a look at the first chapters of:



A Practitioner's Guide to Software Test Design
Lee Copeland
Artech House Publishers
Norwood, MA, 2004.



Introdução ao Teste de Software
Márcio Eduardo Delamaro
José Carlos Maldonado
Mario Jino
Campus Elsevier
Rio de Janeiro, RJ, 2007.

References I

H. Agrawal, R. A. DeMillo, R. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. H. Spafford. Design of mutant operators for the C programming language. Technical Report SERC-TR41-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, March 1989.

B. Beizer. *Black-Box Testing : Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, New York, 1995.

T. A. Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, New Haven, CT, 1980.

T. A. Budd. *Mutation Analysis: Ideas, Example, Problems and Prospects*, chapter Computer Program Testing, pages 129–148. North-Holand Publishing Company, 1981.

L. Copeland. *A Practitioner's Guide to Software Test Design*. Artech House Publishers, 2004.

M. E. Delamaro, J. C. Maldonado, and A. M. R. Vincenzi. Proteum/IM 2.0: An integrated mutation testing environment. In *Mutation 2000 Symposium*, pages 91–101, San Jose, CA, October 2000. Kluwer Academic Publishers.

M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, 27 (3):228–247, March 2001.

R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–43, April 1978.

J. B. Goodenough and S. L. Gerhart. Towards a theory of test data selection. *IEEE Transactions on Software Engineering*, 2(3):156–173, September 1975.

J. C. Maldonado. *Potential-Uses Criteria: A Contribution to the Structural Testing of Software*. PhD thesis, DCA/FEE/UNICAMP, Campinas, SP, Brazil, July 1991. (in Portuguese).

References II

J. C. Maldonado, E. F. Barbosa, A. M. R. Vincenzi, M. E. Delamaro, S. R. S. Souza, and M. Jino. Introdução ao teste de software. Technical Report 65 – Version 2004-01, Instituto de Ciências Matemáticas e de Computação – ICMC-USP, April 2004. Available at: http://www.icmc.usp.br/~biblio/index.php?destino=notas_didaticas. Accessed on: 02/19/2007 (in Portuguese).

E. Martins. Teste baseado na implementação (fluxo de dados). Slides de Aula, April 2003. Disponível em: http://www.ic.unicamp.br/~eliane/Cursos/Transparencias/VVTestes/testesc Acesso em: 18/01/2005.

M. G. Mendonça, J. C. Maldonado, M. C. F. de Oliveira, J. Carver, S. C. P. F. Fabbri, F. Shull, G. H. Travassos, E. N. Höhn, and V. R. Basili. A framework to coordinate and evolve software engineering controlled experiments. 2007. (in preparation).

G. J. Myers. *The Art of Software Testing*. Wiley, New York, 1979.

G. J. Myers, C. Sandler, T. Badgett, and T. M. Thomas. *The Art of Software Testing*. Wiley, New York, 2004.

S. Rapps and E. J. Weyuker. Data flow analysis techniques for program test data selection. In *6th International Conference on Software Engineering*, pages 272–278, Tokio, Japan, September 1982.

S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, April 1985.

F. Shull, M. G. Mendonça, V. Basili, J. Carver, J. C. Maldonado, S. Fabbri, G. H. Travassos, and M. C. Ferreira. Knowledge-sharing issues in experimental software engineering. *Empirical Software Engineering*, 9(1-2):111–137, 2004. ISSN 1382-3256. doi: 10.1023/B:EMSE.0000013516.80487.33.