Software Verification and Validation[1]

David Kung, University of Texas at Arlington, USA
Hong Zhu, Oxford Brookes University, UK

Software verification and validation are software quality assurance activities that aim to ensure that the software system is developed according to a development process and meets the customer's needs [1]. In other words, verification is about "are we building the product right", and validation is about "are we building the right product" [2]. Validation is further divided into static validation and dynamic validation. Static validation checks the correctness of the software product without executing the software system or a prototype while dynamic validation executes the software system or a prototype. Software testing is one form of dynamic validation.

To explain, verification is concerned about the process to produce the product. That is, are we building the product in the right way? This includes two aspects: 1) the right process and 2) correctly follows the right process. As a minimum condition, the "right process" must require that a lower level artifact satisfy the requirements stated in the higher level artifact. Unlike verification, which is concerned about the "correctness" of the process, validation is concerned about the correctness of the product. That is, are we building the correct product? We need both verification and validation because either of them alone is not sufficient. For example, an implementation may satisfy the specification but the specification itself may be incorrect. Moreover, the implementation may satisfy the specification and the specification is also correct but the code may be hard to understand, test and maintain. Customer review and/or expert review of requirements specifications would detect the former while code review and code inspection would detect the latter. Therefore, a "right" (which means "preferred") software development process should include these verification and validation activities.

Software verification and validation are important because software has been used in all sectors of our society. Today's software systems are extremely large, complex and process billions of transactions a day in the financial, retailing, manufacturing, transportation, telecommunications, and many other sectors. Many software systems are embedded systems, real time systems, or mission-critical systems. Failures of today's software systems are financially very costly, and politically not acceptable because the failures may incur recall of products, property damages, injury to human body, or even lost of human life. Therefore, software verification, validation and testing have received increasing attention from academic research and industry.

In the following sections, we first provide definitions of commonly encountered verification and validation concepts, followed by verification and validation in the software lifecycle, formal verification and software testing techniques.

---

[1] Accepted for publication in Encyclopedia of Computer Science and Engineering, Benjamin Wah (Ed.), John Wiley & Sons, Inc.

1. Definitions

This section presents the definitions of commonly used terminologies in software verification and validation.

Bug a defect in the program code

Desk Checking examination of software artefact, typically the source code, by the developer to detect bugs, anomalies, and other potential problems.

Error an unanticipated condition that puts the system into an incorrect state

Failure a result produced by the software under test does not satisfy the expected outcome.

Fault a defect in the software system.

Inspection a step-by-step checking of the software artefact/product against a predefined list of criteria, called *check list*.

Peer Review evaluating the software artefacts by peers who are required to answer a list of questions to assess the artefact and provide improvement suggestions, if any.

Regression Test rerun some of the test cases to ensure that the modified software system still delivers the functionality as required

Software Attribute property or characteristic of software

Software Metrics measurements of software (attributes)

Software Quality Assurance activities to ensure that the software under development or modification will meet desired quality requirements

Test Driver code to invoke the component under test and check the outcome of the component under test

Test Harness test driver and test stub

Test Script code to test functionality of software system

Test Stub code to replace the module or procedure that is invoked by the component under test so that the component under test can be executed

Testing executing a program with the intent to uncover bugs

Verification and Validation according to Barry Boehm, verification is “are we building the product right?” and validation is “are we building the right product?”

Walk-through manually reviews software artefact by following the described logic step by step with certain scenario of operating the system and/or on certain input data as the test case. The artifact reviewed could be requirements specifications, high level design, detailed design and source code, etc. The walk-through of source code is often performed as manually executing the software with test data to simulate machine execution of the software.

2. Verification and Validation in the Lifecycle

This section presents verification and validation in the software lifecycle. That is, what is checked in each of the lifecycle phases and who performs the checking. Moreover, what are the techniques used to perform the checking.

2.1. Verification and Validation for the Requirements Phase

Verification and validation in the requirements phase aims at detecting errors in the requirements specification and the analysis models. The techniques used include requirements reviews, inspection, walkthrough, and prototyping. Requirements reviews include customer/user reviews, technical reviews, and expert reviews.

Customer/user reviews are performed by involving the customer and/or users of the system. Customer/user reviews should examine the requirements specification and look for problems in the following areas:

1) The correspondence between requirements and the real world. That is, the requirements specification correctly describes the functional requirements of the application for which the system is to be built or extended.
2) The user interfaces. This includes the appearance, the look and feel, sequence of interaction, input/output data and formats, and the GUI implementation technologies.
3) Non-functional requirements. That is, whether the non-functional requirements, including performance requirements, security requirements and user friendliness requirements, are correctly stated.
4) Constraints. That is, whether application specific constraints are correctly stated. Application specific constraints may include constraints on the operating environment, political constraints, technological constraints, etc.

Technical review is an internal review performed by the technical team. Technical review techniques include peer review, inspection, and walkthrough:

- In peer review, the requirements specification and the analysis models are reviewed by peers, who are guided by a list of review questions designed to qualitatively assess the quality of the product being reviewed. Answers to the

questions by the peers may vary drastically because the answers represent the reviewers' opinion about the product under review and depend heavily on the reviewers' knowledge, experience, background, and criticality. This is similar to the product review reports published in consumer magazines. The review reports by different writers may differ drastically. The review meeting usually runs about 2 hours and takes place one to two weeks after the review assignments, allows the developer and the peers to discuss the feedback and identify action items to address the issues.

- Inspection checks the requirements and the analysis models against a list of items that are found to be error prone or problematic [3,4]. Unlike reviews, inspection looks for more specific problems and the answers can be more objectively. This is similar to car inspection in which the inspector checks the engine, brake, lights, etc. to see if each of them is working properly. Since it is more well-defined, a computer can perform car inspection nowadays.
- Walkthrough is carried out by explaining and examining the requirements [5]. In particular, the analyst who wrote the requirements specification explains each requirement to the peers who would raise questions and stimulate doubts. The analyst would answer the questions and address the doubts. In addition, each of the analysis models is examined carefully. That is, the analyst who drafted the model leads the peers go through the model and provides necessary clarification while the peers may ask questions and raise doubts. This is similar to the new car salesperson at a car dealer who demonstrates a new car to potential buyers by showing various kinds of operations of the car. The buyers usually would raise questions and concerns during the demonstration and the salesperson would address the questions and concerns.

The above verification and validation activities should aim to reveal following problems:

- Incompleteness, which includes
    1. Definition incompleteness, for example, some application specific concepts are not defined.
    2. Internal incompleteness, for example, some requirement expression has an "if part" but does not have the "else part". Another example is that a decision tree or decision table has not considered all possible combinations of the conditions used to construct the decision tree or decision table.
    3. External incompleteness, that is, there are cases that exist in the real world application but not included in the requirements specification. For example, a decision table or decision tree does not include a condition that should have been included.
- Inconsistency, which includes
    1. Type inconsistency, that is, inconsistent specification of one or more data types in the requirements or analysis model.
    2. Logical inconsistency, that is, contradictory conclusions can be inferred from the specification.

3. View inconsistency, that is, inconsistency exists between views of the system by different user groups. For example, the perception of user group A is contradictory to the perception of user group B.

- Ambiguity, which includes
  1. Ambiguity in the definition of application specific concepts, and
  2. Ambiguity in the formulation of requirements
- Redundancy, which includes
  1. Duplicate definitions of the same concept
  2. Duplicate formulations of the same requirement or constraint, and
  3. Unnecessary concepts or constraints
- Intractability, which means the high level requirements do not correspond to the lower level requirements. If the system is being developed using the object-oriented paradigm, then the technical review must ensure that the use cases are tractable to the requirements and vice versa. This is often facilitated by constructing a requirements-use cases tractability matrix during the analysis phase. The matrix shows which requirement is to be realized by which use cases. The review should ensure at a minimum that each requirement is realized by some use cases and each use case serves to realize some requirements.
- Infeasibility in terms of performance, security, and cost constraint. That is, can the development team deliver the functional capabilities as stated in the requirements specification with the expected performance and security within the cost and schedule constraints?
- Unwanted implementation details. Implementation details must not be mentioned in the requirement specification because this limits the design space. Examples include mentions of pointers, physical data structures, and use of pseudo-code or programming language statements.

Expert review in the requirements phase means review of the requirements specification by domain experts, looking for

1) Incorrect or inaccurate formulation of domain specific laws, rules, behaviors, policies, standards, and regulations
2) Incorrect, inaccurate, inappropriate, or inconsistent use of jargons
3) Incorrect perception of the application domain, and
4) Other potential domain specific problems or concerns

In the requirements phase, prototyping or rapid prototyping can take many different forms. The main purpose is to quickly construct a prototype of the system and use it to acquire customer/users' feedback. That is, prototyping is used as a validation technique in the requirements phase to help ensure that the team understands well the customer/users' requirements are correctly captured.

The simplest prototype could be a set of drawings that illustrate the user interfaces of the future system. The most sophisticated prototype could be a partially implemented system that the users can experiment with to gain hands-on experience. The most commonly seen prototype is one that the team can demonstrate the functionality and user interfaces of the

system to the customer or end users. Which type of prototype to use is an application dependent issue. For instance, applications that are mostly concerned with mission critical operations would benefit from prototypes that demonstrate the functionality and behavior. Applications that are end user oriented would benefit from prototypes that demonstrate the user interfaces.

The requirement phase is ideal for preparing system test cases to be used to validate the system before deployment. If use cases or scenarios have been used in requirements analysis, then they can be used to prepare system test cases. First, for each use case or scenario, the user input parameters are identified. Next, the possible input values of each of the input parameters are determined. This can be done as follows. For each input parameter, there are at least three possible cases to consider: 1) using a valid value, 2) using an invalid value and 3) the input parameter is not applicable or not available. A more refined approach will consider other partitions of the input parameter according to the application at hand. In addition to equivalence partitioning, boundary values for each input parameter can also be used. A table with the columns representing the input parameters and the rows representing the test cases can then be constructed and used during the system testing phase.

2.2. Verification and Validation for the Design Phase

Verification and validation in the design phase are aimed at assessing the correctness, consistency, and adequacy of the design with respect to the requirements and analysis models. Verification and validation activities in the design phase use review, inspection, walkthrough, formal verification, and prototyping techniques. Depending on who perform these activities, we have peer review, customer review, and expert review. Peer review, inspection, walkthrough and formal verification are performed by the development team. These are mostly verification activities although some of them may concern with design validation.

Peer review, inspection, walkthrough and formal verification check the design documentation to ensure that

1) Correct use of the design language. This includes:
    - The notions and notations of the design specification language are used correctly.
    - The design specification expresses clearly and correctly the design of the proposed system.
2) Adequacy. The design specification prescribes a solution that is implemented will satisfy the requirements of the proposed system. This can be done as follows:
    - The high level verification ensures that each requirement is realized by some modules in the design and each module in the design is necessary for satisfying some requirements.
    - The detail level verification aims at ensuring that the capability stated in each requirement can actually be delivered when the system is implemented according to the design specification. This can be

accomplished by a design traversal to demonstrate how the requirement can be satisfied.

3) Non-redundancy. This includes:
    - The design does not include items that are not necessary for satisfying the requirements or significantly improving design quality. (For instance, design patterns may introduce additional classes but proper use of design patterns significantly improves design quality.)
    - The design does not contain items that are already covered by other part of the design. For instance, a rule in a decision table may already be covered by other rule(s).
4) Consistency. This includes:
    - Logical consistency. That is, the various portions of the design specification do not contain contradictory design descriptions. For example, decision table or decision tree are commonly used in the design phase to describe process logic for modules. A decision table or decision tree is inconsistent if two or more rules have the same condition combination but different action sequences. When the design is represented in a modeling language such as UML, it may contain a number of diagrams to represent the system from different views and/or at different levels of abstraction. These diagrams must also be checked for consistency across the diagram.
    - Definition-use consistency. That is, the use of a component, class, data structure, data element, or function corresponds to the respective definition and interaction sequence. For example, the invocation of a function must correspond to the definition of the function signature and return type. A commonly seen inconsistency in object interaction design or sequence diagramming is an object calling another object but the called function is not defined.
    - Design/specification consistency. That is, the design specification is consistent with the models constructed in the analysis phase.
5) Internal completeness. Checking internal completeness is to ensure that the design has covered all possible combinations of a given set of conditions. For example, if a decision table has three binary conditions then it must contain eight independent rules to cover the eight possible combinations of the three binary conditions.
6) Design principle compliance. The design follows well-known design principles such as separation of concerns, high cohesion, and low coupling. This can be facilitated by computing and analyzing a set of design quality metrics such as cohesion, coupling, scope of effect, scope of control, fan-in, fan-out, class size, height of inheritance tree, and design complexity metrics. For example, the class size metric is the number of methods in a class. If the class size less the number of getters and setters is large, then the class may have been assigned too many responsibilities. This in term may signify that the cohesion of the class will be low. The reviewers can then focus their effort in examining such classes.
7) Module interface. That is, communication between modules is explicit and easy to understand. Moreover, there should be no hidden assumptions for invoking a module.

While peer review, inspection and walkthrough are mostly concerned with design verification, customer review and prototyping are mainly concerned with design validation. They are usually performed jointly by the development team and the customer (or customer representative including the system analyst). As a design validation activity, customer review and prototyping aim at detecting mismatches, omissions, or inconsistencies between the design and the customer's interpretation of the requirements, including

1) Mismatch between designed functionality and/or behavior and the functionality and/or behavior as expected by the customer/users.
2) Mismatch between system states, events and cases and the actual states, events and cases in the business domain. Note this includes checking of external completeness.
3) Mismatch between the system's user interface design and what is expected by the customer/users.
4) Mismatch between the system's interfaces to other systems and the required interfaces in the real world.

Another validation activity in the design phase is the preparation of functional test cases, behavioral test cases, and integration test cases. The design phase is idea for the preparation of these test cases because all needed information is contained in the design documents. For example, if decision tables have been used in the design phase to express process logic, then each rule of the decision table is a cause-effect test case. If state machines have been used in the design phase to describe state dependent behaviors, then the state machines can be used to derive transition sequences to test the implemented state dependent behaviors.

Integration test cases can be derived from structured charts (also called routine diagrams) using a pre-order traversal in top-down integration and post-order traversal in bottom-up integration. If the system is being developed using an object-oriented approach, then the integration test cases can be derived from sequence diagrams or collaboration diagrams. That is, deriving test cases that will exercise message passing paths according to the coverage criteria selected.

2.3. Verification and Validation for the Implementation Phase

Verification and validation for the implementation phase are aimed to ensure that the source code complies with the organization's coding standards, implements the required functionality, satisfies performance, real time and security requirements, and properly handles exceptional situations. Desk checking, code review, inspection, and walkthrough are commonly referred to as verification and static validation methods while testing is commonly referred to as the dynamic validation method. All these are commonly used in the implementation phase.

In desk checking, the programmer checks the program written by him/her. The programmer may use a pencil, a calculator and/or other devices. It is an informal process

and hence the effectiveness and efficiency depends on the individual programmer. In code review, the program is reviewed by peers who are required to comment on the quality of the code and answer a set of questions. Code inspection checks the code against a list of problems or defects that are commonly found in programs. The most famous code inspection method was proposed by Fagan and is called the Fagan inspection method [3, 4]. In walkthrough, the reviewers use test data or a specific scenario in the operation of the software and manually follow step by step the logic described in the artifact under review to understand how the system operates and then to detect errors [5]. For example, when the artifact under review is a piece of source code, the reviewers manually execute the program by following the control flow between the statements and expressions in the code. Finally, testing is actually executing the program with test cases derived using ad hoc or systematic test case generation methods. Testing is distinct and indispensable because testing can detect performance bottlenecks and incorrect interface. These usually cannot be detected by the static validation methods.

Desk checking, code review, code inspection, and walkthrough are aimed to detect problems such as the following:

1) Incorrect/inadequate implementation of functionality
2) Mismatch of implementation and design
3) Mismatch of module interfaces
4) Coding standards are not followed
5) Poor code quality as measured by various code quality metrics such as cyclomatic complexity (e.g., some companies require this to be no more than 10), information hiding, cohesion and coupling, and modularity
6) Improper use of the programming language
7) Incorrect/improper implementation of data structures or algorithms
8) Errors/anomalies in the definition and use of variables such as variables or objects are defined but not used, not initialized or not correctly initialized
9) Infinite loop
10) Incorrect use of logical, arithmetic, or relational operators
11) Incorrect invocation of functions
12) Inconsistencies caused by concurrent updates to shared variables
13) Performance bottlenecks and/or inability to fulfill timing requirements/constraints
14) Incorrect interface to devices and/or handling of device interrupts

Desk checking, code review, code inspection and walkthrough are effective in detecting errors and anomalies if applied properly. In particular, ordinary testing methods may not detect problems as described by 4), 5), 7), 13)—14) in the above.

2.4. Verification and Validation for Integration Phase

In the integration phase the software modules are integrated to form a complete software system. Dynamic validation or testing is the main activity of this phase. The purpose of integration testing is to detect errors in the interfaces between the software modules. These include

1) Incorrect assignment of actual parameters to formal parameters
2) Incorrect assignment of values to variables in one module and/or incorrect use of the variables in another module
3) Incorrect interaction between modules. For example, incorrect sequence of function calls or module invocations
4) Incorrect state behavior resulting from module interactions

Integration testing can be carried out by using one or more of the following strategies. These strategies assume that the architectural design has a tree or lattice structure with a top-level module that invokes second level modules which invoke third level modules, etc.:

1) Top-down strategy. Integration testing begins with testing the interfaces between the top-level module that corresponds to the overall system and modules that are invoked by the top-level module. Lower level modules that are invoked by modules being integrated are replaced by test stubs. A test stub is a module that is specifically constructed to provide the output values as expected by the higher level module. We need to use test stubs because we have not tested the interfaces between the modules being integrated and the modules being replaced; if any of these interfaces is incorrect, then the error may propagate up and affect the integration testing result at the higher level.
2) Bottom-up strategy. Integration testing begins with testing the interfaces between the lowest level modules and their parent module and progresses up the hierarchy. A test driver is needed to invoke the parent module because the interface between the parent module and its parent module has not been tested.
3) Hybrid strategy. As the name suggests, integration testing may proceed using both of the above strategies in various combinations.
4) Criticality based strategy. Integration testing begins with integrating critical modules of the system first as long as the modules are available. This strategy allows the critical modules to be exercised more often and hopes to detect more errors in these modules.
5) Availability based strategy. Integration testing is carried out incrementally by adding modules that are ready to be integrated into the software system.
6) Monolithic strategy. Integration testing is performed by integrating all the modules of the system at once.

2.5. Verification and Validation for System Testing

During the system testing phase, the software system is integrated with other systems and tested against the software/system requirements. System testing is usually performed in the development environment. The end product of system testing is a system that is ready for deployment and acceptance test in the customer’s target environment.

As indicated in the above, system testing is performed against the software/system requirements including functional and non-functional requirements. The objective is to

ensure that the system satisfies the functional and non-functional requirements. In addition, the system must also satisfy the constraints stated in the requirements specification. System testing with respect to functional requirements can be carried out using one or more of the following approaches:

- Use case based testing. As described in the section titled "2.1. Verification and Validation for the Requirements Phase", if system use cases have been derived from the requirements, then system testing can be performed by testing that the system satisfies each of the use cases. Please see the section for more detail.
- Random testing. Test data are selected randomly to test the system against the requirements. This may or may not use an input data distribution profile, which can be obtained from existing or similar systems' usage log.

In addition to functional testing, performance and stress testing are also performed during the system testing phase. Performance testing includes testing the throughput and response time according to the predefined workload and stress testing is concerned with system throughput and response time under a workload that is multiple times or even ten folds of the normal workload.

2.6. Verification and Validation for Acceptance Testing

During the acceptance testing phase, the analyst or a consultant hired by the customer will conduct or direct the testing of the system in the customer's target environment to ensure that the system operate properly in that environment. Since the difference between system testing and acceptance testing is the environment, acceptance testing can be carried out by executing a subset of the test cases used during system testing. Clearly, test cases selection should be guided by changes to environment parameters such as system configuration, run conditions, network configurations, etc.

2.7. Verification and Validation for Maintenance

Once the system is installed and operational in the target environment, the maintenance phase begins. Therefore, the operation and maintenance phases are in fact one combined, indivisible phase. Due to system dynamics [6], continual changes are made to the system once it is released to field operation. Changes or enhancements performed on the system are collectively called maintenance activities. These include:

- Corrective maintenance to correct errors in the system
- Enhancements to add additional capabilities to the system
- Improvements to system including performance, response time, user friendliness, and other quality aspects
- Migration to new hardware, new technologies, or new operating environment
- Preventive maintenance to prepare the system for possible problems such as virus attack

The verification and validation techniques such as review, inspection, walkthrough, and testing can still be used in the maintenance phase to verify and validate the changes. However, there are several issues that must be considered during the maintenance phase:

- Change impact analysis. Changes can affect other parts of the system and the impact must be identified and analyzed before the changes are made. This is usually described in the Engineering Change Proposal along with change cost and schedule and evaluated by a Change Control Board. This topic is beyond the scope of this article and covered by Software Configuration Management.
- Review, inspection, walkthrough may be conducted for new, changed, and affected modules.
- New test cases must be designed to test the newly introduced modules.
- The changed and affected modules must be retested using existing test cases to ensure that no undesired side-effect has been introduced. This is commonly called regression testing.

3. Formal Verification

Formal verification is a means to verify a specification or a design mathematically. There are two main approaches to formal verification.

The first approach is based on theorem-proving methods [7, 8, 9]. We call this approach the proof-theoretical approach. In this approach, a system specification consists of a set of declarative statements or declarative sentences. These statements typically specify properties of real world and/or system entities or objects, their behaviors and their relationships. In mathematical terms, the set of statements is called a theory and assumed to be true at all time because the statements state what are about the system. In computer science and software engineering, the statements are called nonlogical axioms because they are not logically true but assumed to be true according to laws of the real world application. For example, “every customer has an account” and “every account is owned by a customer” cannot be proved to be true logically but they could be true for some bank application. Formal verification in the proof-theoretical approach is to prove that desired system properties or constraints are logical consequences of the nonlogical axioms. That is, desired properties or constraints can be logically derived from the nonlogical axioms.

Consider for example, an overly simplified formal specification of a stack:

1) Maximal size of stack.
MAX=2

2) Initial size of stack.
size(S0)=0
where S0 denotes the initial state.

3) Operation “push” (we focus only on the size but nothing else).
size(S)=s & s < MAX $\rightarrow$ size(push(S))=s+1

(If stack size in state S is s and s is less than MAX then stack size in the state resulting from pushing a element onto the stack is s+1.)

4) Operation “pop”.
size(S)=s & s > 0 → size(pop(S))=s−1

Now suppose we want to prove the desired property stating that “there is some state in which the stack size will be MAX, formally

5) (∃S)size(S)=MAX

That is, there exists a state S in which the size of the stack is MAX.

We will illustrate the proof using the resolution proof technique proposed by Robinson [10]. To prove that Q is a logical consequence of P1, P2, ..., Pn, we prove that ~Q, P1, P2, ..., Pn cannot be true at the same time, where Q, P1, P2, ..., Pn are statements. Resolution proof begins with the set of statements {Q, P1, P2, ..., Pn} and each resolution tries to deduct a statement called resolvent from two statements using the logical inference rule “A & (A → B) ⇒ B” or equivalently “A & (~A ∨ B) ⇒ B”. That is, from statement “A” and statement “~A ∨ B” we can deduct “B”. Clearly, each resolution step takes two statements and produces one new statement. If the set of statements can be deduced to produce the nil statement, denoted by “□” and representing a contradiction, then the theorem is proved. The proof of our stack example is shown in Figure 1.

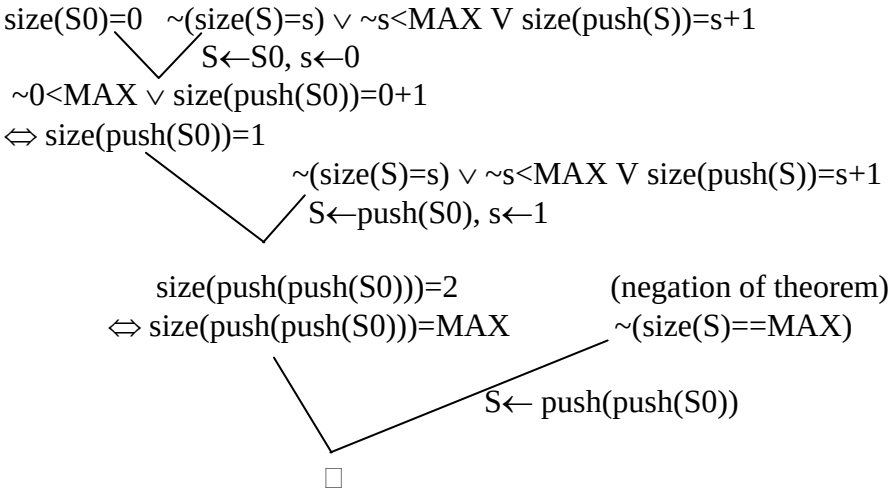

Figure 1. Resolution proof of the simplified stack specification

The above simple example is a special case because it does not use the so-called “frame axiom” originally proposed by McCarthy and Green [11]. In their effort to construct the first question answering system using logical inference, McCarthy and Green discovered that the specification of the effect of an operation like 3) and 4) in the above stack specification example is not enough. The specification must also state that everything that

is not changed by the operation remains true in the new state resulting from the operation. This is commonly referred to as the “frame axiom”. Fortunately, there is nothing not changed by the operations push and pop, therefore, our simple example does not have to use the frame axiom.

Now suppose we want to prove another desired property that states “the size of the stack is always greater than or equal to zero”, formally

6) $(\forall S)$ size(S) >= 0

The reader will soon discover that applying resolution to prove this property is extremely difficult (almost impossible). A proof technique that is commonly used to proving theorems that state properties true for all cases like this is the proof by induction technique. Using induction proof, the property is proved for the basis case, and then it is assumed to be true for all cases up to a number k, finally, the property is proved for the k+1 case. We illustrate this in the following. We use op(S) to denote either push(S) or pop(S) and $op^k(S)$ a sequence of k push or pop operations applied in S.

The basis step. Since size(S0)=0 is given in 2) in the specification, this implies size(S0)>=0. Therefore, property is true in S0.

The hypothesis step. Now assume that $size(op^k(S0))>=0$ for all sequences of k push or pop operations applied in the initial state.

The induction step. We need to prove $size(op^{k+1}(S0))>=0$. Since there are only two operations, therefore, $size(op^{k+1}(S0))$ can only be $size(push(op^k(S0)))$ or $size(pop(op^k(S0)))$. Since $size(push(op^k(S0))) = size(op^k(S0))+1$ according to 3) and $size(op^k(S0))>=0$ due to hypothesis, $size(push(op^k(S0))) > 0$ and hence $size(push(op^k(S0))) >=0$. Moreover, since $size(op^k(S0))>=0$ and pop can only be applied in state $op^k(S0)$ if $size(op^k(S0))>0$. Thus, $size(push(op^k(S0))) >=0$. Therefore, $size(op^{k+1}(S0))>=0$.

A property about a software system that is true in all states, like the above example, is called invariants.

The second approach is called model checking [12,13, 14,15]. This approach can also be called the model-theoretical approach. In this approach, the system is represented by an operational model, which typically depicts the system behavior. The commonly used operational model for model checking is a state machine consisting of vertices representing system states and directed edges representing system behaviors that cause state transitions. Each system state is specified by a logical or conditional statement. That is, the system is in that state if and only if the condition is evaluated to true using system attributes. Formal verification in the model checking approach begins with the initial system state and generates the states by applying the operations. The desired properties or constraints are checked against each of the states generated and violations are reported.

Consider a simplified thermostat example consisting of only a season switch, an AC relay and a furnace relay as shown in Figure 2. The desired properties for the thermostat could be the following:

C1. Not (SeasonSwitchOff and (FurnaceOn or ACOn))
C2. Not (FurnaceOn and ACOn)
C3. Not (SeasonSwitchCool and FurnaceOn)
C4. Not (SeasonSwitchHeat and ACOn)

Applying the operations of the thermostat results in the tree as shown in Figure 3. A system state is represented by a triple (S1, S2, S3), where S1 denotes the state of the season switch, S2 denotes the state of the furnace relay, and S3 the state of the AC relay. The figure shows that starting in the initial state, the thermostat can enter into a state in which the season switch is at cool, and the furnace and AC are both on. This violates constraint C2 and constraint C3. In practice, model checking can be used to check not only static constraints like C1-C4 in the above but also temporal constraints that involve sequences of states rather than a single state. This is also true for theorem proving approach. Furthermore, the model checker would explore millions of state rather than only a few states as shown in Figure 3.

In practice, the state machine models are converted into the specification language of the model checker. Using SPIN [14] this would be the Promela language, which is a subset of the C programming language. The property to be verified is expressed as a temporal logic expression. The checker will explore the state space and verify the property.

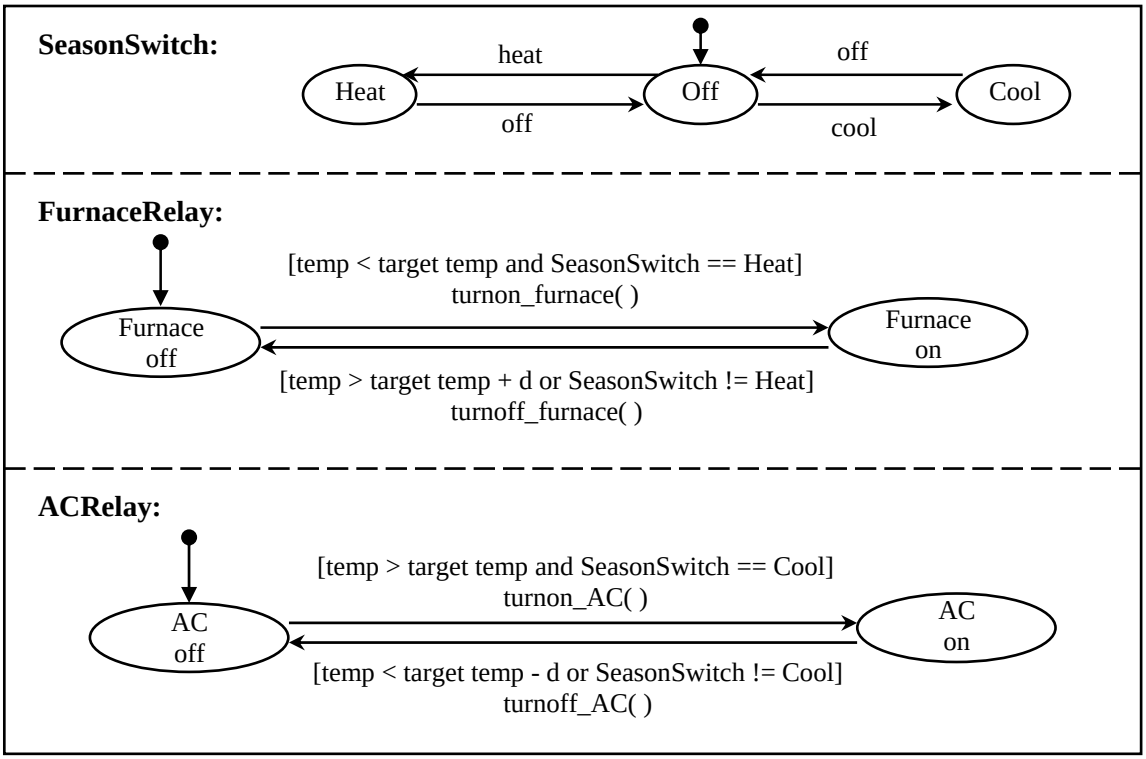

Figure 2. Thermostat specification

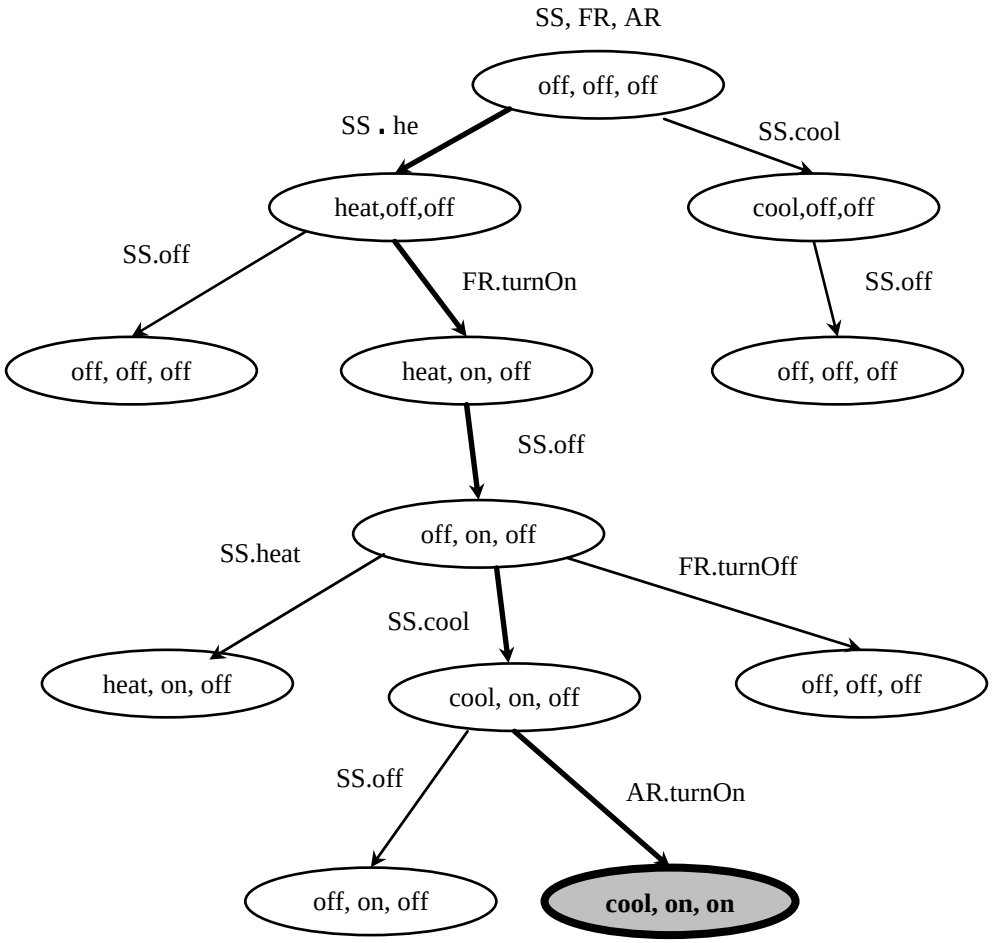

Figure 3. Partial state space of the thermostat example

In recent year, model checking has been applied to checking code or implementation rather than the specification [16,17,18]. This has been termed “software model checking”. In software model checking, the model is constructed from code or implementation rather than from the specification. The construction can be manual or semi-automatic.

4. Software Testing Techniques

This section gives a brief introduction to well-known software testing techniques and methods.

4.1. Software Testing Processes

Generally speaking, software testing is an iterative process that involves a number of technical and managerial activities. In this section, we will focus on the technical aspects.

As shown in Figure 3, the main technical activities in software testing process include planning, generating and selecting test cases, preparing test environment, testing the program under test and observing its dynamic behavior, analyzing the observed behavior

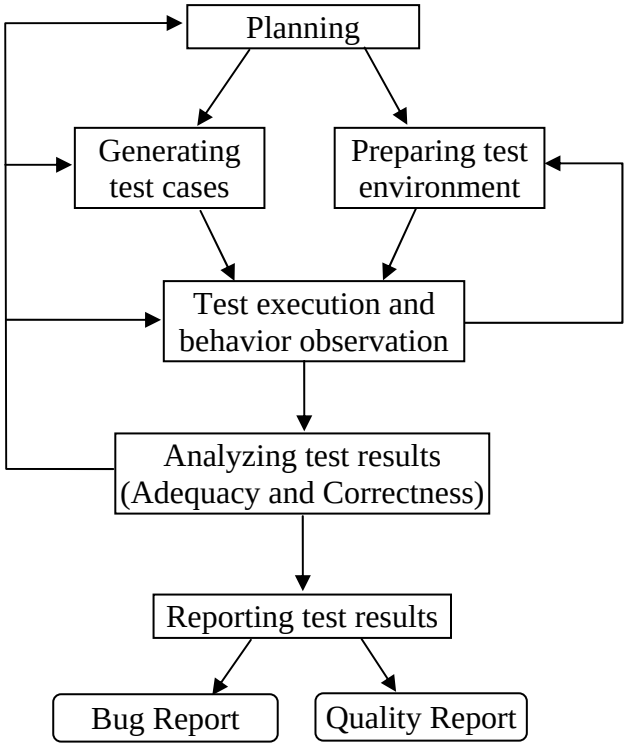

Figure 4. Illustration of activities in software testing process

on each test case, reporting test results, and assessing and measuring test adequacy.

In software testing practice, testers are confronted with questions like: Which test cases should be used? How to determine whether a testing is adequate? Or when can a testing process stop? These questions are known as the *test adequacy* problem [19]. They are the central issues in software testing and the most costly and difficult issues to address. A large number of test criteria have been proposed and investigated in the literature to provide guidelines to answer these questions. Some of them have been used in software testing practice and required by software development standards. A great amount of research has been reported to assess and compare their effectiveness and efficiency.

The observation of dynamic behavior of a program under test is essential for all testing. Such observations are the basis of validating software's correctness. The most often observed software behaviors are the input-output of the program during testing. However, in many cases, observation of the internal states, the sequences of code executed as well as other internal execution histories are necessary to determine the correctness of the software under test. Such internal observations are often achieved by inserting additional code into the program under test, which is known as *software instrumentation*. Automated tools are available for the instrumentation of programs in various programming languages. Behavior observation can also be a very difficult task, for

example, in the testing of concurrent systems due to non-deterministic behavior , in testing component-based systems because of the unavailability of source code, in testing real-time systems due to their sensitiveness to timing and load, in testing systems that are history sensitive such as machine learning algorithms where the reproduction of a behavior is not always possible, in testing of service-oriented systems due to the lack of control of third-party services, and so on.

Checking the correctness of a program's output as well as other aspects of dynamic behavior s is known as the *test oracle problem*. A test oracle is a piece of program that simulates the behavior of the program under test. It could be as simple as a person or a program that judges the output of the program under test according to the given input. If a formal specification of the system is available, then the output can be judged automatically, e.g., by using algebraic specifications [20,21,22]. A recent development in the research on metamorphic software testing method enables testers to specify relationships between outputs of a program on a number of test cases and to check if the relationships held during testing [23].

4.2. Testing methods

Testing activities, especially test case selection and generation and test adequacy assessment, can be based on various types of information available during the testing process. For example, at requirements stage, test cases can be selected and generated according to the requirements specification. At design stage, test cases can be generated and selected according to the architectural design and detailed design of the system. At the implementation stage, test cases are often generated according to the source code of the program. At the maintenance stage, test cases for regression testing should take into consideration the part of the system that has been modified, either the functions added or changed or the parts of the code that are modified. In general, software testing methods can be classified as follows[2].

- *Specification-based testing methods*

In a specification-based testing method, test results can be checked against the specification, and test cases can be generated and selected based on the specification of the system. For example, test cases can be generated from algebraic specifications [24], derived from specifications in Z [25,26], or using model checkers to automatically generate test cases from state machine specifications [27,28].

- *Model-based testing methods*

[2] Traditionally, testing methods were classified into *white-box* and *black-box* testing. White-box testing was defined as testing according to the details of the program code, while black-box testing does not use the internal knowledge of the software. Many modern testing methods have difficulty to be classified either as black-box or white-box. Thus, many researchers now prefer a more sophisticated classification system to better characterize testing methods.

A model-based testing method selects and generates test cases based on diagrammatic models of the system, which could be a requirements model or design model of the system. For example, in traditional structured software development, test cases can be derived from data flow, state transition, and entity-relationship diagrams [29]. For testing object-oriented software systems, techniques and tools have been developed to generate test cases from various UML diagrams [30, 31].

- *Program-based testing methods*

A program-based testing method selects and generates test cases based on the source code of the program under test. Tools and methods have been developed to generate test cases to achieve statement, branch, and basis path coverage. Another program-based testing method is the so-called *decision condition testing* method, such as *modified condition/decision coverage* (MC/DC) criterion [32] and its variants [33], which focus on exercising the conditions in the program that determine the directions of control transfers.

- *Usage-based testing methods*

A usage-based testing method derives test cases according to the knowledge about the usage of the system. For example, a random testing method uses the knowledge about the probability distribution over the input space of the software, such as the operation profile. Another commonly used form of usage-based testing is to select test cases according to the risks associated to the functions of the software.

It is worth noting that it has been recognized for a long time that testing should use all types of information available rather than just rely on one type of information [34]. In fact, many testing methods discussed above can be used together to improve test effectiveness.

4.3. Testing Techniques

A number of software testing techniques have been developed to perform various testing methods. These testing techniques can be classified as follows.

- *Functional testing techniques*

Functional testing techniques aim at thoroughly testing the functions of the software system. It starts with the identification of the functions of the system under test. The identification of functions can be based on the requirements specification, the design and/or the implementation of the system under test. For each identified function, its input and output spaces and the function in terms of the relation between the input and output are also identified. Test cases are generated in the function’s input/output spaces according to the details of the function. The number of test cases selected for each function can also be based on the importance of the function, which often requires a

careful risk analysis of the software application. Usually, functions are classified into high risk, medium risk or low risk according to the following criteria.

- The cost and the consequences that a failure of the function may cause
- The frequency that the function will be used
- The extent to which the whole software systems' functionality and performance depends on the function's correctness and performance
- The likelihood that the implementation of the function contains faults, say because of high complexity, the capability and maturity of the developers, or any priori knowledge of the system

A heuristic rule of functional testing is the so-called 80-20 rule, which states that *80% of test efforts and recourses should be spent on 20% of the functions of the highest risks*.

An advantage of functional testing techniques is that various testing methods can be combined. For example, functions can be identified according to the requirements specification. If additional functions are added during design, they can also be identified and added into the list of functions to be tested. An alternative approach is to identify functions according to the implementation, such as deriving from the source code. When assigning risks to the identified functions, many factors mentioned in the above criteria can be taken into consideration at the same time. Since some of the factors are concerned with users' requirements and some are related to the design and implementation, it naturally combines requirements-based with design and implementation-based methods. The main disadvantage is that functional testing techniques are largely manual operations, although they are applicable to almost all software applications.

- *Structural testing techniques*

Structural testing techniques regard a software system as a structure that consists of a set of elements of various types interrelated to each other through various relationships. They intend to cover the elements and their interrelationships in the structure according to certain criteria. Typical structural testing techniques include control flow testing and data flow testing techniques and various techniques developed based on them.

Control flow testing techniques represent the structure of the program under test as a flow graph that is a directed graph where nodes represent statements and arcs represent control flows between the statements. Each flow graph must have a unique entry node where computation starts and a unique exit node where computation finishes. Every node in the flow graph must on at least one path from the entry node to the exit node. For instance, the following program that computes the greatest common divisor of two natural numbers using Euclid's algorithm can be represented as a flow diagram shown in Figure 5.

```
Procedure Greatest-Common-Divisor;
Var x, y: integer;
Begin
   input (x,y);
   while (x>0 and y>0) do
         if (x>y)
```

```
                then x:= x-y
                else y:= y-x
            endif
        endwhile;
        output (x+y);
    end
```

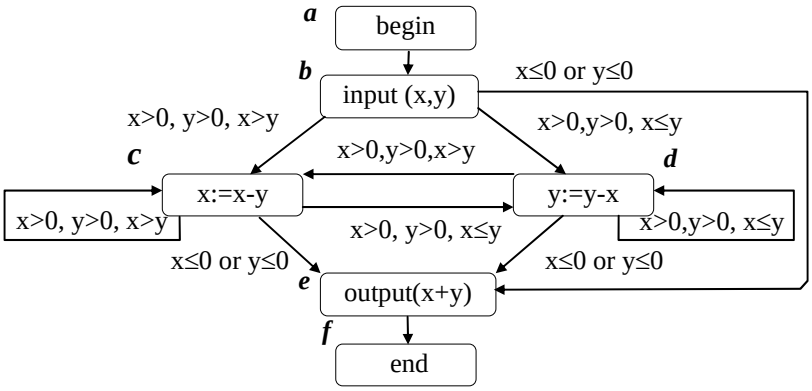

Figure 5. Flow graph of the Greatest Common Divisor program

As a control flow testing method, statement testing requires the test executions of the program on test cases exercise all the statements, i.e. nodes, in the flow graph. For example, paths p=(a, b, c, d, e, f) in Figure 5 covers all nodes in the flow graph, thus the test case t_1= (x=2, y=1) that causes the path p to be executed is adequate for statement testing. Obviously, an adequate statement testing may not execute all the control transfers in the program. Branch testing requires the test cases to exercise all the arcs in the flow graph, i.e. all the control flows, thus the branches, of the program. The test case t_1 is therefore inadequate for branch testing. Various path testing techniques require test executions cover various types of paths in the flow graph, such as all paths of length-N for certain fixed natural number N, all simple paths (i.e. the paths that contain no multiple occurrences of any arcs), all elementary paths (i.e. paths that contain no multiple occurrences of nodes), etc.

Data flow testing techniques focus on how values of variables are assigned and used in a program. Each variable occurrence is therefore classified to be either a definition occurrence or a use occurrence:

- *definition occurrence*: where a value is assigned to the variable.
- *use occurrence* (also called *reference occurrence*): where the value of the variable is referred to. Use occurrences are further classified into *computation uses* (*c-use*) and *predicate uses* (*p-use*).
    - *predicate use*: where the value of a variable is used to decide whether a predicate is true for selecting an execution path;
    - *computation use*: where the value of a variable is used to compute a value for defining other variables, or as an output value.

For example, in the assignment statement y:= x_1– x_2 variables x_1 and x_2 have a computation use occurrence while variable y has a definition occurrence. In the if-statement *if x=0 then goto L endif*, variable x has a predicate use occurrence. Figure 6 shows the flow graph with data flow information of the program given in Figure 5.

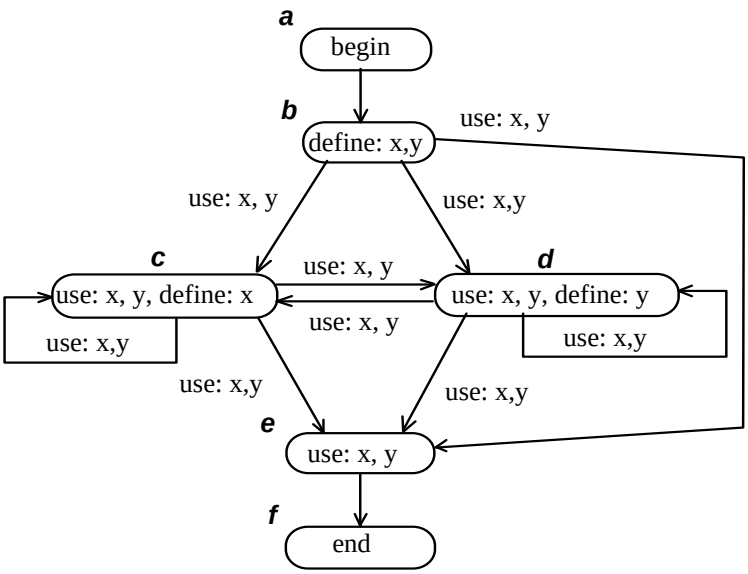

Figure 6. Flow graph with dataflow information

Using such data flow information, the data flow in a program can be expressed by the paths from a node where a variable *x* is defined to a node where the variable is used, but no other definition occurrence of the same variable *x* on the path (which is called *definition-clear paths of x*). Such a path is called a *definition-use association*. The principle underlying all data flow testing is that the best way to test if an assignment to a variable is correct is to check it when its assigned value is used. Therefore, data flow test criteria are defined in the form of exercising definition-use associations or various compositions of the relation. For example, a data flow test criterion in Weyuker-Rapps-Frankl's data flowing testing techniques require testing all definition-use associations [35, 36]. Other data flow testing techniques include Laski and Korel's *definition context coverage criteria* [37], and Ntafos' *interaction chain coverage criteria* [38].

- *Fault-based testing techniques*

Fault-based testing techniques aim at detecting all faults of certain kinds in the software. For example, *mutation testing* aims at detecting all the faults that are equivalent to mutants generated by a set of mutation operators [39, 40]. In general, a mutation operator is a transformation that modifies the software with a single small change and preserves the software's syntax to be well-formed. For example, a typical mutation operator changes a greater than symbol > in an expression to be the less than symbol <. When this mutation operator is applied to the condition of the if-statement in the program given in Figure 5, the following mutant will be generated.

```
Procedure Greatest-Common-Divisor;
Var x, y: integer;
Begin
   input (x,y);
   while (x>0 and y>0) do
         if (x<y) /*Mutation operator applied */
               then x:= x-y
```

```
                else y:= y-x
            endif
        endwhile;
        output (x+y);
    end
```

Figure 7. A mutant of the Greatest Common Divisor program

Each mutation operator represents a kind of errors that could be made by software developers. If a test case enables the original software under test and the mutant to produce different outputs, we say that the mutant is killed by the test case or simply the mutant is dead. This means that the modified part of the program has been executed and that the part actually affects the behavior of the system. Therefore, if the original program contains a fault at the location where the mutation operator is applied, the test case should be able to detect it. Otherwise, solely based on the test executions on the test cases, we would have no evidence to claim that the test cases are capable of differentiate the mutants from the original. In other words, if there is a fault, the test cases would not be able to detect it. Of course, there are two reasons that a mutant remains alive after testing on all test cases. First, the mutant is equivalent to the original. Thus, it cannot be killed. Second, the test cases were unable to kill it due to its inadequacy. The proportion of non-equivalent mutants that remain alive after testing, which is called *mutation score* in software testing literature, gives a clear indication of the adequacy of the test set, and serves as a test adequacy criterion.

Measuring the mutation score of a test set is, therefore, an analysis of the test adequacy. Different levels of mutation analysis can be done by applying mutation operators to the corresponding syntactical structures in the program [41, 42, 43, 44]. Table 1 below summarizes the levels of mutation analysis and the methods to achieve the goals of the analysis.

Table 1. Levels of Mutation Analysis

Level	Goal	Method (Mutation operators)
Interface Analysis	Ensure interfaces between software components are correct and adequately tested in integration testing.	(1) Mutation operators are designed to model integration errors, (2) Tests only the connections between two modules, a pair at a time, and (3) Applies integration mutation operators only to module interfaces such as function calls, parameters or global variables.
Language Specific Feature Analysis	Ensure language specific features were used properly.	For example, for test Java specific features: Delete and insert *This* keyword; Delete and insert *Static* keyword; Delete member variable initialization; etc.
Polymorphism Analysis	Exercise all possible dynamic type bindings to ensure the correctness of polymorphic behavior of object references.	Change the instantiation type of an object reference to a child or parent class; Delete, insert or change type cast operator; Delete overloading method declarations; Change the parameters of overloading method calls.

Inheritance relationship Analysis	Ensure the inheritance relationships including variable hiding, method overriding, uses of super, and definition of constructors) are correctly defined.	Delete or insert overriding methods and hiding variables; Change the calling position of overriding methods, Rename overriding methods; Delete and insert keyword 'Super'; Delete and insert parent constructor calls; etc.
Class Encapsulation Analysis	Ensure class declarations correctly use encapsulation facilities for various accessibility levels.	Change the access modifiers (i.e. private, protected, public, and unspecified) of the attributes and methods in class declarations.
Statement Analysis	Ensure that every branch is taken and every statement is necessary.	Replace statement with CONTINUE; Replace logical and relational with true or false; Check labels on arithmetic IF statements for usage; Replace DO statements with FOR statements.
Predicate Analysis	Exercise predicate boundaries	Alter predicate and DO loop limit sub-expressions by small amounts; Insert absolute value operators into predicate sub-expressions; Alter relational operators.
Domain Analysis	Exercise different data domains	Change constants and sub-expressions by small amounts;
Coincidental Correctness Analysis	Guard against coincidental correctness	Change data references and operators to other syntactically correct alternatives.

Mutation testing tool such as Mothra for Fortran [42] and MuJava for Java [45] have been developed to automatically generate a large number of mutants from a program under test and to execute the program under test and the mutants and to collect the data about dead and alive mutants. Test cases can also be generated to kill a mutant [46]. The equivalence of a mutant to the original is not decidable, but can be automatically determined for a large proportion of mutants.

The idea of program mutation testing can also be extended to specification-based testing in which mutants of specifications are generated [47]. A specification mutant is killed if the correctness of the output of the program under test is judged differently by the original specification.

More recently, the idea of mutation has also been applied to generate test data. A set of mutation operators designed so that when applied to a test case, they generate a set of test data that are of subtle differences from the original test case [48]. This technique can be

applied to test software systems whose test cases are of complicated structure, such as modeling tools and other test case generation techniques would have difficulties.

- *Error-based testing techniques*

Error-based testing techniques aims at checking all error-prone aspects of the system, where errors are mistakes made software developers. For example, test cases are often selected to test if division by zero error were processed properly by the program.

Among the was well established error-based testing techniques is the boundary analysis testing techniques, which select test cases on the boundary and nearby the boundary of an input space in order to make sure that the programmer has correctly computed the boundary, which has been recognized for a long time as error-prone. As illustrated in Figure 8, two types of boundary errors have long been recognized as the most common programming errors. They are shift errors, in which the border of an input domain is shifted parallel to the correct border either towards the outside of the input domain or towards the outside of the domain, and rotation errors in which the border is rotated with respect to the correct border.

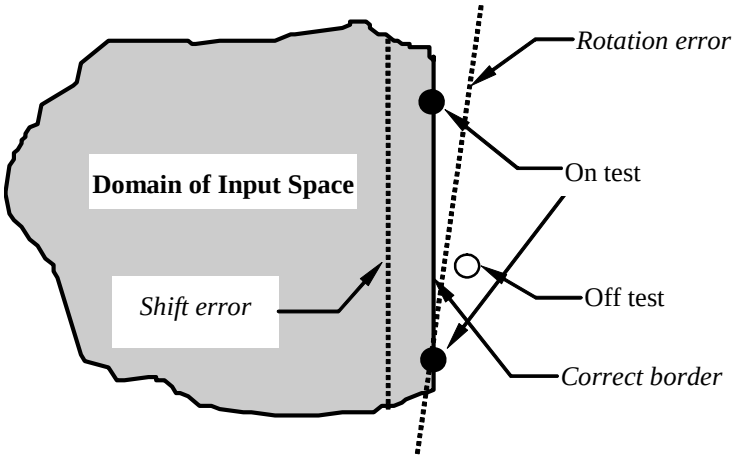

Figure 8. Illustration of boundary shift and rotation errors.

To detect shift errors of a border in *N*-dimensional input space, *N* test cases must be selected on the border and an additional test case must be selected nearby the border. If the input on the border belongs to the input domain, which are called on tests, the test case nearby the border must be selected outside the input domain, which is called off test. Otherwise, if the inputs on the border do not belong to the valid input of the domain, the test case nearby the border should be selected inside the input domain. In this case, the test cases on the border are off tests while the test case nearby the border is on test. As illustrated in Figure 9 for 2-dimensional input spaces, by selecting data according to this *N×1 criterion*, all shift errors can be detected provided that the computed functions in the input domain and outside the domain are different and the border is linear; e.g. a straight line in 2 dimensional space [49].

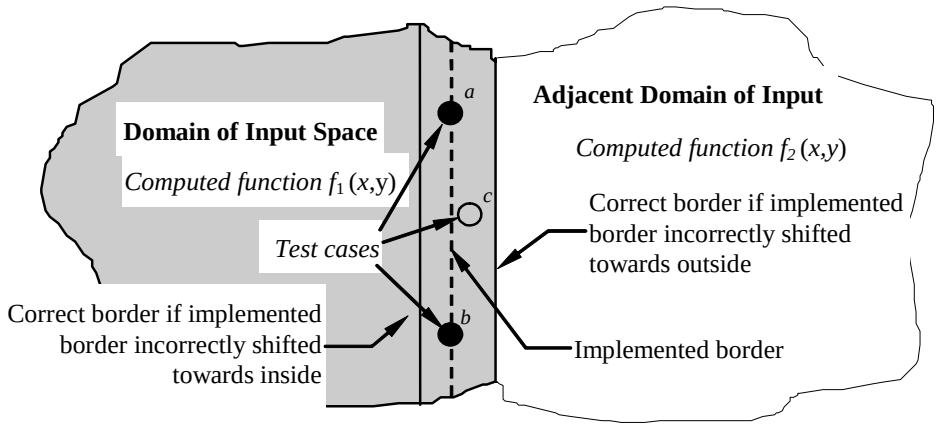

Figure 9. Selection of test cases using N×1 criterion.

However, $N\times1$ criterion cannot guarantee the detection of rotation errors. To detect rotation errors, in addition to the selection of N test cases on the border, N test cases must also be selected nearby the border in the same way as the $N\times1$ criterion. It is the so called $N\times N$ criterion [50].

References

[1] Wallace D. R. and R. U. Fujii. Software verification and validation: an overview, IEEE Software, Vol. 6, No. 3, pp. 10-17, May 1989.

[2] Boehm B.W., Software engineering economics, IEEE Trans. on Software Eng., Vol. 10, no. 1, Jan. 1984. pp. 4 - 21.

[3] Gild, T. and Gramham, D., Software Inspection. Addison-Wesley, 1993.

[4] Wheele, D. A. (ed.), Software Inspection: An Industry Best Practice. IEEE Computer Society Press, 1996.

[5] Yourdon, E., Structured Walkthroughs (4th ed.). Prentice-Hall International, Englewood Cliffs, London, 1989.

[6] Belady L.A. and M.M. Lehman, A model of large program development, IBM Systems Journal, Vol.3, 1976. pp. 225 - 252.

[7] Monty Newborn, ``Automated Theorem Proving: Theory and Practice," Springer, 2000.

[8] Hoare, C.A.R., An axiomatic basis for computer programming, CACM vol.12, no.10, 1969. pp. 576 - 583.

[9] Hoare, C. A. R., et al, Laws of programming, CACM, Vol. 30, No. 8, August 1987. pp. 672 - 687.

[10] Robinson J.A., ``A machine-oriented logic based on the resolution principle," Journal of ACM, vol.12, no.1, Jan. 1965. pp. 23 - 41.

[11] McCarthy, J. and P. Hayes, Some philosophical problems from the stanpoint of artificial intelligence, in Machine Intelligence no.4, B.Meltzer and D.Michie Edinburgh University Press, Edinburgh, 1969. pp. 463 - 502.

[12] Edmund M, Clarke, Jr., Orna Grumberg, and D.A. Peled, ``Model Checking," The MIT Press, 1999.

[13] E. M. Clarke, ``Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," ACM Transactions on Programming Languages and Systems, Vol. 8, No. 2, pp. 244 - 263, April, 1986.

[14] Gerard J. Holzmann, ``The Model Checker SPIN," IEEE Transactions on Software Engineering, Vol. 23, No. 5, May 1997.

[15] T. Henzinger et al., ``Symbolic Model Checking for Real-Time Systems," Proceedings, The Seventh Annual IEEE Symposium on Logic in Computer Science, pp. 394-406, 1992.

[16] W. Visser, K. Havelund, G. Brat, S. Park and F. Lerda, ``Model Checking Programs," Automated Software Engineering Journal, Volume 10, Number 2, April 2003.

[17] Dwyer, Matthew B., Hatcliff, John, Robby, Robby, Pasareanu, Corina S., Visser, Willem, ``Formal Software Analysis Emerging Trends in Software Model Checking," Future of Software Engineering, May 23-25, 2007. pp. 120 - 136.
[18] Chandra, S., Godefroid, P., Palm, C., ``Software model checking in practice: an industrial case study," Proceedings of the 24th International Conference on Software Engineering, 2002. pp. 431 - 441.
[19] Zhu, H., Hall, P. and May, J., Software unit test coverage and adequacy, ACM Computing Surveys, Vol. 29, No. 4, Dec. 1997, pp366-427.
[20] Bernot, G., Gaudel, M. C. and Marre, B., Software testing based on formal specifications: a theory and a tool, Software Engineering Journal, Nov. 1991, pp387- 405.
[21] Zhu, H., A Note on Test Oracles and Semantics of Algebraic Specifications, Proc. of QSIC'03, Dallas, USA, Nov. 2003, pp91-99.
[22] Chen, H. Y., Tse T. H. and Chen, T. Y., TACCLE: a methodology for object-oriented software testing at the class and cluster levels, ACM TSEM, Vol. 10, No.1, Jan. 2001, pp56-109.
[23] Chen, T.Y., Tse, T.H. and Zhou, Z.Q., Fault-based testing without the need of oracles, Information and Software Technology, Vol.45, No.1, 2003, pp. 1-9.
[24] Bouge, L., Choquet, N., Fribourg, L. and Gaudel, M. C., Test sets generation from algebraic specifications using logic programming, J. of Systems and Software Vol. 6, No.4, 1986, pp343 -360.
[25] Stocks, P.A. and Carrington, D. A., Test templates: A specification-based testing framework, Proc. of ICSE'93, pp405~414, May 1993.
[26] Ammann, P. and Offutt, J., Using formal methods to derive test frames in category-partition testing, Proceedings of 9'th Annual Conf. on Computer Assurance, IEEE, Gaithersburg, MD, USA, June 1994, pp69-79,
[27] Ammann, P., Black, P. E. and Majurski. W., Using model checking to generate tests from speci_cations. Proc. of 2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98), page 46, Brisbane, Australia, Dec. 1998.
[28] Hong, H. S., Cha, S. D., Lee, I., Sokolsky, O. and Ural, H., Data flow testing as model checking. Proc. of ICSE'03, Portland, Oregon, May 3-10, 2003.
[29] Zhu, H., Jin, L., Diaper, D., Software requirements validation via task analysis, Journal of System and Software, March 2002, Vol 61, Issue 2, pp145~169.
[30] Offutt, J. and Abdurazik, A., Using UML Collaboration Diagrams for Static Checking and Test Generation. The Third International Conference on the Unified Modeling Language (UML '00), pp383-395, York, UK, October 2000.
[31] Li, S., Wang, J., and Qi, Z., Property-oriented Test Generation from UML Statecharts, Proceedings of the 19th International Conference on Automated Software Engineering (ASE'04).
[32] RTCA/DO-178B. Software considerations in airborne systems and equipment certification, December 1992.
[33] Chilenski, J., An investigation of three forms of the Modified Condition Decision Coverage (MCDC) criterion. Technical Report DOT/FAA/AR-01/18, FAA, Washington, D.C., 2001.
[34] Goodenough, J.B. and Gerhart, S.L., Toward a theory of test data selection, IEEE TSE, Vol.SE_3, June 1975.
[35] Rapps, S. and Weyuker, E.J., Selecting software test data using data flow information, IEEE TSE, Vol.SE_11, No.4, pp367-375, April 1985.
[36] Frankl, P.G. & Weyuker, J.E., An applicable family of data flow testing criteria, IEEE TSE, Vol.SE_14, No.10, pp1483-1498, October 1988.
[37] Laski, J. and Korel, B, A data flow oriented program testing strategy, IEEE TSE, Vol. SE-9,pp33-43, May 1983.
[38] Ntafos, S.C., On required element testing, IEEE TSE, Vol. SE_10, No. 6, pp795-803, November 1984.
[39] DeMillo, R.A., Lipton, R.J. & Sayward, F.G., Hints on test data selection: Help for the practising programmer, Computer, Vol.11, pp34-41, April 1978.
[40] Hamlet, R. G., Testing programs with the aid of a compiler, IEEE TSE, Vol. 3, No. 4, July 1977, pp279-290.
[41] Budd, T. A., Mutation analysis: Ideas, examples, problems and prospects, in Chandrasekaran, B., and Radicchi, S., (eds), Computer Program Testing, North- Holland, 1981, pp129-148.
[42] King, K.N. and Offutt, A.J., A FORTRAN language system for mutation-based software testing, Software--Practice and Experience, Vol.21, No.7, pp685-718, July 1991.

[43] Delamaro, M. E., Maldonado, J. C. and Mathur, A. P., Integration Testing Using Interface Mutation. In Proceedings of International Symposium on Software Reliability Engineering (ISSRE ’96), pp112-121, April 1996.
[44] Kim, S., Clark, J. and McDermid, J., Class mutation: Mutation testing for object-oriented programs. Proc. of Net.ObjectDays Conference on Object-OrientedSoftware Systems, October 2000.
[45] Ma, Y. S., Offutt, J. and Kwon, Y. R., MuJava: An automated class mutation system. Software Testing,Verification and Reliability, Vol.15, No.2, pp97-133, June 2005.
[46] DeMillo, R.A. and Offutt, A.J., Experimental results from an automatic test case generator, ACM Transactions on Software Engineering and Methodology, Vol.2, No. 2, April 1993, pp109-127.
[47] Woodward, M.R., Errors in algebraic specifications and an experimental mutation testing tool, SEJ, July 1993, pp211-224.
[48] Shan, L. and Zhu, H., Testing Software Modelling Tools Using Data Mutation, Proc. of ICSE’06-AST’06, Shanghai, China, May 23, 2006, ACM Press, pp43-49.
[49] White, L.J., and Cohen, E.I., A domain strategy for computer program testing, IEEE TSE, Vol.SE_6, No.3, pp247-257, May 1980.
[50] Clarke, L.A., Hassell, J. and Richardson, D.J., A close look at domain testing, IEEE TSE, Vol.SE-8, No.4, pp380-390, July 1982.