Software Verification and Testing

Lecture Notes: Testing II

Functional Testing

idea: derive test cases from component specification

- component as black box, i.e., invisible for tester
- appropriate for analysing global functionality of component
- specification can be formal (Z) or informal
  (e.g., interface with natural language documentation)

problem:

- complete functional testing usually not feasible
- selection of test cases should yield high probability of error detection

remark: this is also called conformance testing

Functional Testing

techniques for functional testing

- functional equivalence classes
- boundary value analysis
- testing of special values
- random testing
- state automata testing

Functional Equivalence Classes

idea: partition the sets of input- and output-parameters into equivalence classes

mathematical background:

- a partition of a set S is a family $S_1 \ldots S_n$ of subsets of S such that $S_i \cap S_j = \varnothing$ (pairwise) and $S_1 \cup \cdots \cup S_n = S$
- an equivalence is a reflexive, symmetric and transitive relation
- the equivalence class of an $x \in S$ wrt equivalence $\equiv$ is $[x] = \{y \in S \mid y \equiv x\}$
- every partition defines an equivalence
- every equivalence defines a partition
- definitions extend to infinite families. . .

Functional Equivalence Classes

idea: partition the sets of input- and output-parameters into equivalence classes

- test for one representative of each class
- hope that the other elements in a class behave like the representative

remark:

- because of heuristic nature, one usually neglects that classes should be intersection-free
- "equivalence classes" are not meant in strict mathematical sense
- "representative testing" might be a better notion. . .

Functional Equivalence Classes

question: how can one identify equivalence classes?

heuristics: input equivalence classes

1. if an input-condition specifies a connected domain of values, then choose the domain and its complement
2. if an input-condition specifies a set of values with expected different behaviour, then choose one class per value and one class outside the domain
3. if an input-condition specifies a boolean decision, then choose two classes according to the truth value

heuristics: output equivalence classes

1. same conditions at inputs
2. input equivalence classes must correspond to output equivalence classes

Functional Equivalence Classes

example: counting digits again

- specification
  - the procedure reads input from the keyboard; it stops when some input is not an upper case character or some upper value `Max-Size` has been reached
  - if the input is an upper case character, then the counter `InCount` is incremented; if it is a vowel, then the counter `VoCount` is incremented
  - both counters are input and output parameters
  - the invariant `VoCount <= InCount` holds
- interface

```
void CountDigits( int &VoCount, int &InCount);
```

Functional Equivalence Classes

example: counting digits again

- equivalence classes for `InCount`

1. $0 \leq \texttt{InCount} < \texttt{MaxSize}$
2. $\texttt{InCount} = \texttt{MaxSize}$

- equivalence class for `VoCount`

3. $0 \leq \texttt{VoCount} \leq \texttt{InCount}$

- equivalence classes for keyboard input `Digit`

4. $\texttt{Digit} < '\texttt{A}'$
5. $'\texttt{A}' \leq \texttt{Digit} \leq '\texttt{Z}'$
6. $'\texttt{Z}' < \texttt{Digit}$
7. $\texttt{Digit} = '\texttt{A}'$
8. $\texttt{Digit} = '\texttt{E}'$
9. $\texttt{Digit} = '\texttt{I}'$
10. $\texttt{Digit} = '\texttt{O}'$
11. $\texttt{Digit} = '\texttt{U}'$

Functional Equivalence Classes

example: `SetMonth(short Month);`

- invariant $1 \leq \texttt{Month} \leq 12$
- equivalence classes
  1. $1 \leq \texttt{Month} \leq 12$
  2. $\texttt{Month} \leq 0$
  3. $13 \leq \texttt{Month}$
- derived test cases:
  1. $\texttt{Month} = 3$
  2. $\texttt{Month} = -7$
  3. $\texttt{Month} = 41$

Boundary Value Analysis

observation: values at boundaries of equivalence classes are often critical

example: lower and upper indices of loops

idea: test boundary values, when values can be ordered

example: for `SetMonth`, test values $0, 1, 12, 13$ of `Month`

extension: tests for critical values

- test for 0 in arithmetics
- test for special values (e.g., keyboard commands)

Random Values

remarks:

- no stand-alone technique
- useful in combination with other techniques, because testers often unintentionally generate patterns in test cases
- random value testing can be used to select representatives of equivalence classes

Testing State Automata

overview:

- state automata or state machines are similar to labelled transition systems
- they can be used for specifying reactive and concurrent systems
- examples are UML statecharts
- procedure is similar to statement coverage and branch coverage testing
- more in context of OO testing
- currently active research area (in our department)
- question: how to test liveness properties?

Functional and Structural Testing Combined

problems:

- structural testing
  - cannot detect absence of functionality
  - may generate trivial test cases that are irrelevant for functionality
- functional testing
  - does not consider implementation details

solution: combine the two

Functional and Structural Testing Combined

requirements: a testing method should

- satisfy minimal criteria like branch coverage, conformance with specification
- generate of error- and conformance-sensitive test-data
- be cost-effective
- be systematic and operational
- be intersubjective and reproducible
- use appropriate tools, including metrics and regression tests

Functional and Structural Testing Combined

requirements: a tester needs

- structural information
  - which parts of a components are affected by a test case
  - which parts are strongly used (information for optimisation)
- functional information
  - which parts of the specification are correctly implemented
  - whether domain and type constraints are satisfied
  - whether special cases and exceptions are handled correctly

Functional and Structural Testing Combined

requirements: the management is interested in organisatorial aspects

- how to control the progress of testing
- how to estimate the time required
- who tests what and to which extend
- how to quantify the degree of testing

Functional and Structural Testing Combined

procedure: assumes a testing tool, a specification and an implementation

1. functional testing
   - instrument test object with testing tool
   - identify equivalence classes, boundary values, special values, test cases
   - perform tests
2. structural testing
   - evaluate coverage metrics from functional testing
   - specify test cases for non-covered branches, paths or conditions (or delete these entities)
   - perform tests until the desired coverage is obtained
3. regression testing

Functional and Structural Testing Combined

question: how extensively should one test?

answer:

- no general answer possible
- often branch coverage of $80 - 99\%$ sufficient

industrial experience: (not mine)

- testing without tools yields poorer quality
- functional test cases should be developed in design phase
- identifying functional test cases requires much thought and experience
- functional testing alone is usually not sufficient

Testing Object-Oriented Components

particularities:

- functional testing must be modified, since objects are parametrised by their states
- encapsulation makes it difficult to directly observe the current state
- classes can be instantiated in various ways (that cannot all be tested)
- inheritance creates additional dependencies
- polymorphism and dynamic binding require new techniques
- genericity (C++ templates) requires types to be instantiated before testing

question: can a design and programming paradigm be good if it makes testing difficult?

Testing Object-Oriented Components

example: modification of a subclass requires retesting all methods inherited from its superclasses

- consider a window-manager class and its subclass for the Linux OS
- assume that some method initialises the screen to blank
- let some other method that sets the screed background
- the order of invoking these methods can be different in different contexts

conclusion: OO simplifies design, but complicates testing

- combining encapsulation with inheritance requires integrated instead of reduced testing
- inheritance makes changes non-local and dependent on complex mechanisms

Object-Oriented Specification

design by contract: OO development by realising contract between designer and implementor

- designer is responsible for respecting publicly available services in an interface
- implementor is responsible for providing the functionality specified by interface

remarks: helpful view for

- specifying and refining preconditions, postconditions and invariants of a system
- distinguishing the "two sides" of an interface
- specification, implementation and refinement of OO systems

literature: B. Meyer, Object-Oriented Software Construction, Prentice Call, 1988.

Class Testing

problems:

- classes can only be tested via instances
- data flow analysis is complicated by dynamic binding
- sequences of method calls must be tested

Class Testing

useful test scenario:

1. generate an instrumented object of the class to be tested
2. test each individual method; first getters and then setters
   - define test cases by equivalence classes and boundary values; initialise object appropriately
   - test object state and output parameters after each method execution
3. test execution sequences of dependent methods (all potential uses in practical applications)
   - in presence of object life-cycle, perform statement or branch coverage test
4. use instrumentation to check coverage; perform additional tests

Class Testing

testing environment:

1. test driver: simulates inherited attributes and methods, triggers execution of selected methods
2. message generator: generates input parameters
3. object initialiser: brings object in desired precondition
4. state evaluator: checks object state and output parameters after method execution against postconditions
5. test monitor: manages method execution sequences

Class Testing

testing levels:

1. instance testing: select representative objects to be tested
2. context testing: test objects in all relevant dependencies:
   - message exchanges
   - exceptions
   - potential dynamic bindings
3. completeness testing: coverage of methods and methods sequences, object dynamics
4. state model testing: generate all relevant states of objects and all state transitions

Class Testing

testing abstract classes:

- derive concrete class
- implement abstract methods as simply as possible

testing parametrised classes:

- generate very simple concrete classes
- chose parameters wrt simplicity of testing

Subclass Testing

recipes:

- all test cases for inherited and non-redefined methods must be repeated
- new test cases for redefined methods must be developed
- in case of method refinement: old test cases can be reused,
  but new ones might be necessary

Example: Method Sequence Testing

observations:

- result of method execution depends on “history” of previous method executions
- method sequencing is constrained by system functionality (stack object initialised as empty should accept pop messages only after push messages)

idea: specify method sequences as regular expressions

Example: Method Sequence Testing

example: a simple bank account class

- interface methods: $Create$, $Deposit$, $Open$, $Withdraw$, $Close$, $Delete$
- transaction method sequences: $TransSeq = Deposit \cdot (Deposit \cup Withdraw)^*$
- account method sequences: $AccSeq = Open \cdot TransSeq \cdot Close$
- method sequences: $MethodSeq = Create \cdot AccSeq \cdot Delete$

- $MethodSeq$ defines the set of valid messages accepted by the account class
- the following sequences are in $MethodSeq$:

$$Create \cdot Open \cdot Deposit \cdot Withdraw \cdot Close \cdot Delete$$

$$Create \cdot Open \cdot Deposit \cdot Deposit \cdot Close \cdot Delete$$

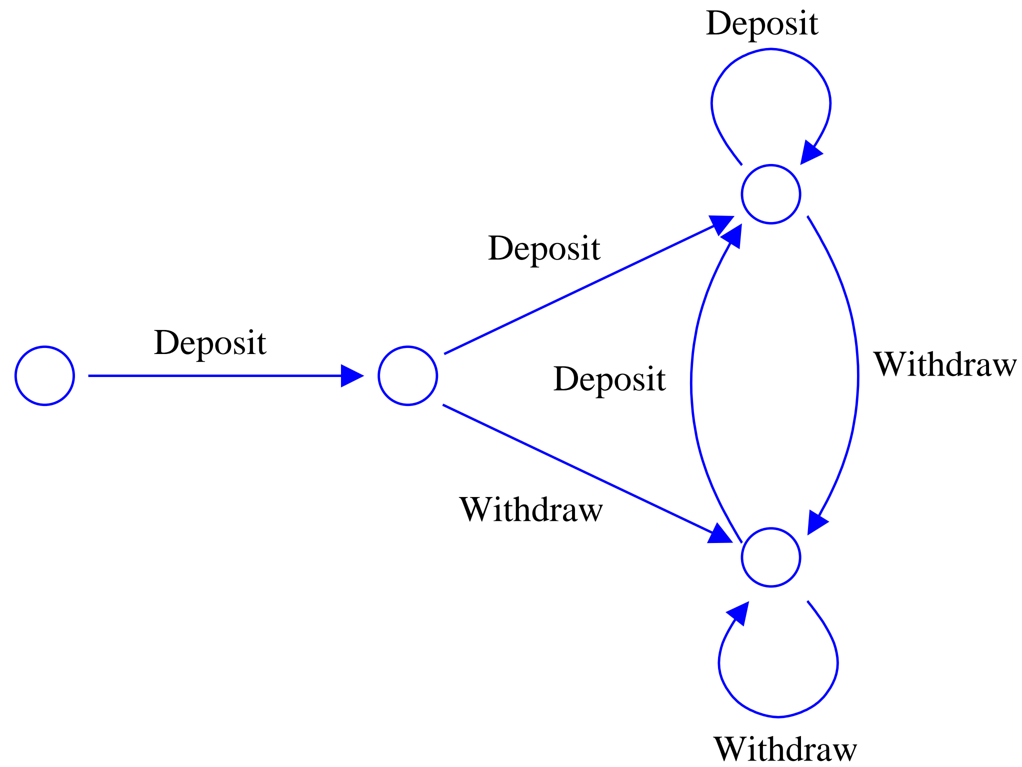
Example: Method Sequence Testing

method sequences:

- they specify the intended execution sequences of a system
- execution sequences of the methods implemented can be tested against these sequences
- state coverage, branch coverage or path coverage can be applied
- message-sequence diagrams can be very useful in this context

Example: Method Sequence Testing

example: testing $TransSeq$



Example: Method Sequence Testing

example: testing $TransSeq$

- node coverage:

$$Deposit \cdot Deposit$$

$$Deposit \cdot Withdraw$$

- branch coverage:

$$Deposit \cdot Deposit \cdot Deposit \cdot Withdraw$$

$$Deposit \cdot Withdraw \cdot Withdraw \cdot Deposit$$

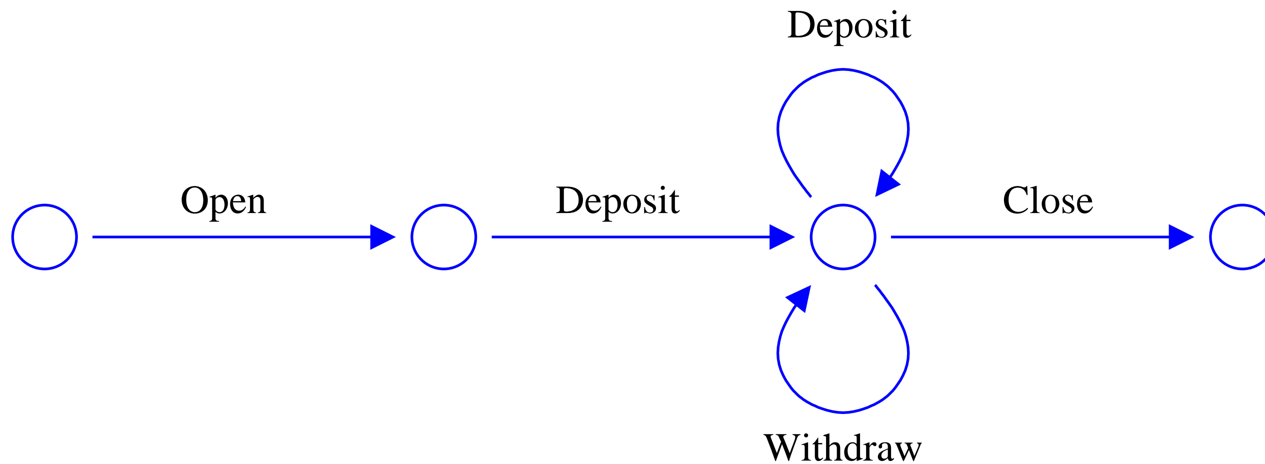
Example: Method Sequence Testing

deriving method sequences:

- present design as transition system (state chart)
- method sequences present all possible paths (obtained by unwinding)
- this is called the language accepted by transition system
- process can easily be automated ("Kleene's theorem")

Example: Method Sequence Testing

example: bank account LTS



regular expression derived: $Open \cdot Deposit \cdot (Deposit \cup Withdraw)^* \cdot Close$

Example: Method Sequence Testing

inheritance:

- specialisation inheritance: child class can contain additional methods
  semantics of inherited methods is preserved
- refinement inheritance: child methods can be redefined
  semantics of inherited methods is changed
  (preconditions weakened, postconditions strengthened)
- implementation inheritance: not all methods need be inherited,
  some can be redefined

Example: Method Sequence Testing

refinement inheritance: parent method sequences of are obtained
by renaming some child method sequences by parent methods

intuition: child method sequences can safely be used for parent methods;
but they can introduce additional behaviour

example: refine $Deposit$ by $Deposit \cdot Interest$ for savings account class

implementation inheritance: combination of refinement and specialisation
inheritance

Integration Testing

task:

- test interplay of components in a program
- this is a variant of structural testing

test drivers: feeds components (interfaces) with parameters needed

dummies/stubs: simulate components needed that are specified, but not yet implemented

Integration Testing

strategies:

- incremental/non-incremental
- test-goal driven: integrate components as needed to perform tests
- architecture driven (obvious)
- business-process driven: integrate components as needed for use case/business process
- function driven: integrate components to achieve specification goal
- availability driven (obvious)

Integration Testing

architecture driven testing: top-down integration

- features
  - start with high-level system components (e.g., GUIs)
  - simulate lower levels by stubs
- pros:
  - no test drivers needed
  - early prototypes allow client to simulate system use
  - easy to detect and handle changing requirements
  - design and implementation can be interleaved
- cons:
  - stubs can be difficult to implement
  - test case generation for low-level components can become quite difficult
  - interoperability of software with system software and hardware is tested very late

Integration Testing

architecture driven testing: bottom-up integration

- features
  - start testing components that require no further services
  - integrate from level to level
- pros:
  - no stubs needed
  - test cases simple to develop and to interpret
  - interoperability of software with system software and hardware is tested very early
- cons:
  - test drivers needed
  - running system obtained very late; errors in global functionality occur late
  - exception handling difficult to test, since parameter are given by components that have already been tested

Integration Testing

dynamic integration testing:

- control flow oriented: test call-dependencies between components
  - consider every call/execution of an exported/imported operation
  - consider call sequences of imported operations
- data flow oriented: test data flow between components
  - of input parameters and global variables before and after executing an imported operation
  - of output parameters after operation calls
- functional testing: as usual

System Testing

task:

- final test of software in real environment before delivery
- this is a variant of functional testing
- client is not yet involved

conformance: product-model, requirement, GUI-concept, user manual

System Testing

function testing:

- are all required functions present and working?
- test sequences defined in requirement specification

performance testing:

- load testing: how does the system work with massive data?
- time testing: are timing constraints satisfied?
- reliability testing: how does the system work under peak loads?
- stress testing: how does the system react beyond load limits?
- usability testing: is the system comprehensive and usable?
- security testing: does the system provide data security?
- interoperability testing: how does the system behave in interaction?
- configuration testing: how does the system behave on different platforms?
- document testing: are the user manual etc. useful?

System Testing

acceptance testing: system test together with client

- tests in normal working environment
- test cases for endurance testing

important points:

- configuration control
- system configuration from source programs
- testing according to predefined procedure, incl. user manual
- outcome of tests documented in protocol
- "free testing" after every testing step
- acceptance after final meeting in which errors are weighted and discussed

System Testing

alpha testing: system test in target environment by client

beta testing: system test by selected clients

Product Certification

idea: the quality of a software product is the result of the process quality

measurement: this can be collected in process or system certifications

standardisation: there are different standards in different countries

certification: given by independent accredited agencies

Testing Process and Documentation

testing process: three steps

- test planning (documented in test plan, test specification)
- test performance (documented in test plan, test specification)
- test control (documented in test report)

remark: there are several ANSI/IEEE norms for test documentation

- ANSI/IEEE Std 829–1983 Software Test Documentation
- ANSI/IEEE Std 1008–1987 Standard for Software Unit Testing
- ANSI/IEEE Std 1012–1986 Standard Verification and Validation Plans

Who verifies the verifier?

Who tests the tester?