courses, such as "Software Quality Assurance" and "Software Verification and Validation". With its comprehensive coverage of all the major topics in software quality engineering in an integrated framework, this book is suitable as the main textbook for such a course.

In addition, this book could be used as a technical reference about software testing, QA, and quality engineering by other readers, particularly professionals who perform QA activities as testers, inspectors, analysts, coordinators, and so forth. It should also be useful to people involved in project planning and management, product release, and support. Similarly, this book could help prepare students for their internship assignments or future employment related to testing or QA.

For more information on this book, please visit the following website:

www.engr.smu.edu/∼tian/SQEbook/

Supplementary material for instructors is available at the Wiley.com product page:

www.wiley.com/WileyCDA/WileyTitle/productCd-0471713457.html

Acknowledgments

First, I thank all my students in the SMU/CSE 5314/7314 classes since 1995, particularly, Katherine Chen, Tony Cluff, DeLeon English, Janet Farrar, Nishchal Gupta, Gina Habash, Chris Jordan, Zhao Li, Sateesh Rudrangi, Zahra Shahim, and Nathan Vong, for reading the manuscript and offering many invaluable suggestions. I also thank Tim Culver, for sharing his detailed class notes with me, and Li Ma, for checking the exercise questions.

I thank the co-authors of my technical papers and the sponsors of my research projects for the material included in this book based on related publications. Since all these publications are individually cited in the bibliography, I only single out my project sponsors and industrial collaborators here: National Science Foundation, through awards MRI-9724517, CCR-9733588, and CCR-0204345; Texas Higher Education Coordinating Board, through awards 003613-0030-1999 and 003613-0030-2001; IBM, Nortel Networks, and Lockheed-Martin.

I am grateful to SMU for granting me a sabbatical leave for the 2003/2004 academic year to work on my research and to write this book. I thank my colleagues at SMU, particularly Prof. Hesham El-Rewini, for their encouragement and help. I also appreciate the opportunity to work for the IBM Software Solutions Toronto Laboratory between 1992 and 1995, where I gained invaluable practical experience in software QA and testing.

This book would not be possible without the love and support of my wife Sharon and my daughters Christine and Elizabeth. Sharon, a professional tester for many years, also helped me greatly by offering her invaluable technical critique. Utilizing her strength in reading and writing, Christine edited the entire manuscript (and many of my previous papers too).

I also thank my editor Val Moliere, her assistant Emily Simmons, and my production editor Melissa Yanuzzi, for their professional help.

JEFF (JIANHUI) TIAN

Plano, Texas

- The *errors* as missing or incorrect human actions are not directly depicted within one box, but rather as actions leading to the injection of faults in the middle box because of some error sources in the left box.
- Usage scenarios and execution results, depicted in the right box, describe the input to software execution, its expected dynamic behavior and output, and the overall results. A subset of these behavior patterns or results can be classified as failures when they deviate from the expected behavior, and is depicted as the collection of circled failure instances.

With the above definitions and interpretations, we can see that failures, faults, and errors are different aspects of defects. A causal relation exists among these three aspects of defects:

$$\text{errors} \rightarrow \text{faults} \rightarrow \text{failures}$$

That is, errors may cause faults to be injected into the software, and faults may cause failures when the software is executed. However, this relationship is not necessarily 1-to-1: A single error may cause many faults, such as in the case that a wrong algorithm is applied in multiple modules and causes multiple faults, and a single fault may cause many failures in repeated executions. Conversely, the same failure may be caused by several faults, such as an interface or interaction failure involving multiple modules, and the same fault may be there due to different errors. Figure 2.1 also illustrates some of these situations, as described below:

- The error source *e3* causes multiple faults, *f2* and *f3*.
- The fault *f1* is caused by multiple error sources, *e1* and *e2*.
- Sometimes, an error source, such as *e5*, may not cause any fault injection, and a fault, such as *f4*, may not cause any failure, under the given scenarios or circumstances. Such faults are typically called *dormant* or *latent* faults, which may still cause problems under a different set of scenarios or circumstances.

Correctness-centered properties and measurements

With the correctness focus adopted in this book and the binary partition of people into consumer and producer groups, we can define quality and related properties according to these views (internal views for producers vs. external views for consumers) and attributes (correctness vs. others) in Table 2.1.

The correctness-centered quality from the external view, or from the view of consumers (users and customers) of a software product or service, can be defined and measured by various failure-related properties and measurement. To a user or a customer, the primary concern is that the software operates without failure, or with as few failures as possible. When such failures or undesirable events do occur, the impact should be as little as possible. These concerns can be captured by various properties and related measurements, as follows:

- *Failure properties and direct failure measurement*: Failure properties include information about the specific failures, what they are, how they occur, etc. These properties can be measured directly by examining failure count, distribution, density, etc. We will examine detailed failure properties and measurements in connection with defect classification and analysis in Chapter 20.

removed as a part of or as follow-up to these activities. Consequently, no operational failures after product release will be caused by these faults.

- Still other faults, such as *f2*, are blocked through fault tolerance for some execution instances. However, fault-tolerance techniques typically do not identify and fix the underlying faults. Therefore, these faults could still lead to operational failures under different dynamic environments, such as *f2* leading to *x2*.

- Among the failure instances, failure containment strategy may be applied for those with severe consequences. For example, *x1* is such an instance, where failure containment is applied to it, as shown by the surrounding dotted circle.

We next survey different QA alternatives, organized in the above classification scheme, and provide pointers to related chapters where they are described in detail.

3.2 DEFECT PREVENTION

The QA alternatives commonly referred to as defect prevention activities can be used for most software systems to reduce the chance for defect injections and the subsequent cost to deal with these injected defects. Most of the defect prevention activities assume that there are known error sources or missing/incorrect actions that result in fault injections, as follows:

- If human misconceptions are the error sources, education and training can help us remove these error sources.

- If imprecise designs and implementations that deviate from product specifications or design intentions are the causes for faults, formal methods can help us prevent such deviations.

- If non-conformance to selected processes or standards is the problem that leads to fault injections, then process conformance or standard enforcement can help us prevent the injection of related faults.

- If certain tools or technologies can reduce fault injections under similar environments, they should be adopted.

Therefore, root cause analyses described in Chapter 21 are needed to establish these preconditions, or *root causes*, for injected or potential faults, so that appropriate defect prevention activities can be applied to prevent injection of similar faults in the future. Once such causal relations are established, appropriate QA activities can then be selected and applied for defect prevention or to implement a defect prevention process (**?**).

3.2.1 Education and training

Education and training provide people-based solutions for error source elimination. It has long been observed by software practitioners that the people factor is the most important factor that determines the quality and, ultimately, the success or failure of most software projects. Education and training of software professionals can help them control, manage, and improve the way they work. Such activities can also help ensure that they have few, if

- *Overall probability threshold* for complete end-to-end operations to ensure that commonly used complete operation sequences by target customers are covered and adequately tested.
- *Stationary probability threshold* to ensure that frequently visited states are covered and adequately tested.
- *Transition probability threshold* to ensure commonly used operation pairs, their interconnections and interfaces are covered and adequately tested.

To use the overall probability threshold, the probability for possible test cases (or complete operations) need be calculated and compared to this threshold. For example, the probability of the sequence ABCDEBCDC in Figure 10.3 can be calculated as the products of its transitions, that is,

$$1 \times 1 \times 0.99 \times 0.7 \times 1 \times 1 \times 0.99 \times 0.3 = 0.205821.$$

If this is above the overall end-to-end probability threshold, this test case will be selected and executed.

If the Markov chain is stationary, it can reach an equilibrium or become "stationary" (Karlin and Taylor, 1975). In such a state, the stationary probability π_i for being in state i remains the same before and after state transitions over time. The set $\{\pi_i\}$ can be obtained by solving the following set of equations:

$$\pi_j = \sum_i \pi_i p_{ij}, \quad \pi_i \geq 0, \quad \text{and} \quad \sum_i \pi_i = 1,$$

where p_{ij} is the transition probability from state i to state j. The stationary probability π_i indicates the relative frequency of visits to a specific state i after the Markov chain reaches this equilibrium. Therefore, testing states above a given threshold is to focus on frequently used individual operations or system states. For the many Markov chains that are not stationary (Karlin and Taylor, 1975), the same idea of focused testing can still be used by approximating stationary probabilities with the recorded relative frequencies of visit.

A mirror case to test states with stationary probabilities above a given threshold is to test links with transition probabilities above a given threshold. In this case, the testing is actually much easier to perform, because all the p_{ij}'s are specified in the UMMs. A larger value of p_{ij} indicates a commonly used operation (if we associate individual operations with transitions) or operational pair (if we associate individual operations with states) in the sense that whenever i is reached, j is likely to follow.

Some combinations of these thresholds could also be used if they make sense for some specialized situations. For example, if state i is visited very infrequently (low π_i), then even larger values of p_{ij} may not be that meaningful if state j is not tightly connected as the destination of other links (that is, low $p_{kj},\ k \neq i$). In this example, we would combine stationary probability threshold with link probability threshold to select our test cases.

10.5.2 Testing based on other criteria and UMM hierarchies

Coverage, importance and other information or criteria may also be used to generate test cases. In a sense, we need to generate test cases to reduce the risks involved in different usage scenarios and product components, and sometimes to identify such risks as well (Frankl and Weyuker, 2000). The direct risks involved in selective testing include missing important

$\{n \geq 1\}$
1 $y \leftarrow 1;$
2 $i \leftarrow n;$
3 `while` $i > 1$ `do`
4 `begin`
5 $\quad y \leftarrow y \times i;$
6 $\quad i \leftarrow i - 1;$
7 `end`
$\{y = n!\}$

Figure 15.1 A program segment with its formal specification

In addition, when we finished loop, we should have $i = 1$. Therefore, we select our loop invariant to be $I_1 \wedge (i \geq 1)$, or :

$$I \equiv \left(y = \frac{n!}{i!}\right) \wedge (i \geq 1).$$

The loop condition is: $B \equiv i > 1$, and $\neg B \equiv i \leq 1$. Therefore, at loop termination, we have the post-conditions as: $I \wedge \neg B$, with:

$$\begin{aligned} & I \wedge \neg B \\ & \equiv I_1 \wedge (i \geq 1) \wedge (i \leq 1) \\ & \equiv I_1 \wedge (i = 1) \\ & \equiv \left(y = \tfrac{n!}{i!}\right) \wedge (i = 1) \\ & \equiv (y = n!). \end{aligned}$$

which is exactly our post condition for the entire program segment.

Now we need to show that I is indeed the invariant for the loop. First, by applying Axiom A3 to line 6, we get:

$$\begin{aligned} & \{(y = \tfrac{n!}{(i-1)!}) \wedge (i - 1 \geq 1)\} \\ & i \leftarrow i - 1; \\ & \{(y = \tfrac{n!}{i!}) \wedge (i \geq 1)\}. \end{aligned}$$

And, again applying Axiom A3 to line 5, we get:

$$\begin{aligned} & \{\left(y \times i = \tfrac{n!}{(i-1)!}\right) \wedge (i - 1 \geq 1)\} \\ & y \leftarrow y \times i; \\ & \{\left(y = \tfrac{n!}{(i-1)!}\right) \wedge (i - 1 \geq 1)\}. \end{aligned}$$

The precondition to line 5 can be rewritten as:

$$\left(y \times i = \frac{n!}{(i-1)!}\right) \wedge (i - 1 \geq 1) \equiv \left(y = \frac{n!}{i!}\right) \wedge (i \geq 2)$$

Because for integer i, $(i \geq 2) \equiv (i > 1)$, we have:

$$\left(y = \frac{n!}{i!}\right) \wedge (i \geq 2) \equiv \left(y = \frac{n!}{i!}\right) \wedge (i > 1)$$

We can then establish the equivalence between this precondition with $I \wedge B$, as follows:

$$I \wedge B \equiv \left(y = \frac{n!}{i!}\right) \wedge (i \geq 1) \wedge (i > 1) \equiv \left(y = \frac{n!}{i!}\right) \wedge (i > 1)$$

Therefore, the verified pre-condition to line 5 is $I \wedge B$.

Combining the above for line 5 and line 6 using Axiom A4, and letting

$$P_i \equiv (y = \frac{n!}{(i-1)!}) \wedge (i - 1 \geq 1)$$

we get:

$$\frac{\{I \wedge B\} y \leftarrow y \times i; \{P_i\},\ \{P_i\} i \leftarrow i - 1; \{I\}}{\{I \wedge B\}\ y \leftarrow y \times i;\ i \leftarrow i - 1;\ \{I\}}.$$

Now, when we apply Axiom A7, we get:

$$\frac{\{I \wedge B\}\ y \leftarrow y \times i;\ i \leftarrow i - 1;\ \{I\}}{\{I\}\ \texttt{while}\ B\ \texttt{do begin}\ y \leftarrow y \times i;\ i \leftarrow i - 1;\ \texttt{end}\ \{I \wedge \neg B\}}.$$

The last couple of steps for the statements before the "`while`" loop can then be verified. For line 2, using Axiom A3 with post-condition I, we get:

$$\left\{\left(y = \frac{n!}{n!}\right) \wedge (n \geq 1)\right\}\ i \leftarrow n;\ \left\{\left(y = \frac{n!}{i!}\right) \wedge (i \geq 1)\right\}.$$

The pre-condition to line 2 can be reduced to $(y = 1) \wedge (n \geq 1)$. Again, applying Axiom A3 to line 1 yields:

$$\{(1 = 1) \wedge (n \geq 1)\}\ y \leftarrow 1;\ \{(y = 1) \wedge (n \geq 1)\}.$$

The pre-condition to line 1 is exactly the same to the pre-condition of our program segment, $n \geq 1$. Now, combining line 1 and line 2 using Axiom A3, we get:

$$\frac{\{n \geq 1\}\ y \leftarrow 1;\ \{(y = 1) \wedge (n \geq 1)\},\ \{(y = 1) \wedge (n \geq 1)\}\ i \leftarrow 1;\ \{I\}}{\{n \geq 1\}\ \text{line } 1 - 2\ \{I\}}.$$

Finally, combine lines 1–2 with the "`while`" loop, again using Axiom A3, we get:

$$\frac{\{(n \geq 1)\}\ \text{line } 1 - 2\ \{I\},\ \{I\}\ \texttt{while} - \text{loop}\ \{y = n!\}}{\{(n \geq 1)\}\ \text{whole program} - \text{segment in Figure 15.1}\ \{y = n!\}}.$$

This finishes our verification process or the correctness proof for the program-segment in Figure 15.1.

15.3 OTHER APPROACHES

Besides the axiomatic approach described above, two other widely used formal verification approaches are the weakest pre-condition approach and the functional approach. We next introduce the basic ideas of these approaches, then discuss some limitations of all these three approaches, and introduce the idea of model checking and other formal or semi-formal approaches that attempt to provide only a partial verification of certain properties.