Why inspection before testing?

(1) A test may not be carried out effectively due to either crash in execution or non-termination of the program.

(2) Not necessarily all the program paths can be tested.

(3) Even if every path is tested, there is no guarantee that every functional scenario defined in the specification is correctly implemented.
**Scenario-based inspection:** a strategy for `divide and conquer`
The principle of scenario-based inspection:

(1) Check whether every functional scenario defined in the specification is correctly implemented by a set of program paths in the program.

(2) Every program path contributes to the implementation of some scenario.

Formally, the principle is described as follows:

\[ M : S \rightarrow \text{power}(P) \]
\[ (\forall f \in S \ \exists q \in \text{power}(P) \cdot M(f) = q \land q \neq \{ \}) \land \\
(\forall p \in P \ \exists f \in S \cdot p \in M(f)) \]
A Process for Scenario-Based Inspection

1. Derive functional scenarios
2. Generate paths from the program
3. Link scenarios to paths
4. Inspection
5. Inspection report
6. checklist \{Q1, Q2, ..., Qu\}

scenarios \{f_1, f_2, ..., f_n\}
paths \{p_1, p_2, ..., p_m\}
16.1 Derivation of functional scenarios from a specification

Definition 1. Let $OP$ be an operation, $pre_{OP}$ denote its pre-condition, and $post_{OP} = C_1 \text{ and } D_1 \text{ or } C_2 \text{ and } D_2 \text{ or } \cdots \text{ or } C_n \text{ and } D_n$ be its post-condition, where $C_i (i \in \{1,\ldots,n\})$ is a guard condition and $D_i$ is a defining condition. Then, a functional scenario $fs$ of $OP$ is a conjunction $\neg pre\_OP \text{ and } C_i \text{ and } D_i$, and such a form of pre-post conditions is called functional scenario form or FSF for short. That is,

$(\neg pre\_OP \text{ and } C_1 \text{ and } D_1) \text{ or } (\neg pre\_OP \text{ and } C_2 \text{ and } D_2) \text{ or } \cdots \text{ or } (\neg pre\_OP \text{ and } C_n \text{ and } D_n)$
For example,

process A(x: int) y: int
pre  x > 0
post x > 10 and y = x + 1 or
     x <= 10 and y = x - 1
end_process
The steps for deriving scenarios

No.1 Transform post_OP into a disjunctive normal form.

No.2 Transform the disjunctive normal form into a functional scenarios form.

No.3 Obtain the set of functional scenarios from the functional scenario form and the ~pre_OP.
For example, suppose

```
process A(x: int) y: int
pre  x > 0
post x > 10 and y = x + 1 or
     x <= 10 and y = x - 1
end_process
```

No.1 Transform post-condition into a disjunctive normal form:

- $x > 10$ and $y = x + 1$
- $x \leq 10$ and $y = x - 1$
No.2 Transform the disjunctive normal form into the functional scenario form:

\[ x > 0 \text{ and } x > 10 \text{ and } y = x + 1 \text{ or } \\
\text{ } x > 0 \text{ and } x \leq 10 \text{ and } y = x - 1 \text{ or } \\
\text{ not } x > 0 \]

No.3 Obtain the following functional scenarios:

\[ f_1: \quad x > 0 \text{ and } x > 10 \text{ and } y = x + 1 \]
\[ f_2: \quad x > 0 \text{ and } x \leq 10 \text{ and } y = x - 1 \]
\[ f_3: \quad \text{not } x > 0 \]
More complicated example

Formal specification
process M(x, y: int) z: int
ext wr w: real
pre x <> y
post
x > 0 and
z = y / x and
w > w~**2 and
x >= y or
x > 0 and
z = x * y and
x < y and
w = z * w~ or
x = 0 and
z = y and
w = w~ or
x < 0 and
z = x + y + w~ and
w < w~

FSF of the specification
~pre_M and C1 and D1
or
~pre_M and C1 and D1
or
...
or
~pre_M and C1 and D1

C1: guard condition
D1: defining condition
~pre_M: pre-condition
Example

Formal specification

\[ M(x, y: \text{int})z: \text{int} \]
\[ \text{ext} \ w r \ w: \text{real} \]
\[ \text{pre} \ x \not= \ y \]
\[ \text{post} \]
\[ x > 0 \text{ and } \]
\[ z = y / x \text{ and } \]
\[ w > w^{**2} \text{ and } \]
\[ x \geq y \text{ or } \]
\[ x > 0 \text{ and } \]
\[ z = x * y \text{ and } \]
\[ x < y \text{ and } \]
\[ w = z * w^{~} \text{ or } \]
\[ x \not= y \text{ and } \]
\[ x < 0 \text{ and } \]
\[ z = x + y + w^{~} \text{ and } \]
\[ w < w^{~} \]
16.2 The Generation of execution paths from a program

For example,

```java
int A(int x) {
    if (x > 0) {
        if (x > 10) y = x + 1;
        else y = x - 1;
        return y;
    }
    else
        System.err.println("the pre is violated");
}
```

Generation of paths

1
- x > 0;
- x > 10;
- y = x + 1;
- return y;

2
- x > 0;
- x <= 10;
- y = x - 1;
- return y;

3
- x <= 0;
- System.err.println("the pre is violated");
16.3 Linking functional scenarios to their execution paths

Two strategies:

- **Forward linking**: from scenarios to paths.
- **Backward linking**: from paths to scenarios.
Techniques for the linking

- Identifying paths by testing, provided that the program can terminate normally.

- Identifying paths by comparing the logical expression of the functional scenario to the statements and conditions in the paths.
Functional scenarios in specification

f_1: \( x > 0 \) and \( x > 10 \) and \( y = x + 1 \)
f_2: \( x > 0 \) and \( x \leq 10 \) and \( y = x - 1 \)
f_3: \( x \leq 0 \)

Execution paths

1
\[
\begin{align*}
x & > 0; \\
x & > 10; \\
y & = x + 1; \\
\text{return } y;
\end{align*}
\]

2
\[
\begin{align*}
x & > 0; \\
x & \leq 10; \\
y & = x - 1; \\
\text{return } y;
\end{align*}
\]

3
\[
\begin{align*}
x & \leq 0; \\
\text{System.err.println("the pre is violated");}
\end{align*}
\]
16.4 Analyzing paths (two techniques)

- **Static checking based on a checklist.**

Example questions on the checklist:

1. Is the guard condition in the scenario implemented accurately in the paths?

2. Is every defining condition in the scenario implemented correctly in the paths?

3. Is every input, output, and external variable used in the scenario implemented properly in terms of its name, type, and use in the paths?

- **Walkthrough with test cases.**
f_1: x > 0 and x > 10 and y = x + 1
f_2: x > 0 and x <= 10 and y = x - 1
f_3: x <= 0

1
x > 0; x = 15
x > 10;
y = x + 1;
return y;
2
x > 0;
x <= 10;
y = x - 1;
return y;
3
x <= 0;
System.err.println("the pre is violated");
16.5 A Prototype Software Tool

Automatic transformation from a SOFL specification to a set of functional scenarios.
Automatic derivation of program paths from a Java method.
The goal:

Dynamically check whether the functions defined in the specification are `correctly` implemented by the program.

A program $P$ correctly implements a specification $S$ iff

$$\forall \sim \sigma \in \Sigma \cdot S_{pre}(\sim \sigma) \Rightarrow S_{post}(\sim \sigma, P(\sim \sigma))$$
17. Specification-based testing

What to do

Functional Specification

Transformation

How to do it

Program

Testing
The goal:
Dynamically check whether the functions defined in the specification are "correctly" implemented by the program.

A program $P$ correctly implements a specification $S$ iff

$$\forall \sim \sigma \in \Sigma \cdot S_{pre}(\sim \sigma) \Rightarrow S_{post}(\sim \sigma, P(\sim \sigma))$$
The features of specification-based testing:

(1) Test cases are generated on the basis of the specification.

(2) The program is executed using the test cases.

(3) Decisions about the existence of bugs in the program are made on the basis of the test cases, execution results, and the specification.
A Decompositional Approach to Automatic Test Case Generation Based on Formal Specifications
Overview

1. Goals of Automatic Specification-Based Testing
2. Strategy and Criteria for Test Case Generation
3. Algorithms for Test Set Generation
4. “Spring Vibration” Test Set Generation for a Relation
5. Test Result Analysis
7. Conclusion and Future Research
17.1 Goals of Automatic Specification-Based Testing
A Long Term Goal of Automatic Specification-Based Testing (ASBT)

Press a Button

Adequate test cases

Method(int x, int y, int z){
    int w;
    if(x < y) {
        w = y/x;
        while(w < z) {
            ...
        }
    } else {
        ...
    }
}

Next
The practical goals of automatic testing

1. Every function defined in the specification is tested (at least once) (**User’s view**).

2. Every representative program path is traversed or some required coverage criteria are satisfied. (**Program’s view**)
Our interest is how to automatically generate test cases from a pre-post style specification to allow every function defined in the specification to be tested at least once.

Let \( S(S_{\text{iv}}, S_{\text{ov}})[S_{\text{pre}}, S_{\text{post}}] \) denote an operation specification. A set of functional scenarios can be derived from the specification, each defining an independent function in terms of input-output relation.
Definition 1.1 (FSF)

Let $S_{\text{post}} \equiv C_1 \land D_1 \lor C_2 \land D_2 \lor \cdots \lor C_n \land D_n$, $C_i$: guard condition, $D_i$: defining condition. $i = 1, \ldots, n$.

Then, a functional scenario form (FSF) of $S$ is:

$$(\sim S_{\text{pre}} \land C_1 \land D_1) \lor (\sim S_{\text{pre}} \land C_2 \land D_2) \lor \cdots \lor (\sim S_{\text{pre}} \land C_n \land D_n)$$

where $f_i = \sim S_{\text{pre}} \land C_i \land D_i$ is called a functional scenario) and

$\sim S_{\text{pre}} \land C_i$ is called a testing condition
Example:

process A(x: int) y: int
pre x > 0
post (x > 10 => y = x + 1) and
   (x <= 10 => y = x - 1)

Functional scenarios:
(1) x > 0 ∧ x > 10 ∧ y = x + 1
(2) x > 0 ∧ x <= 10 ∧ y = x - 1

Test case generation

Specification

Program

Test result analysis
Test strategy:

Let operation $S$ have an FSF
$$(\neg S_{\text{pre}} \land C_1 \land D_1) \lor (\neg S_{\text{pre}} \land C_2 \land D_2) \lor \cdots \lor (\neg S_{\text{pre}} \land C_n \land D_n), \text{ where } (n \geq 1).$$

Let $T$ be a test set for $S$. Then, $T$ must satisfy the condition

$$\forall i \in \{1, \ldots, n\} \exists t \in T \cdot \neg \neg S_{\text{pre}}(t) \land C_i(t)) \text{ and } \exists t \in T \cdot \neg \neg S_{\text{pre}}(t)$$

where $\neg \neg S_{\text{pre}}(t)$ describes an exceptional situation.
Notation: \( G: L_{\text{E}} \rightarrow T_{\text{s}} \)

where

\( L_{\text{E}} \) is the universal set of logical expressions involved.

\( T_{\text{s}} \) is a set of test sets

**Criterion 1:**

\[ G((\neg S_{\text{pre}} \land C_1 \land D_1) \lor (\neg S_{\text{pre}} \land C_2 \land D_2) \lor \cdots \lor (\neg S_{\text{pre}} \land C_n \land D_n)) = \]

\[ G(\neg S_{\text{pre}} \land C_1 \land D_1) \lor G(\neg S_{\text{pre}} \land C_2 \land D_2) \lor \cdots \lor G(\neg S_{\text{pre}} \land C_n \land D_n). \]
Criterion 2:

Let \( \sim S_{pre} \land C_i \land D_i \) (\( i = 1, ..., n \)) be a functional scenario of specification \( S \). Then,

\[
G(\sim S_{pre} \land C_i \land D_i) = G(\sim S_{pre} \land C_i)
\]
Criterion 3:

Let \( P_1 \lor P_2 \lor \cdots \lor P_m \) be a DNF of the test condition \( \sim S_{pre} \land C_i \). Then, we define

\[
G(\sim S_{pre} \land C_i) =
G(P_1 \lor P_2 \lor \cdots \lor P_m) =
G(P_1) \cup G(P_2) \cup \cdots \cup G(P_m)
\]
Criterion 4:

Let \( S_{iv} = \{x_1, x_2, \ldots, x_r\} \) and \( Q(x_1, x_2, \ldots, x_q) \) (\( q \leq r \)) be a relation involving variables \( x_1, x_2, \ldots, x_q \).

Then,

\[
G(Q(x_1, x_2, \ldots, x_q)) = \{ T_c \mid (\forall x \in \{x_1, x_2, \ldots, x_q\} \cdot Q(T_c(x_1), T_c(x_2), \ldots, T_c(x_q))) \land (\forall x \in (S_{iv}\setminus\{x_1, x_2, \ldots, x_q\}) \cdot T_c(x) = \text{any})\}
\]

where \( T_c: S_{iv} \rightarrow \text{Values} \)

Example: \( G(x > y) = \{ \{(x, 5), (y, 3), (z, 8)\},\),

\[\{(x, 8), (y, 2), (z, 300)\} \} \)
Criterion 5:

Let $Q_i^1 \wedge Q_i^2 \wedge \cdots \wedge Q_i^w$ be a conjunction of $w$ atomic predicates in the test condition of a functional scenario of $S$. Then, we have

$$G(Q_i^1 \wedge Q_i^2 \wedge \cdots \wedge Q_i^w) = G(Q_i^1) \cap G(Q_i^2) \cap \ldots \cap G(Q_i^w)$$

Example: $G(x > y \wedge x < y + z) = G(x < y + z) \cap G(x > y) =$

$\{ \{(x, 5), (y, 3), (z, 8)\}\} \cap \{ \{(x, 5), (y, 3), (z, 8)\}, \{(x, 8), (y, 4), (z, 8)\}\} \}$
(2) Algorithms for Test Set Generation

(2.1) For Atomic Predicates

Let \( Q(x_1, x_2, \ldots, x_m) \) be an atomic predicate. It may have three formats:

(1) \( x_1 \Theta E \), where \( \Theta \in \{=, >, <, >=, <=, <>\} \), \( x_1 \) is a single variable, \( E \) a constant.

(2) \( E_1 \Theta E_2 \), where \( E_1 \) and \( E_2 \) are both arithmetic expressions that involve only variable \( x_1 \).

(3) \( E_1 \Theta E_2 \), where \( E_1 \) and \( E_2 \) are both arithmetic expressions that may involve variables \( x_1, x_2, \ldots, x_m \).
An algorithm of test case generation for \( x_1, x_2, \ldots, x_m \) in format (3) \( E_1 \ominus E_2 \):

**Step1** Randomly choose values \( v_2, v_3, \ldots, v_m \) from the corresponding types of variables in \( E_1 \ominus E_2 \);

**Step2** Substitute \( v_2, v_3, \ldots, v_m \) for variables \( x_2, x_3, \ldots, x_m \) in \( E_1 \ominus E_2 \);

**Step3** Convert \( E_1 \ominus E_2 \) to the format \( x_1 \ominus E \) by applying appropriate algorithms depending on \( \ominus \).
Algorithm 4: Let $Q_i^1 \land Q_i^2 \land \cdots \land Q_i^w$ be a conjunction of atomic predicate expressions. Let $x_1, x_2, \ldots, x_r$ be all input variables of the operation specification. Then, an algorithm for generating a test case from the conjunction is:
First applying the corresponding algorithm to generate a test case for $Q_i^1$, and then use it to evaluate $Q_i^2$, $\ldots$, $Q_i^w$. If all of them are true, the test case is generated; otherwise, repeating the same procedure until a test case is generated.
A simple algorithm of test case generation from the disjunction $P_1 \lor P_2 \lor \cdots \lor P_m$:

The essential part of the algorithm is a while-loop, which produces one test case from each disjunctive clause until all the disjunctive clauses are covered, and then form a test set that contains all of the produced test cases.
17.6 “Vibration” test set generation from a relation

Let $E_1(x_1, x_2, ..., x_q) \ominus E_2(x_1, x_2, ..., x_q)$ denote that expressions $E_1$ and $E_2$ have relation $R$, where $x_1, x_2, ..., x_q$ are all input variables involved in these expressions.
Applying the V-Method to this relation, we first produce values for $x_1, x_2, ..., x_q$ such that the relation $E_1(x_1, x_2, ..., x_q) \ominus E_2(x_1, x_2, ..., x_q)$ holds with any "distance" between $E_1$ and $E_2$, and then repeatedly create more values for the variables such that the relation still holds but the "distance" between $E_1$ and $E_2" vibrates" (changes repeatedly) between the initial “distance” and the “maximum” “distance”. 
Distance definition

Distance function definition (for nat0, nat, int, real types):

Distance(E₁, E₂, ">") ≡ E₁ - E₂
Distance(E₁, E₂, "<") ≡ E₂ - E₁
Distance(E₁, E₂, ">=") ≡ E₁ - E₂
Distance(E₁, E₂, "<=") ≡ E₂ - E₁
Distance(E₁, E₂, "=") ≡ 0
Distance(E₁, E₂, "<>") ≡ abs(E₁ - E₂)
Distance function definition for set types:

Distance($E_1$, $E_2$, "subset") $\equiv$ $\text{card}(E_2) - \text{card}(E_1)$

Distance($E_1$, $E_2$, "psubset") $\equiv$ $\text{card}(E_2) - \text{card}(E_1)$

Distance($E_1$, $E_2$, "inset") $\equiv$ $\text{card}(E_2) - \text{index}(E_1,E_2)$

Distance($E_1$, $E_2$, "notin") $\equiv$ $\text{card}(E_2)$

Distance($E_1$, $E_2$, "=") $\equiv$ 0

Distance($E_1$, $E_2$, "<>") $\equiv$ $\text{abs}($card($E_1$)-card($E_2$))
17.7 Test Result Analysis

Definition 3.1: If the condition
\[ \exists t \in T \cdot S_{\text{pre}}(t) \land \neg S_{\text{post}}(t, P(t)) \]
holds, it indicates the existence of a defect in program \( P \).
A(x: int) y: int
pre x > 0
post x > 10 ∧ y = x + 1 ∨
        x <= 10 ∧ y = x - 1

Functional scenarios:
(1) x > 0 ∧ x > 10 ∧ y = x + 1
(2) x > 0 ∧ x <= 10 ∧ y = x - 1
(3) x <= 0 (optional)

Test result analysis

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>Apre</th>
<th>Apost</th>
<th>Apre ∧ ¬Apost</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>15</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>
17.8 Process for Automatic Testing

Formal specification

\[
M(x, y: \text{int})z: \text{int} \\
\text{ext wr w: real} \\
\text{pre } x <> y \\
\text{post} \\
x > 0 \text{ and} \\
z = y / x \text{ and} \\
w > w^{-2} \text{ and} \\
x >= y \text{ or} \\
x > 0 \text{ and} \\
z = x * y \text{ and} \\
x < y \text{ and} \\
w = z * w^{-} \text{ or} \\
x = 0 \text{ and} \\
z = y \text{ and} \\
w = w^{-} \text{ or} \\
x < 0 \text{ and} \\
z = x + y + w^{-} \text{ and} \\
w < w^{-} \\
\]

**Step 1**

Transform pre-post specification into FSF

\[
\neg \text{pre}_M \text{ and } C1 \text{ and } D1 \text{ or} \\
\neg \text{pre}_M \text{ and } C1 \text{ and } D1 \text{ or} \\
\vdots \text{ or} \\
\neg \text{pre}_M \text{ and } C1 \text{ and } D1
\]

C1: guard condition
D1: defining condition
\neg \text{pre}_M: pre-condition
Example

Specification:
M(x, y: int)z: int
ext wr w: real
pre x <> y
post
x > 0 and
z = y / x and
w > w~**2 and
x >= y or
x > 0 and
z = x * y and
x < y and
w = z * w~ or
x = 0 and
z = y and
w = w~ or
x < 0 and
z = x + y + w~ and
w < w~
Step 2
Test case generation

Appropriate test set in specification format
{st1, st2, ..., stn}

FSF of specification:
~pre_M and C1 and D1
or
~pre_M and C1 and D1
or
... or
~pre_M and C1 and D1
Step 3
Transformation from specification test cases to program test cases

Appropriate test set in specification format
{st1, st2, ..., stn}

Test set in program format
{pt1, pt2, ..., ptn}
Step 4
Generation of test driving program
Step 5: Execution of program and the results management

- Appropriate test set in program format \{pt\_1, pt\_2, \ldots, pt\_n\}
- Program execution outputs and updated state variable values \{pr\_1, pr\_2, \ldots, pr\_n\}
- Traversed paths \{p\_1, p\_2, \ldots, p\_m\}
- Relation among test case, test result, and traversed paths \{(pt\_1, pr\_1, p\_1), \ldots, (pt\_n, pr\_n, p\_m)\}

Driving program
Step 6
Transformation from test results in program format

Program execution outputs and updated state variable values 
{pr1, pr2, …, prn}

Relation among test case, test result, and traversed paths 
{(pt1, pr1, p1),…, (ptn, prn, pm)}

Test results in specification format 
{sr1, sr2, …, srn}

Relation among test case, test result, and traversed paths 
{(st1, sr1, p1),…, (stn, srn, pm)}
Step 7
Determine bugs in the program

- Appropriate test case
  \{st_1, st_2, \ldots, st_n\}

- Test result
  \{sr_1, sr_2, \ldots, sr_n\}

- Relation among test cases that found bugs, test result, and traversed paths
  \{(st_1, sr_1, p_1), \ldots, (st_w, sr_w, p_w)\}
Relation among test Cases that found bugs, test result, and traversed paths
\{(st1, sr1, p1),..., (stw, srw, pw)\}

Step 8
Debugging

Debugging information
\{(bp1, bt1, bc1),..., (bpk, btk, bck)\}
17.9 Conclusion and Future Research

17.9.1 Conclusion:

(1) Automatic test case generation based on pre-post style formal specifications using our decompositional method is possible and beneficial, but not necessarily easy, especially when data structures involved are complex.

(2) Test result analysis based on formal specifications can be automatically done, but it would be extremely difficult for informal specification-based testing.
17.9.2 Future Research

(1) Develop more efficient algorithms for test case generation from a conjunction.

(2) Explore more effective methods for generating test cases that satisfy required coverage criteria.

(3) Build a powerful software tool that allows us to press one button for the detection of all potential bugs.
19. Intelligent Software Engineering Environments (ISEE)
What is an Intelligent software engineering environment?

**Description**  An intelligent software engineering environment (ISEE) is a collection of inter-related software tools to support software development and maintenance, with the feature that the ISEE guides and controls the development process.
The major potential features of ISEE

- The ISEE guides and controls the developers to use software tools in the environment based on the necessity of software development and maintenance.
- The ISEE guides and controls the management and progress of software projects.
- The human developers are treated as a “software tool” that supply intellectual inputs to the ISEE. In other words, the developers are under the control of the ISEE and guided by the ISEE to progress their development activities.
- The ISEE supports an effective integration of construction and verification of documents so that most errors may be prevented during the process of their constructions.
- The ISEE provides a user-friendly human-machine interface.
The types of ISEE

- **Domain-based ISEE** supports the development of application software in a specific domain (e.g., banking systems, railway control systems, library systems)

- **Method-based ISEE** supports the development of software systems using a specific method (e.g., VDM, B-Method, SOFL, UML)

- **Combination of domain-based and method-based ISEE** supports the development of application software in a specific domain using a specific method.
The abstract architecture of an ISEE

Knowledge base

Rule base (domain, method)

Documents

Control program

User’s input

Data

Instruction

Human-Machine Interface

Knowledge base

Rule base (domain, method)

Documents

Control program

User’s input

Data

Instruction

Human-Machine Interface
Necessary condition for building an effective ISEE

Formalization !!!

- Formalization of documents produced in software development, by adopting formal description languages.
- Formalization of the development methods by adopting well-defined and precise rules (e.g., rules for constructing specifications, inspection, and testing).
- Formalization of domain knowledge.
Future research

Build a more mature software engineering environment for SOFL on the basis of the existing prototype tools.

- Develop highly reliable, large-scale, and complex computer systems using SOFL under the support of its SEE
- Evolve the SEE of SOFL to a method-based ISEE.
- Extend the method-based ISEE to a method-domain-based ISEE to support domain specific applications.
Ongoing research projects

(1) Intelligent Tool Support for Requirements Analysis and Specification
(2) Pattern-Based Approach to Constructing Formal Specifications and Its Intelligent Support
(3) A Formal Engineering Framework for Service-Oriented Systems Development
(4) A Formal Engineering Approach to Safety-Critical Systems Development
(5) Integration of Prototyping and Formal Specification for Rigorous and Agile Software Development
(6) Specification Animation and Verification
(7) Automatic Specification-Based Testing and Verification
The end!