18. Specification Testing

The outline of this part:

- Motivation
- The process of specification testing
- The Principle of specification testing
- Testing predicate expressions
- Unit testing
- Integration testing
- A prototype tool for specification testing
- Related publications
18.1 Motivation

- Remove faults in high level documentation (e.g., specification, design) helps to reduce the cost of software production, because detection and elimination of faults in high level documentation is less costly than those for lower level documentation (e.g., detailed design and program).

- Testing is the most effective and practical method for validation of software systems in all the existing verification and validation techniques (formal proof, model checking, inspection etc), provided that the systems are executable.
Formal specifications may not be executable (e.g., specifications in pre-post style) and how to verify and validate them by testing becomes an interesting problem.

The cost of software testing can be considerably reduced if the test cases generated for specification testing can be reused in program testing.
18.2 The Process of Specification Testing

The process of specification testing consists of the following three steps:

- Test cases generation
- Evaluation of specification
- Test result analysis
A simple example

Let process **SQRT** be defined as follows:

```plaintext
process SQRT(x: int) y: int
pre x >= 0
post y ** 2 = x
end_process
```

This specification is a non-deterministic specification, because there are two possibilities of \( y \) for the same \( x \).
A test for the process

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>pre</th>
<th>post</th>
<th>pre =&gt; post</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>
## The differences between specification and program testing

<table>
<thead>
<tr>
<th>Program testing</th>
<th>Specification testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test cases are generated only for input variables (for running the program)</td>
<td>Test cases need to be generated for both input and output variables</td>
</tr>
<tr>
<td>The testing target is program</td>
<td>The testing target is specification</td>
</tr>
<tr>
<td>Execution of program is necessary (dynamic testing)</td>
<td>Execution of program is not required</td>
</tr>
</tbody>
</table>
The differences between specification testing and specification-based testing

<table>
<thead>
<tr>
<th>Specification-based testing</th>
<th>Specification testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>The testing target is program</td>
<td>The testing target is specification</td>
</tr>
<tr>
<td>The objective is to verify whether the program satisfies its specification</td>
<td>The objective is to verify the consistency and validity of specification</td>
</tr>
<tr>
<td>Test cases are generated based on the structure of specification</td>
<td>Test cases are generated based on both domain knowledge and the structure of specification</td>
</tr>
<tr>
<td>---------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Specification can serve as a test oracle</td>
<td>Specification may not function as a test oracle (i.e., whether a test has detected faults may not be decided based on the specification)</td>
</tr>
<tr>
<td>Execution of program is necessary</td>
<td>Execution of program is not required</td>
</tr>
</tbody>
</table>
18.3. The Principle of Specification Testing

(1) Testing consistency

The process of testing is:

Step 1: derive proof obligations (predicate expressions) that ensures the consistency properties of the specification.
Step 2: test the proof obligation with test cases.
Step 3: Evaluate the test results.
(2) Testing validity
The process of testing is:
Sept 1: obtain all the required functions from the domain description (usually informal).
Sept 2: Generate test cases and expected test results based on the functions.
Step 3: Analyze the test results by comparing the actual test results and the expected results.
(3). The way of testing an entire specification

A general structure of SOFL specification:

```
class S1;
const; type; var; inv;
method Init;
method P1;
method P2;
method P3;
end-class;
```

```
class S2;
const; type; var; inv;
method Init;
method Q1;
method Q2;
method Q3;
end-class;
```

```
module SYSTEM;
const; type; var; inv;
process Init;
process A1;
process A2;
end-module;
```

```
module A2-decom;
const; type; var; inv;
process Init;
process B1;
process B2;
process B3;
end-module;
```

```
module A2-decom;
const; type; var; inv;
process Init;
process B1;
process B2;
process B3;
end-module;
```

Classes  Modules  CDFDs
A simple example:

A1: [input, output, pre_A1, post_A2]
A2: [input, output, pre_A2, post_A2]
A3: [input, output, pre_A3, post_A3]
18.4 Testing Predicate Expressions

1. Testing criteria
Let \( P \equiv P_1 \lor P_2 \lor \cdots \lor P_n \) be a disjunctive normal form

\( P_i \equiv Q_{i1} \land Q_{i2} \land \cdots \land Q_{im} \) be a conjunction of atomic expression (e.g., relational expression, negation of relational expression, quantified expression)

where \( i = 1 \ldots n \) and \( j = 1 \ldots m \).
Criterion 1 Evaluate $P$ with a test set $T_d$ to true and false, respectively.
Criterion 2: Evaluate $P_i$ with $T_d$ for every $i = 1...n$ to true and false, respectively.

<table>
<thead>
<tr>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>...</th>
<th>$P_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>⋄</td>
<td>⋄</td>
<td>...</td>
<td>⋄</td>
</tr>
<tr>
<td>false</td>
<td>⋄</td>
<td>⋄</td>
<td>...</td>
<td>⋄</td>
</tr>
<tr>
<td>⋄</td>
<td>true</td>
<td>⋄</td>
<td>...</td>
<td>⋄</td>
</tr>
<tr>
<td>⋄</td>
<td>false</td>
<td>⋄</td>
<td>...</td>
<td>⋄</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>⋄</td>
<td>⋄</td>
<td>⋄</td>
<td>...</td>
<td>true</td>
</tr>
<tr>
<td>⋄</td>
<td>⋄</td>
<td>⋄</td>
<td>...</td>
<td>false</td>
</tr>
</tbody>
</table>

where ⋄ denotes either true or false.
Although it is possible to test the cases of each disjunctive clause $P_i$ being true and false respectively by using this criterion, we cannot guarantee to test the case that each disjunctive clause evaluates to true while every other disjunctive clauses as false.

For example, let $P_i$ be the following predicate expression:

$$x + y > 0 \text{ or } x - y < 0$$

where $x = 5$ and $y = 8$.

Thus, we may not be able to test what happens when disjunctive clause $P_i$ leads to the truth of the entire predicate expression independently.
Criterion 3 Evaluate $P_i$ with $T_d$ to true while all $P_1, \ldots, P_{i-1}, P_{i+1}, \ldots, P_n$ to false, and to false while all $P_1, \ldots, P_{i-1}, P_{i+1}, \ldots, P_n$ to true, respectively.

<table>
<thead>
<tr>
<th></th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>...</th>
<th>$P_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>...</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>...</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>...</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>...</td>
<td>true</td>
</tr>
</tbody>
</table>
However, for some predicate expressions, for example,

\[ x > 0 \text{ or } x > 3 \]

it is impossible to directly apply this criterion because when \( x > 3 \) evaluates to true, there is no way to evaluate \( x > 0 \) to false. In this case, the correctness of the predicate expression need to be checked, and other criteria may need to be applied.
Criterion 4 When evaluating $P_i$ to true with $T_d$, evaluate every $Q_j$ to true. When evaluating $P_i$ to false, evaluate each $Q_j$ ($j=1..m$) to false, respectively.

<table>
<thead>
<tr>
<th>Q_1</th>
<th>Q_2</th>
<th>Q_3</th>
<th>...</th>
<th>Q_m</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>...</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>⋆</td>
<td>⋆</td>
<td>...</td>
<td>⋆</td>
</tr>
<tr>
<td>⋆</td>
<td>false</td>
<td>⋆</td>
<td>...</td>
<td>⋆</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>...</td>
<td>.</td>
</tr>
<tr>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
<td>...</td>
<td>false</td>
</tr>
</tbody>
</table>
**Criterion 5** When evaluating $P_i$ to true with $T_d$, evaluate every $Q_j$ to true. When evaluating $P_i$ to false, evaluate $Q_j$ to false while all $Q_1,\ldots,Q_{j-1}, Q_{j+1}, \ldots, Q_m$ to true.

<table>
<thead>
<tr>
<th>Q_1</th>
<th>Q_2</th>
<th>Q_3</th>
<th>...</th>
<th>Q_m</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>...</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>...</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
<td>...</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>false</td>
<td>...</td>
<td>true</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>...</td>
<td>.</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>...</td>
<td>false</td>
</tr>
</tbody>
</table>
2. Test case generation for variables

Guidelines: let $T$ be a type and $x$ a variable of $T$.

(1) Boundary values, including lower and higher boundary values.
(2) Special values (e.g., 0)
(3) Outside values of type $T$
(4) Representative values within $T$
Examples:

(1) int

Boundary value:
  low boundary: minimum
  high boundary: maximum

Special value: 0

Outside value: a = 3.4

Representative value:
  \( i \in \{\text{minimum}, \ldots, \text{maximum}\} \)
(2) Set type: \( T = \text{set of int} \)

**Boundary value:**

- low boundary: \( \{ \} \) (empty set)
- high boundary: a reasonably big set

**Outside value:** 5

**Representative value:**

\[ s \in \{X \mid \forall x \in X \cdot x \in T\} \]
(3) Map type: $T = \text{map A to B}$

Boundary value:
- low boundary: $\{ \rightarrow \}$ (empty map)
- high boundary: a reasonably big map

Outside value: 2

Representative value: any nonempty map in type $T$
A unit can be any of the following:

1. **Type invariant**, defined in a module
2. **Process**
3. **Function**
4. **Method**, defined in a class
18.5.1 Invariant testing

type
  D = R; /*type D is declared as type R*/

inv
  forall[x: D] P(x)

Then type D is defined by both type R (constraints on structure of values of D) and the invariant (constraint on values of D).
The proof obligation for ensuring the consistency of the invariant is:

exists[r: R] | P(r)

Testing of this proof obligation ensures that type D defined by means of both R and the invariant is not empty. This also implies that P(x) given in the invariant is not a contradiction, therefore allowing the existence of a mathematical model for type D.
Example:

type
Customer = R;
R = composed of
    id: int
    name: string
end.

inv
forall[x : Customer] | x.id >= 10 and
                        x.id < 1000 and
                        len(x.name) <= 15
The proof obligation:

\[ \exists [r: R] \mid r.id \geq 10 \quad \text{and} \quad r.id < 1000 \quad \text{and} \quad \text{len}(r.name) \leq 15 \]

A test:

<table>
<thead>
<tr>
<th>r</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0000, &quot;Mark&quot;)</td>
<td>false</td>
</tr>
<tr>
<td>(0001, &quot;John&quot;)</td>
<td>false</td>
</tr>
<tr>
<td>(0011, &quot;Mike&quot;)</td>
<td>true</td>
</tr>
<tr>
<td>(0350, &quot;Chris&quot;)</td>
<td>true</td>
</tr>
<tr>
<td>(0023, &quot;Ginny&quot;)</td>
<td>true</td>
</tr>
</tbody>
</table>

sufficient
18.5.2 Process testing

Let a process $CP$ be defined as:

$CP: [I, O, pre, post]$

where

$I = \{x_1, x_2, \ldots, x_n, v_1, v_2, \ldots, v_p\}$,

$x_i$ (i=1..n) are input variables and

$\sim v_j$ (j=1..p) denote the values of external

variables $v_j$ before execution of process $CP$. 
O={y_1, y_2, ..., y_m, v_1, v_2, ..., v_p}, y_k (k=1..m) are output variables and v_j denote the values of the state variables after the execution of process CP

pre = pre_CP(x_1, x_2, ..., x_n, v_1, v_2, ..., v_p)
post = post_CP(x_1, x_2, ..., x_n, v_1, v_2, ..., v_p, v_1, v_2, ..., v_p, y_1, y_2, ..., y_m)
The proof obligation for CP to be satisfiable:

\[ \forall \text{Binding}(I) \mid \text{pre}(I) \Rightarrow \exists \text{Binding}(O) \mid \text{post}(I, O) \]

For example, the SQRT process is defined as:

\[ \text{SQRT}: \{x, y\}, x \geq 0, y^2 = x \]

The proof obligation:

\[ \forall x: \text{int} \mid x \geq 0 \Rightarrow \exists y: \text{int} \mid y^2 = x \]
(1) Satisfiability testing:

A test:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>pre</th>
<th>post</th>
<th>pre ⇒ post</th>
<th>Proof obligation</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

This test is called failed test.
A test:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>pre</th>
<th>post</th>
<th>pre (\Rightarrow) post</th>
<th>Proof obligation</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

This test is also a failed test.
A test:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>pre</th>
<th>post</th>
<th>pre $\Rightarrow$ post</th>
<th>Proof obligation</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

This test is called uncertain test.
(2) Validity testing:

Validity testing aims to check whether a process specification satisfies the user’s conception of requirements.

For a validity testing, we need to provide

- Test cases for input variables
- Test cases for output variables
- Expected test results
Example, to test the validity of process P:

```plaintext
process P(x:int) y: int
ext wr z: int
pre  x > 0 and z > 0
post  z > x + y + ~z
end_process
```

We generate the following test:
A test for validity:

<table>
<thead>
<tr>
<th>x</th>
<th>~z</th>
<th>z</th>
<th>y</th>
<th>pre</th>
<th>Exp_pre</th>
<th>post</th>
<th>Exp_post</th>
<th>pre =&gt; post</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>20</td>
<td>8</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>30</td>
<td>9</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>-5</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
<td>13</td>
<td>7</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
<td>40</td>
<td>12</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>
18.6 Integration testing

The goal:

Integration testing aims to detect faults leading to the violation of the consistency between the interfaces of processes in a CDFD.

Example:
Possible structures in a CDFD

(a)

(b)

(c)
(1) Testing sequential structure:

The proof obligation is:

\[
(\text{pre}_A1 \text{ and } \text{post}_A1(x1)) \text{ and } \\
(\text{pre}_A2 \text{ and } \text{post}_A2(x2)) \text{ and } \cdots \text{ and } \\
(\text{pre}_A\text{n} \text{ and } \text{post}_A\text{n}(x\text{n})) \implies \\
\text{pre}_B
\]

If some test case in a test fails to ensure the evaluation of this proof obligation to true, the test is a **successful test**, implying that a fault is detected; otherwise, the test is a failed test (finding no fault).
Example:
Let

A1: [\{\}, \{x1\}, true, x1 = 5]
A2: [\{\}, \{x2\}, true, x2 > 10]
A3: [\{\}, \{x3\}, true, x3 = 20]

B: [\{x1, x2, x3\}, \{\}, x1 + x2 + x3 < 30, true]

A test is given next.
<table>
<thead>
<tr>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>post_A1</th>
<th>post_A2</th>
<th>post_A3</th>
<th>pre_{B}</th>
<th>imp</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>15</td>
<td>20</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>20</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>5</td>
<td>30</td>
<td>20</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>20</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>12</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

This test is a failed test.
(2) Testing Conditional Structures

The proof obligation for the single-condition structure:

\[(\text{pre} \land \text{post}(x))[x1/x] \text{ and } \quad \text{C}(x)[x1/x] \quad \Rightarrow \quad \text{pre}_B(x1)\]
The proof obligation for the binary-condition structure:

\[(\text{pre} \land \text{post}(x))[x1/x] \text{ and } C(x)[x1/x]
\implies\pre_{\text{B1}}\]

\[(\text{pre} \land \text{post}(x))[x2/x] \land \neg C(x)[x2/x]
\implies\pre_{\text{B2}}\]
The proof obligation for the multiple-condition structure:

\[(\text{pre} \land \text{post}(x))[\xi/x] \text{ and } C_i(x)[\xi/x] \]
\[\Rightarrow \]
\[\text{pre}_B i \]

\[(\text{pre} \land \text{post}(x))[y/x] \land \neg(C_1(x) \text{ or } \cdots \text{ or } C_n(x))[y/x] \]
\[\Rightarrow \]
\[\text{pre}_A \]

where \(i = 1 \ldots n\).
(3) Test decompositions of processes
The proof obligation for the consistency between the high level process $A$ and its decomposition:

(1) $\text{pre}_A \Rightarrow \text{pre}_A1$
(2) $\text{pre}_A$ and $\text{post}_A1 \Rightarrow \text{pre}_A2$ and $\text{pre}_A3$
(3) $\text{pre}_A2$ and $\text{post}_A2 \Rightarrow \text{pre}_A4$
(4) $(\text{pre}_A3$ and $\text{post}_A3) \land (\text{pre}_A4$ and $\text{post}_A4)$
$\Rightarrow \text{post}_A$
18.7. A prototype tool for specification testing

A prototype tool, known as SOSTEM (SOfl Specification TEsting Machine), was built to support the specification testing technique. The functional services provided by the tool include:

- Interactive testing
- Batch testing
- Test results analysis
- Test data management (e.g., test cases, test targets)
18.8. Summary

(1) Specification testing is an effective technique for detecting faults of software systems in early phases (requirements, design) and for reducing the development cost.

(2) Testing formal specifications is more rigorous than testing informal specifications, and therefore has greater potential in detecting faults.

(3) Specification testing based on formal notation can be supported in depth by software tool than that based on informal notation.
Related publications


Exercises

1. Answer the following questions:
   a. What is a test case?
   b. What is a test set?
   c. What is a test suite?
   d. What is a test target?
   e. What are possible ways of generating test cases?
   f. What are the three steps for testing a specification?
   g. What is a failed test, a successful test, and an uncertain test?
   h. Is it possible to have a successful test for a process? If so, give an example. If not, explain why.
2. Generate a test based on Criterion 2 for process A1:

```plaintext
process A1(x: seq of nat0) d1, d2: seq of nat0
pre  len(x) > 0
post forall[a: elems(d1)] a < 60 and
     forall[b: elems(d2)] | b >= 60 and
     union(elems(d1), elems(d2)) = elems(x)
end_process;
```
3. Try to generate a test based on Criterion 3 for process A2 given below. Criterion 2 can be used when Criterion 3 is not applicable.

```plaintext
process A2(d1: seq of nat0) d3: seq of nat0
post d1 = [ ] and d3 = [ ] or
    d1 <> [ ] and
    subset(elems(d3), elems(d1)) and
    forall[e: elems(d3)] | e >= 40
end_process
```