Software Testing

Gregory M. Kapfhammer
Department of Computer Science
Allegheny College
`gkapfham@allegheny.edu`

I shall not deny that the construction of these testing programs has been a major intellectual effort: to convince oneself that one has not overlooked "a relevant state" and to convince oneself that the testing programs generate them all is no simple matter. The encouraging thing is that (as far as we know!) it could be done.

Edsger W. Dijkstra [Dijkstra, 1968]

1 Introduction

When a program is implemented to provide a concrete representation of an algorithm, the developers of this program are naturally concerned with the correctness and performance of the implementation. Software engineers must ensure that their software systems achieve an appropriate level of quality. **Software verification** is the process of ensuring that a program meets its intended specification [Kaner et al., 1993]. One technique that can assist during the specification, design, and implementation of a software system is software verification through **correctness proof**. **Software testing**, or the process of assessing the functionality and correctness of a program through execution or analysis, is another alternative for verifying a software system. As noted by Bowen, Hinchley, and Geller, software testing can be appropriately used in conjunction with correctness proofs and other types of formal approaches in order to develop high quality software systems [Bowen and Hinchley, 1995, Geller, 1978]. Yet, it is also possible to use software testing techniques in isolation from program correctness proofs or other formal methods.

Software testing is not a "silver bullet" that can guarantee the production of high quality software systems. While a "correct" correctness proof demonstrates that a software system (which exactly meets its specification) will always operate in a given manner, software testing that is not fully exhaustive can only suggest the presence of flaws and cannot prove their absence. Moreover, Kaner et al. have noted that it is impossible to completely test an application because [Kaner et al., 1993]: (1) the domain of program inputs is too large, (2) there are too many possible input paths, and (3) design and specification issues are difficult to test. The first and second points present obvious complications and the final point highlights the difficulty of determining if the specification of a problem solution and the design of its implementation are also correct.

Using a thought experiment developed by Beizer, we can explore the first assertion by assuming that we have a method that takes a `String` of ten characters as input and performs some arbitrary operation on the `String`. In order to test this function exhaustively, we would have to input 2^{80} `String`s and determine if they produce the appropriate output.[1] The testing of our hypothetical method might also involved the usage of anomalous input, like `String`s consisting of more or less than ten characters, to determine the robustness of the operation. In this situation, the total number of inputs would be significantly greater than 2^{80}. Therefore, we can conclude that exhaustive testing is an intractable problem since it is impossible to solve with a polynomial-time algorithm [Binder, 1999, Neapolitan and Naimipour, 1998]. The difficulties alluded to by the second assertion are exacerbated by the fact that certain execution paths in a program could be infeasible. Finally, software testing is an algorithmically unsolvable problem since there may be input values for which the program does not halt [Beizer, 1990, Binder, 1999].

[1]Suppose that our `String` is encoded in extended ASCII. There are $256 = 2^8$ different extended ASCII characters characters. Since we are dealing with a `String` of ten characters, there are 2^{80} possible unique `String`s.

Thus far, we have provided an intuitive understanding of the limitations of software testing. However, Morell has proposed a theoretical model of the testing process that facilitates the proof of pessimistic theorems that clearly state the limitations of testing [Morell, 1990]. Furthermore, Hamlet and Morell have formally stated the goals of a software testing methodology and implicitly provided an understanding of the limitations of testing [Hamlet, 1994, Morell, 1990]. Young and Taylor have also observed that every software testing technique must involve some tradeoff between accuracy and computational cost because the presence (or lack thereof) of defects within a program is an undecidable property [Young and Taylor, 1989]. The theoretical limitations of testing clearly indicate that it is impossible to propose and implement a software testing methodology that is completely accurate and applicable to arbitrary programs [Young and Taylor, 1989].

While software testing is certainly faced with inherent limitations, there are also a number of practical considerations that can hinder the application of a testing technique. For example, some programming languages might not readily support a selected testing approach, a test automation framework might not easily facilitate the automatic execution of certain types of test suites, or there could be a lack of tool support to test with respect to a specific test adequacy criterion. Even though any testing effort will be faced with significant essential and accidental limitations, the rigorous, consistent, and intelligent application of appropriate software testing techniques can improve the quality of the application under development.

2 Underlying Principles

2.1 Terminology

The IEEE standard defines a **failure** as the external, incorrect behavior of a program [IEEE, 1996]. Traditionally, the anomalous behavior of a program is observed when incorrect output is produced or a runtime failure occurs. Furthermore, the IEEE standard defines a **fault** as a collection of program source code statements that causes a failure. Finally, an **error** is a mistake made by a programmer during the implementation of a software system [IEEE, 1996].[2] The purpose of software testing is to reveal software faults in order to ensure that they do not manifest themselves as runtime failures during program usage. Throughout this chapter, we will use P to denote the program under test and F to represent the specification that describes the behavior of P.[3] We say that a program's execution is **correct** when its behavior matches the functional and non-functional requirements specified in F [Sommerville, 2000]. Normally, the program under test will operate in an environment that might include Java virtual machines, device drivers, operating systems, databases, and a host of other environmental factors [Whittaker, 2000, Whittaker and Voas, 2000]. The majority of the testing theory and the practical testing techniques discussed in this chapter disregard the environment of a software system and simply focus on the source code of P. However, some of the new approaches to software testing that are described in Section 3.9 do address the testing of an application and its interaction with its environment.

Building upon the definitions used in [Kapfhammer and Soffa, 2003, Memon, 2001], Definition 1 states our understanding of a test suite T that can be used to assess the quality of an application under test. We use Δ_f to denote the externally visible state of the application under test. Informally, Δ_f can be viewed as a set of pairs where the first value of each pair is a variable name and the second value of each pair is a value for the variable name. Equation (1) formally defines Δ_f, the externally visible state after the execution of T_f. In this equation, we use $var_\triangle$ and $val_\triangle$ to denote a variable name and a variable value in an external test state, respectively. Furthermore, we use $U_\triangle$ and $V_\triangle$ to respectively denote the universe of valid variable names and variable values for externally visible test states. Finally, we require $value(var_\triangle, f)$ to be a function that maps a variable name to the value for the variable name in a specified

[2]While these definitions are standard in the software engineering and software testing research community, they are different than those that are normally used in the fault-tolerant computing community. For example, this community defines a fault as the underlying phenomenon that causes an error. Furthermore, an error is recognized as a deviation in the system state from the correct state. For more details, please refer to [Jalote, 1998]

[3]Throughout the remainder of this chapter, we will use the terms "program", "application", and "system" in an interchangeable fashion.

Δ_f. An external test state Δ_f would contain the global variable values within the program under test, and any variable values that are made accessible by live object instances.

Definition 1. A *test suite* T is a triple $\langle \Delta_0, \langle T_1, \ldots, T_e \rangle, \langle \Delta_1, \ldots, \Delta_e \rangle \rangle$, consisting of an initial external test state, Δ_0, a test case sequence $\langle T_1, \ldots, T_e \rangle$ for state Δ_0, and expected external test states $\langle \Delta_1, \ldots, \Delta_e \rangle$ where $\Delta_f = T_f(\Delta_{f-1})$ for $f = 1, \ldots, e$. $\square$

$$\Delta_f = \{(\mathit{var}_\Delta, \mathit{val}_\Delta) \in U_\Delta \times V_\Delta \mid \mathit{value}(\mathit{var}_\Delta, f) = \mathit{val}_\Delta\} \quad (1)$$

Definition 2 notes that a specific test $T_f \in \langle T_1, \ldots, T_e \rangle$ can be viewed as a sequence of test operations that cause the application under test to enter into states that are only visible to T_f. We used δ_h to denote the internal test state that is created after the execution of T_f's test case operation o_h. Intuitively, δ_h can also be viewed as a set of pairs where the first value is a variable name and the second value is a value for the variable name. Equation (2) formally defines δ_h in similar fashion to the definition of Δ_f in Equation (1). An internal test state δ_h would contain the expected and actual values for the test operation o_h, the return value from the program method under test, and the values of any temporary testing variables. Section 3.2 provides several examples of internal test states.

Definition 2. A *test case* $T_f \in \langle T_1, \ldots, T_e \rangle$, is a triple $\langle \delta_0, \langle o_1, \ldots, o_g \rangle, \langle \delta_1, \ldots, \delta_g \rangle \rangle$, consisting of an initial internal test state, δ_0, a test operation sequence $\langle o_1, \ldots, o_g \rangle$ for state δ_0, and expected internal test states $\langle \delta_1, \ldots, \delta_g \rangle$ where $\delta_h = o_h(\delta_{h-1})$ for $h = 1, \ldots, g$. $\square$

$$\delta_h = \{(\mathit{var}_\delta, \mathit{val}_\delta) \in U_\delta \times V_\delta \mid \mathit{value}(\mathit{var}_\delta, h) = \mathit{val}_\delta\} \quad (2)$$

In Definition 3, we describe a restricted type of test suite where each test case returns the application under test back to the initial state, Δ_0, before it terminates [Pettichord, 1999]. If a test suite T is not independent, we do not place any restrictions upon the $\langle \Delta_1, \ldots, \Delta_e \rangle$ produced by the test cases and we simply refer to it as a non-restricted test suite. Our discussion of test execution in Section 3.6 will reveal that the JUnit test automation framework facilitates the creation of test suites that adhere to Definition 1 and Definition 2 and are either independent or non-restricted in nature (although, JUnit encourages the creation of independent test suites) [Gamma and Beck, 2004, Hightower, 2001, Jackson, 2003, Jeffries, 1999].

Definition 3. A test suite T is *independent* if and only if for all $\gamma \in \{1, \ldots e\}$, $\Delta_\gamma = \Delta_0$. $\square$

Figure 1 provides a useful hierarchical decomposition of different testing techniques and their relationship to different classes of test adequacy criteria. While our hierarchy generally follows the definitions provided by Binder and Zhu et. al, it is important to note that other decompositions of the testing process are possible [Binder, 1999, Zhu et al., 1997]. This chapter focuses on **execution-based software testing** techniques. However, it is also possible to perform **non-execution-based software testing** through the usage of **software inspections** [Fagan, 1976]. During a software inspection, software engineers examine the source code of a system and any documentation that accompanies the system. A software inspector can be guided by a **software inspection checklist** that highlights some of the important questions that should be asked about the artifact under examination [Brykczynski, 1999]. While an inspection checklist is more sophisticated than an ad-hoc software inspection technique, it does not dictate how an inspector should locate the required information in the artifacts of a software system. **Scenario-based reading** techniques, such as Perspective-Based Reading (PBR), enable a more focused review of software artifacts by requiring inspectors to assume the perspective of different classes of program users [Laitenberger and Atkinson, 1999, Shull et al., 2001].

Since the selected understanding of adequacy is central to any testing effort, the types of tests within T will naturally vary based upon the chosen adequacy criterion C. As shown in Figure 1, all execution-based testing techniques are either **program-based**, **specification-based**, or **combined** [Zhu et al., 1997]. A program-based testing approach relies upon the structure and attributes of P's source code to create T. A specification-based testing technique simply uses F's statements about the functional and/or non-functional requirements for P to create the desired test suite. A combined testing technique

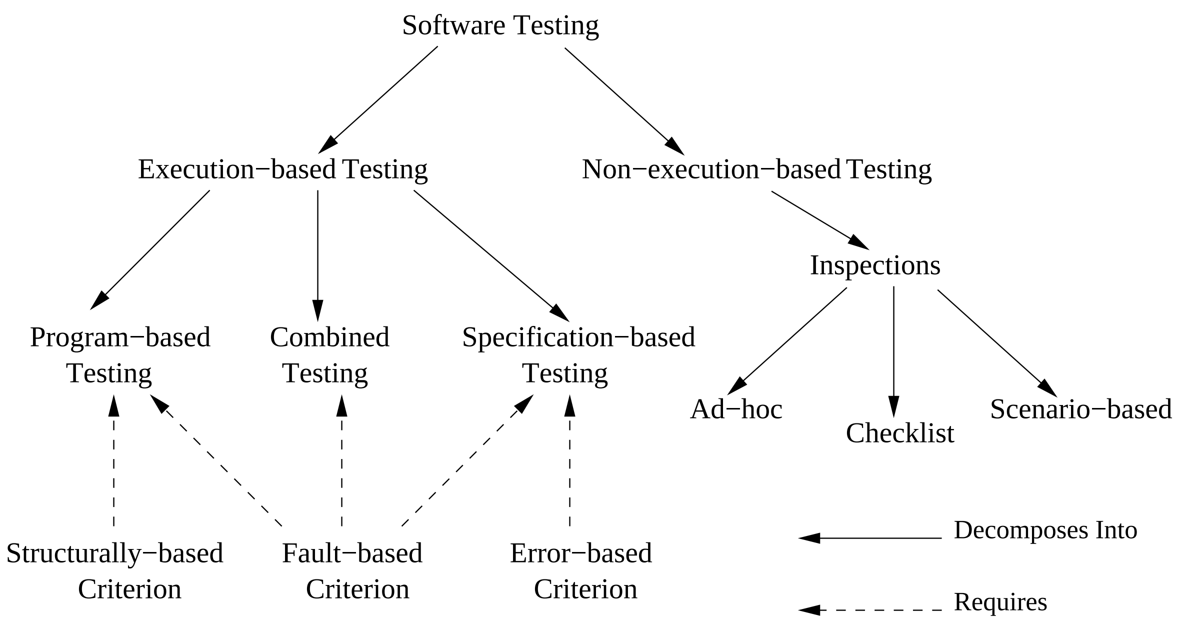


Figure 1: Hierarchy of Software Testing Techniques.

creates a test suite T that is influenced by both program-based and specification-based testing approaches [Zhu et al., 1997]. Moreover, the tests in T can be classified based upon whether they are **white-box**, **black-box**, or **grey-box** test cases. Specification-based test cases are black-box tests that were created without knowledge of P's source code. White-box (or, alternatively, **glass-box**) test cases consider the entire source code of P, while so called grey-box tests only consider a portion of P's source code. Both white-box and grey-box approaches to testing would be considered program-based or combined techniques.

A complementary decomposition of the notion of software testing is useful to highlight the centrality of the chosen test adequacy criterion. The tests within T can be viewed based upon whether they are "good" with respect to a **structurally-based**, **fault-based**, or **error-based** adequacy criterion [Zhu et al., 1997]. A structurally-based criterion requires the creation of a test suite T that solely requires the exercising of certain control structures and variables within P. Thus, it is clear that structurally-based test adequacy criterion require program-based testing. Fault-based test adequacy criterion attempt to ensure that P does not contain the types of faults that are commonly introduced into software systems by programmers [DeMillo et al., 1978, Morell, 1990, Zhu et al., 1997]. Traditionally, fault-based criterion are associated with program-based testing approaches. However, Richardson et al. have also described fault-based testing techniques that attempt to reveal faults in F or faults in P that are associated with misunderstandings of F [Richardson et al., 1989]. Therefore, a fault-based adequacy criterion C can require either program-based, specification-based, or combined testing techniques. Finally, error-based testing approaches rely upon a C that requires T to demonstrate that P does not deviate from F in any typical fashion. Thus, error-based adequacy criteria necessitate specification-based testing approaches.

2.2 Model of Execution-based Software Testing

Figure 2 provides a model of execution-based software testing. Since there are different understandings of the process of testing software, it is important to note that our model is only one valid and useful view of software testing. Using the notation established in Section 2.1, this model of software testing takes a system under test, P, and a test adequacy criterion, C, as input. This view of the software testing process is iterative in nature. That is, the initial creation and execution of T against P can be followed by multiple refinements of T and subsequent re-testings of P. Ideally, the testing process will stop iterating when

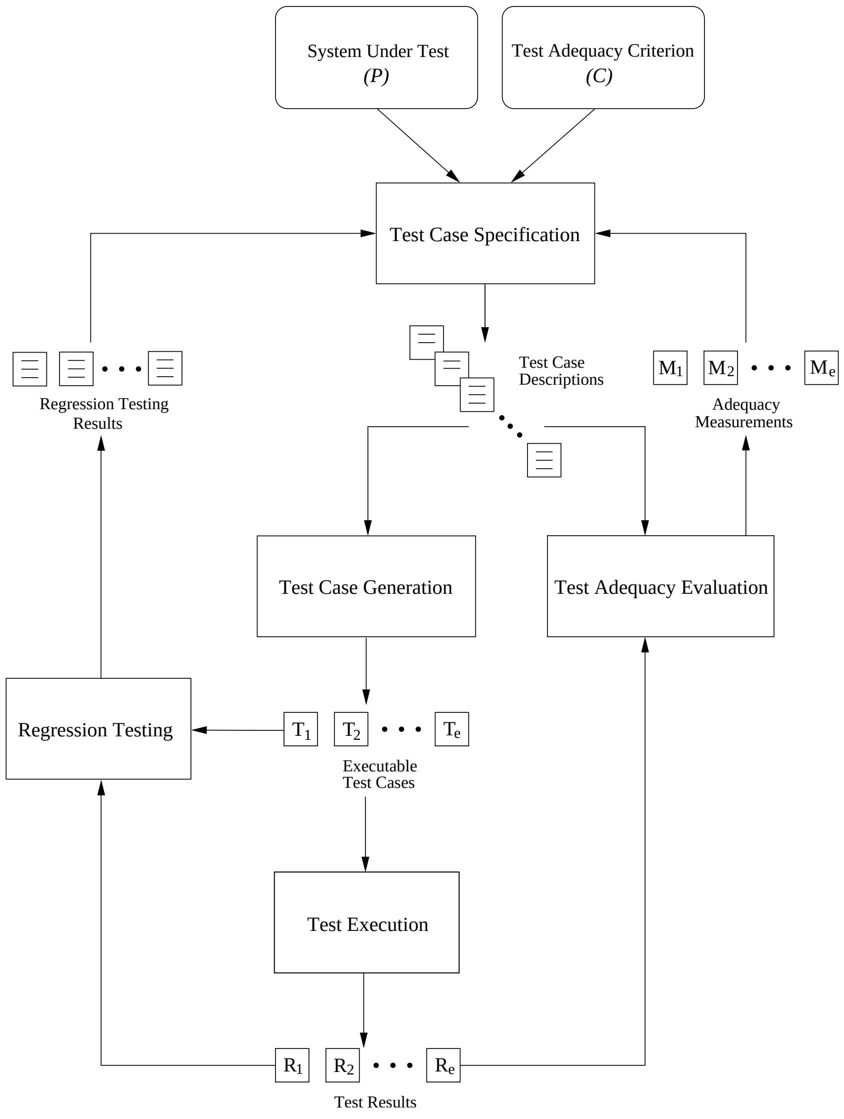


Figure 2: A Model of the Software Testing Process.

the tests within test suite T have met the adequacy criterion C and the testing effort has established the desired level of confidence in the quality of P [Zhu et al., 1997]. In practice, testing often stops when a release deadline is reached, monetary resources are exhausted, or the developers have an intuitive sense that P has reached an acceptable level of quality. Yet, it is important to note that even if the testing effort stops when T meets C, there is no guarantee that T has isolated all of the defects within P. Since C traditionally represents a single view of test case quality, it is often important to use multiple adequacy criterion and resist the temptation to allow the selected C(s) to exclusively guide the testing effort [Marick, 1998, 1999].[4] The formulation of a test adequacy criterion is a function of a chosen representation of P and a specific understanding of the "good" qualities that T should represent. In Section 3.3 we review the building blocks of programs that are commonly used to formulate test adequacy criterion. Furthermore, Section 3.4 reviews some test adequacy metrics that have been commonly discussed in the literature and/or frequently used in practice. The **test specification** stage analyzes a specific P in light of a chosen C in order to construct a listing of the tests that must be provided to create a completely adequate test suite.

[4]Even though multiple adequacy criteria are normally used during a testing effort, we will continue to use a single adequacy criterion C in order to simplify our notation.

```
 1  import java.lang.Math;
 2  public class Kinetic
 3  {
 4    public static String computeVelocity(int kinetic, int mass)
 5    {
 6      int velocity_squared, velocity;
 7      StringBuffer final_velocity = new StringBuffer();
 8      if( mass != 0 )
 9      {
10        velocity_squared = 3 * (kinetic / mass);
11        velocity = (int)Math.sqrt(velocity_squared);
12        final_velocity.append(velocity);
13      }
14      else
15      {
16        final_velocity.append("Undefined");
17      }
18      return final_velocity.toString();
19    }
20  }
```

Figure 3: The `Kinetic` class that Contains a Fault in `computeVelocity`.

That is, the test case specification stage shown in Figure 2 is responsible for making C directly apply to the system under test. Once a test case specification tool has created test case descriptions for P, the **test case generation** phase can begin. Section 3.5 examines different techniques that can be used to manually or automatically generate test cases.

After the test cases have been generated, it is possible to perform **test execution**. Once again, the execution of the tests within T can be performed in a manual or automated fashion. Also, the results from the execution of the tests can be analyzed in either an automated or manual fashion to determine if each individual test case passed or failed. The executable test cases that were constructed during the generation phase can be analyzed by a **test adequacy evaluator** that measures the quality of T with respect to the test case descriptions produced by the test specifier. The process of test execution is detailed in Section 3.6 and the test adequacy evaluation phase is described in Section 3.7. Of course, the test results from the execution phase and the adequacy measurements produced by the evaluator can be used to change the chosen adequacy criteria and/or augment the listing of test case descriptions that will be used during subsequent testing.

The iterative process of testing can continue throughout the initial development of P. However, it is also important to continue the testing of P after the software application has been released and it enters the **maintenance phase** of the **software lifecycle** [Sommerville, 2000]. **Regression testing** is an important software maintenance activity that attempts to ensure that the addition of new functionality and/or the removal of program faults does not negatively impact the correctness of P. The regression testing process can rely upon the existing test cases and the adequacy measurements for these tests to iteratively continue all of the previously mentioned stages [Onoma et al., 1998].

3 Best Practices

3.1 An Example Program

In an attempt to make our discussion of software testing techniques more concrete, Figure 3 provides a Java class called `Kinetic` that contains a `static` method called `computeVelocity` [Paul, 1996]. The `computeVelocity` operation is supposed to calculate the velocity of an object based upon its kinetic energy and its mass. Since the kinetic energy of an object, K, is defined as $K = \frac{1}{2}mv^2$, it is clear that `computeVelocity` contains a defect on line 10. That is, line 10 should be implemented with the assignment statement `velocity_squared = 2 * (kinetic / mass)`.

```
import junit.framework.*;
public class KineticTest extends TestCase
{
  public KineticTest(String name)
  {
    super(name);
  }
  public static Test suite()
  {
    return new TestSuite(KineticTest.class);
  }
  public void testOne()
  {
    String expected = new String(”Undefined”);
    String actual = Kinetic.computeVelocity(5,0);
    assertEquals(expected, actual);
  }
  public void testTwo()
  {
    String expected = new String(”0”);
    String actual = Kinetic.computeVelocity(0,5);
    assertEquals(expected, actual);
  }
  public void testThree()
  {
    String expected = new String(”4”);
    String actual = Kinetic.computeVelocity(8,1);
    assertEquals(expected, actual);
  }
  public void testFour()
  {
    String expected = new String(”20”);
    String actual = Kinetic.computeVelocity(1000,5);
    assertEquals(expected, actual);
  }
}
```

Figure 4: A JUnit Test Case for the Faulty `Kinetic` Class.

3.2 Fault/Failure Model

In Section 1, we informally argued that software testing is difficult. DeMillo et al., Morell, and Voas have separately proposed very similar fault/failure models that describes the conditions under which a fault will manifest itself as a failure [DeMillo and Offutt, 1991, Morell, 1990, Voas, 1992]. Using the fault/failure model proposed by Voas and the `Kinetic` example initially created by Paul, we can define a simple test suite to provide anecdotal evidence of some of the difficulties that are commonly associated with writing a test case that reveals a program fault [Paul, 1996]. As stated in the **PIE model** proposed by Voas, a fault will only manifest itself in a failure if a test case T_f executes the fault, causes the fault to infect the data state of the program, and finally, propagates to the output [Voas, 1992]. That is, the necessary and sufficient conditions for the isolation of a fault in P are the execution, infection, and propagation of the fault [DeMillo and Offutt, 1991, Morell, 1990, Voas, 1992].

Figure 4 provides the source code for `KineticTest`, a Java class that adheres to the JUnit test automation framework [Gamma and Beck, 2004, Hightower, 2001, Jackson, 2003, Jeffries, 1999]. Using our established notation, we have a T that contains a test case sequence $\langle T_1, T_2, T_3, T_4 \rangle$ with each T_f containing a single testing operation o_1. For example, T_1 contains the testing operation `String actual = Kinetic.computeVelocity(5,0)`. It is important to distinguish between the external test data states and the internal test data states that occur during the execution of a test case from the internal data states within the method under test [Voas, 1992]. To this end, we require λ_b to correspond to the internal data state after the execution of line b in the method under test. Following the established terminology, λ_b can be informally viewed as a set of pairs where the first value of each pair is a variable name and the second value of each pair is a value for the variable name. Equation (3) formally defines a method data state λ_b in a fashion that is similar to Equation (1) and Equation (2) in Section 2.1. A method data state λ_b would include the value of local variables in the method under test and the current value of variables that are members of the external test state. Finally, we use λ_b^E to denote the expected data state that would normally result from the execution of a non-faulty version of line b.

$$\lambda_b = \{(\mathit{var}_\lambda, \mathit{val}_\lambda) \in U_\lambda \times V_\lambda \mid \mathit{value}(\mathit{var}_\lambda, b) = \mathit{val}_\lambda\} \tag{3}$$

Since our illustration describes δ_b for each T_f in T's test case sequence, we use $\delta_h(f)$ to denote the hth internal test state induced by the execution of test T_f. We also use $\lambda_b(f)$ and $\lambda_b^E(f)$ to correspond to the actual and expected method states created by test T_f after the execution of line b in the method under test. Equation (4) describes the test state before the execution of o_1 in T_1. Equation (5) provides the state of the `Kinetic` class after the faulty `computeVelocity` method has been executed.[5] It is important to note that this test case causes the `computeVelocity` method to produce the data state $\delta_1(1)$ where the actual data value corresponds to the expected data value. In this example, we are also interested in the internal states $\lambda_{10}(1)$ and $\lambda_{10}^E(1)$, which correspond to the actual and expected data states after the execution of line 10. However, since T_1 does not execute the defect on line 10, the test does not produce the internal method data states that could result in the isolation of the fault.

$$\delta_0(1) = \{(A, null), (E, \mathit{Undefined}), (K, 5), (m, 0)\} \tag{4}$$
$$\delta_1(1) = \{(A, \mathit{Undefined}), (E, \mathit{Undefined}), (K, 5), (m, 0)\} \tag{5}$$

The execution of T_2 corresponds to the initial and final internal test states as described by Equation (6) and Equation (7), respectively. In this situation, it is clear that the test case produces a final internal test state $\delta_1(2)$ where A correctly matches E. Equation (8) and Equation (9) show the actual and expected internal method states that result after the execution of the faulty line in the method under test. While the execution of this test case does cause the fault to be executed, the faulty statement does not infect the data state (i.e. $\lambda_{10}(2)$ and $\lambda_{10}^E(2)$ are equivalent). Due to the lack of infection, it is impossible for T_2 to detect the fault in the `computeVelocity` method.

$$\delta_0(2) = \{(A, null), (E, 0), (K, 0), (m, 5)\} \tag{6}$$
$$\delta_1(2) = \{(A, 0), (E, 0), (K, 0), (m, 5)\} \tag{7}$$

$$\lambda_{10}(2) = \{(K, 0), (m, 5), (v^2, 0), (v, 0)(v_f, 0)\} \tag{8}$$
$$\lambda_{10}^E(2) = \{(K, 0), (m, 5), (v^2, 0), (v, 0)(v_f, 0)\} \tag{9}$$

Equation (10) and Equation (11) correspond to the initial and final test data states when T_3 is executed. However, state $\delta_1(3)$ still contains an A that correctly corresponds to E. In this situation, the test case does execute the fault on line 10 of `computeVelocity`. Since Equation (12) and Equation (13) make it clear that the method data states $\lambda_{10}(3)$ and $\lambda_{10}^E(3)$ are different, we know that the fault has infected the method data state. However, the cast to an `int` on line 11 creates a **coincidental correctness** that prohibits the fault from manifesting itself as a failure [Voas, 1992]. Due to the lack of propagation, this test case has not isolated the fault within the `computeVelocity` method.

$$\delta_0(3) = \{(A, null), (E, 4), (K, 8), (m, 1)\} \tag{10}$$
$$\delta_1(3) = \{(A, 4), (E, 4), (K, 8), (m, 1)\} \tag{11}$$

$$\lambda_{10}(3) = \{(K, 8), (m, 1), (v^2, 24), (v, 0), (v_f, 0)\} \tag{12}$$
$$\lambda_{10}^E(3) = \{(K, 8), (m, 1), (v^2, 16), (v, 0), (v_f, 0)\} \tag{13}$$

[5]For the sake of brevity, our descriptions of the internal test and method data states use the variables K, m, v^2, v, v_f, A, and E to mean the program variables `kinetic`, `mass`, `velocity_squared`, `velocity`, `final_velocity`, `actual`, and `expected`, respectively.

Test case T_4 produces the initial and final internal test states that are described in Equation (14) and Equation (15). Since the actual data value in A is different from the expected data value, the test is able to reveal the fault in the `computeVelocity` method. This test case executes the fault and causes the fault to infect the data state since the $\lambda_{10}(4)$ and $\lambda_{10}^{E}(4)$ provided by Equation (16) and Equation (17) are different. Finally, the method data state $\lambda_{10}(4)$ results in the creation of the internal test state $\delta_1(4)$. Due to the execution of line 10, the infection of the method data state $\lambda_{10}(4)$, and the propagation to the output, this test case is able to reveal the defect in `computeVelocity`. The execution of the `KineticTest` class in the JUnit test automation framework described in Section 3.6 will confirm that T_4 will reveal the defect in `Kinetic`'s `computeVelocity` method.

$$\delta_0(4) = \{(A, null), (E, 20), (K, 1000), (m, 5)\} \tag{14}$$
$$\delta_1(4) = \{(A, 24), (E, 20), (K, 1000), (m, 5)\} \tag{15}$$

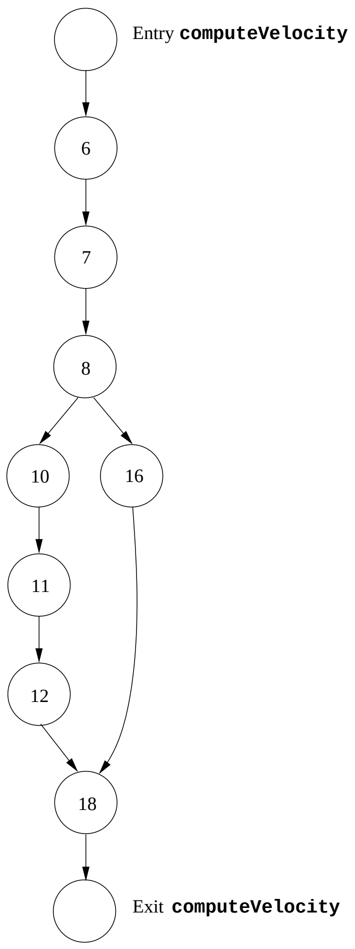
$$\lambda_{10}(4) = \{(K, 1000), (m, 5), (v^2, 600), (v, 0), (v_f, 0)\} \tag{16}$$
$$\lambda_{10}^{E}(4) = \{(K, 1000), (m, 5), (v^2, 400), (v, 0), (v_f, 0)\} \tag{17}$$

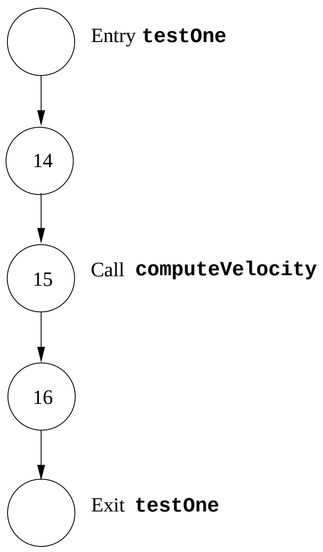
3.3 Program Building Blocks

As noted in Section 2.2, a test adequacy criterion is dependent upon the chosen representation of the system under test. We represent the program P as an **interprocedural control flow graph** (ICFG). An ICFG is a collection of control flow graphs (CFGs) $G_1, G_2, \ldots, G_u$ that correspond to the CFGs for P's methods $m_1, m_2, \ldots, m_u$, respectively. We define control flow graph G_v so that $G_v = (N_v, E_v)$ and we use N_v to denote a set of CFG nodes and E_v to denote a set of CFG edges. Furthermore, we assume that each $n \in N_v$ represents a statement in method m_v and each $e \in E_v$ represents a transfer of control in method m_v. Also, we require each CFG G_v to contain unique nodes $entry_v$ and $exit_v$ that demarcate the entrance and exit points of method m_v, respectively. We use the sets $pred(n_\tau) = \{n_\rho | (n_\rho, n_\tau) \in E_v\}$ and $succ(n_\rho) = \{n_\tau | (n_\rho, n_\tau) \in E_v\}$ to denote the set of predecessors and successors of node n_τ and n_ρ, respectively. Finally, we require $N = \cup\{N_v | v \in [1, u]\}$ and $E = \cup\{E_v | v \in [1, u]\}$ to contain all of the nodes and edges in the interprocedural control flow graph for program P.

Figure 5 provides the control flow graphs G_{cv} and G_{T_1} for the `computeVelocity` method and the `testOne` method that can be used to test `computeVelocity`. Each of the nodes in these CFGs are labeled with line numbers that correspond to the numbers used in the code segments in Figure 3 and Figure 4. Each of these control flow graphs contain unique entry and exit nodes and G_{T_1} contains a node n_{15} labelled "Call `computeVelocity`" to indicate that there is a transfer of control from G_{T_1} to G_{cv}. Control flow graphs for the other methods in the `KineticTest` class would have the same structure as the CFG for `testOne`. Even though these control flow graphs do not contain iteration constructs, it is also possible to produce CFGs for programs that use `for`, `while`, and `do while` statements. Control flow graphs for programs that contain significantly more complicated conditional logic blocks with multiple, potentially nested, `if`, `else if`, `else`, or `switch` statements can also be created.

When certain test adequacy criterion and the testing techniques associated with the criterion require an inordinate time and space overhead to compute the necessary test information, an **intraprocedural control flow graph** for a single method can be used. Of course, there are many different graph-based representations for programs. Harrold and Rothermel survey a number of graph-based representations [Harrold and Rothermel, 1996] and the algorithms and tool support used to construct these representations [Harrold and Rothermel, 1995]. For example, the class control flow graph (CCFG) represents the static control flow between the methods within a specific class [Harrold and Rothermel, 1996, 1994]. This graph-based representation supports the creation of class-centric test adequacy metrics that only require a limited interprocedural analysis. The chosen representation for the program under test influences the measurement of the quality of existing test suites and the generation of new tests. While definitions in Section 3.4 are written in the context of a specific graph-based representation of a program, these definitions are still applicable when different program representations are chosen. Finally, these graph-based representations can be created with a program analysis framework like Aristotle [Harrold and Rothermel, 1995] or Soot [Vallée-Rai et al., 1999].





(a) (b)

Figure 5: The Control Flow Graphs for the `computeVelocity` and `testOne` methods.

3.4 Test Adequacy Metrics

As noted in Section 2.1, test adequacy metrics embody certain characteristics of test case "quality" or "goodness." Test adequacy metrics can be viewed in light of a program's control flow graph and the program paths and variable values that they require to be exercised. Intuitively, if a test adequacy criterion C_α requires the exercising of more path and variable value combinations than criterion C_β, it is "stronger" than C_β. More formally, a test adequacy criterion C_α **subsumes** a test adequacy criterion C_β if every test suite that satisfies C_α also satisfies C_β [Clarke et al., 1985, Rapps and Weyuker, 1985]. Two adequacy criteria C_α and C_β are equivalent if C_α subsumes C_β and vice versa. Finally, a test adequacy criterion C_α **strictly subsumes** criterion C_β if and only if C_α subsumes C_β and C_β does not subsume C_α [Clarke et al., 1985, Rapps and Weyuker, 1985].

3.4.1 Structurally-based

Some software test adequacy criteria are based upon the control flow graph of a program under test. Control flow-based criterion solely attempt to ensure that test suite T covers certain source code locations and values of variables. While several control flow-based adequacy criterion are relatively easy to satisfy, others are so strong that it is generally not possible for a T to test P and satisfy the criterion. Some control flow-based adequacy criteria focus on the control structure of a program and the value of the variables that are used in conditional logic predicates. Alternatively, data flow-based test adequacy criteria require coverage of the control flow graph by forcing the selection of program paths that involved the definition and/or usage of program variables.

Control Flow-Based Criterion

Our discussion of control flow-based adequacy criterion will use the notion of an arbitrary path through P's interprocedural control flow graph $G_1, \ldots, G_u$. We distinguish π as a **complete path** in an interprocedural control flow graph or other graph-based representation of a program module. A complete path is a path in a control flow graph that starts at the program graph's entry node and ends at its exit node [Frankl and Weyuker, 1988]. Unless otherwise stated, we will assume that all of the paths required by the test adequacy criterion are complete. For example, the interprocedural control flow graph described in Figure 5 contains the complete interprocedural path $\pi = \langle entry_{T_1}, n_{14}, n_{15}, entry_{cv}, n_6, n_7, n_8, n_{16}, n_{18}, exit_{cv}, n_{16}, exit_{T_1} \rangle$. Note that the first n_{16} in π corresponds to a node in the control flow graph for `computeVelocity` and the second n_{16} corresponds to a node in `testOne`'s control flow graph. Since the fault/failure model described in Section 3.2 indicates that it is impossible to reveal a fault in P unless the faulty node from P's CFG is included within a path that T produces, there is a clear need for a test adequacy criterion that requires the execution of all statements in a program. Definition 4 explains the **all-nodes** (or, alternatively, **statement coverage**) criterion for a test suite T and a program under test P.

Definition 4. A test suite T for control flow graph $G_v = (N_v, E_v)$ satisfies the all-nodes test adequacy criterion if and only if the tests in T create a set of complete paths Π_{N_v} that include all $n \in N_v$ at least once. $\square$

Intuitively, the all-nodes criterion is weak because it is possible for a test suite T to satisfy this criterion and still not exercise of all the transfers of control (i.e. the edges) within the control flow graph [Zhu et al., 1997]. For example, if a test suite T tests a program P that contains a single `while` loop, it can satisfy statement coverage by only executing the iteration construct once. However, a T that simply satisfies statement coverage will not execute the edge in the control flow graph that returns execution to the node that marks the beginning of the `while` loop. Thus, the **all-edges** (or, alternatively, **branch coverage**) criterion described in Definition 5 requires a test suite to exercise every edge within an interprocedural control flow graph.

Definition 5. A test suite T for control flow graph $G_v = (N_v, E_v)$ satisfies the all-edges test adequacy criterion if and only if the tests in T create a set of complete paths Π_{E_v} that include all $e \in E_v$ at least once. $\square$

Since the inclusion of every edge in a control flow graph implies the inclusion of every node within the same CFG, it is clear that the branch coverage criterion subsumes the statement coverage criterion [Clarke et al., 1985, Zhu et al., 1997]. However, it is still possible to cover all of the edges within an intraprocedural control flow graph and not cover all of the unique paths from the entry point to the exit point of P's CFG. For example, if a test suite T is testing a program P that contains a single `while` loop it can cover all of the edges in the control flow graph by executing the iteration construct twice. Yet, a simple program with one `while` loop contains an infinite number of unique paths since each iteration of the looping construct creates a new path. Definition 6 explores the **all-paths** test adequacy criterion that requires the execution of every path within the program under test.

Definition 6. A test suite T for control flow graph $G_v = (N_v, E_v)$ satisfies the all-paths test adequacy criterion if and only if the tests in T create a set of complete paths Π_v that include all the execution paths beginning at the unique entry node $entry_v$ and ending at the unique exit node $exit_v$. $\square$

The all-paths criterion subsumes both all-edges and all-nodes. However, it is important to note that it is possible for a test suite T to be unable to satisfy even all-nodes or all-edges if P contains infeasible paths. Yet, it is often significantly more difficult (or, impossible) for a test suite to satisfy all-paths while still being able to satisfy both all-edges and all-nodes. The distinct difference in the "strength" of all-paths and the "weaker" all-edges and all-nodes presents a need for alternative adequacy criteria to "stand in the gap" between these two criteria.

As noted in [Michael et al., 2001], there is also a hierarchy of test adequacy criteria that strengthen the all-edges criterion described in Definition 5. The **multiple condition** coverage criterion requires a test suite to account for every permutation of the boolean variables in every branch of the program's control flow graph, at least one time. For example, if `a` and `b` are boolean variables, then the conditional logic statement `if(a && b)`, requires a test suite T that covers the $2^2 = 4$ different assignments to these variables. The addition of a single boolean variable to a conditional logic statement doubles the number of assignments that a test suite must produce in order to fulfill multiple condition coverage.

Another test adequacy criterion that is related to the all-edges criterion is **condition-decision** coverage. We use the term **condition** to refer to an expression that evaluates to `true` or `false` while not having any other boolean valued expressions within it [Michael et al., 2001]. Intuitively, condition-decision coverage requires a test suite to cover each of the edges within the program's control flow graph and ensure that each condition in the program evaluates to `true` and `false` at least one time. For the example conditional logic statement `if(a && b)`, a test suite T could satisfy the condition-decision adequacy criterion with the assignments $(\mathtt{a} = 0, \mathtt{b} = 0)$ and $(\mathtt{a} = 1, \mathtt{b} = 1)$. It is interesting to note that these assignments also fulfill the all-edges adequacy criterion. Yet, $(\mathtt{a} = 1, \mathtt{b} = 1)$ and $(\mathtt{a} = 1, \mathtt{b} = 0)$ fulfill all-edges without meeting the condition-decision criterion. Moreover, for the conditional logic statement `if(alwaysFalse(a,b))` (where `alwaysFalse(boolean a, boolean b)` simply returns `false`), it is possible to fulfill condition-decision coverage with the assignments $(\mathtt{a} = 0, \mathtt{b} = 0)$ and $(\mathtt{a} = 1, \mathtt{b} = 1)$ and still not meet the all-edges criterion. These examples indicate that there is no clear subsumption relationship between all-edges and condition-decision. While multiple condition subsumes both of these criterion, it is clearly subsumed by the all-paths test adequacy criterion.

Data Flow-Based Criterion

Throughout our discussion of data flow-based test adequacy criteria, we will adhere to the notation initially proposed in [Frankl and Weyuker, 1988, Rapps and Weyuker, 1982, 1985]. For a standard program, the occurrence of a variable on the left hand side of an assignment statement is called a **definition** of this variable. Also, the occurrence of a variable on the right hand side of an assignment statement is called a **computation-use** (or **c-use**) of this variable. Finally, when a variable appears in the predicate of a conditional logic statement or an iteration construct, we call this a **predicate-use** (or **p-use**) of the variable.

As noted in Section 3.3, we will view a method in an application as a control flow graph $G_v = (N_v, E_v)$ where N_v is the set of CFG nodes and E_v is the set of CFG edges. For simplicity, our explanation of data flow-based test adequacy criteria will focus on data flow information within a single method's control flow graph of the entire ICFG for program P. However, our definitions can be extended to the interprocedural control flow graph. In practice, interprocedural data flow analysis often incurs significant time and space overhead and many testing techniques limit their adequacy metrics to only require intraprocedural or limited interprocedural analysis. Yet, there are interprocedural data flow analysis techniques such as the exhaustive algorithms proposed by Harrold and Soffa [Harrold and Soffa, 1994] and the demand-driven approach discussed by Duesterwald et al. [Duesterwald et al., 1996], that can be used to compute data flow-based test adequacy criteria for an ICFG.

We define a **definition clear path** for variable var_v as a path $\langle n_\rho, \ldots, n_\tau \rangle$ in G_v, such that none of the nodes $n_\rho, \ldots, n_\tau$ contain a definition or undefinition of program variable var_v [Frankl and Weyuker, 1988]. Furthermore, we define the **def-c-use** association as a triple $\langle n_d, n_{c-use}, \mathit{var}_v \rangle$ where a definition of variable var_v occurs in node n_d and a c-use of var_v occurs in node n_{c-use}. Also, we define the **def-p-use** association as the two triples $\langle n_d, (n_{p-use}, t), \mathit{var}_v \rangle$ and $\langle n_d, (n_{p-use}, f), \mathit{var}_v \rangle$ where a definition of variable var_v occurs in node n_d and a p-use of var_v occurs during the true and false evaluations of a predicate at node n_{p-use} [Frankl and Weyuker, 1988, Rapps and Weyuker, 1982, 1985]. A complete path π^{var_v} *covers* a def-c-use association if it has a definition clear sub-path, with respect to var_v and the method's CFG, that begins with node n_d and ends with node n_{c-use}. Similarly, π^{var_v} covers a def-p-use association if it has definition clear sub-paths, with respect to var_v and the program's CFG, that begin with node n_d and end with the true and false evaluations of the logical predicate contained at node n_{p-use} [Frankl and Weyuker, 1988].

In [Rapps and Weyuker, 1982, 1985], the authors propose a family of test adequacy measures based upon data flow information in a program. Among their test adequacy measures, the **all-uses** data flow adequacy criteria requires a test suite to cover all of the def-c-use and def-p-use associations in a program. The all-uses criterion is commonly used as the basis for **definition-use testing**. Alternatively, the **all-c-uses** criterion requires the coverage of all the c-use associations for a given method under test and the **all-p-uses** adequacy criterion requires the coverage of all the p-use associations. Furthermore, the **all-du-paths** coverage criterion requires the coverage of all the paths from the definition to a usage of a program variable [Rapps and Weyuker, 1982, 1985]. Definition 7 through Definition 10 define several important test adequacy criterion that rely upon data flow information. However, we omit a formal definition of certain data flow-based test adequacy criterion, such as **all-c-uses/some-p-uses**, **all-p-uses/some-c-uses**, and **all-DUs** [Hutchins et al., 1994, Rapps and Weyuker, 1982, 1985]. The all-c-uses/some-p-uses and the all-p-uses/some-c-uses test adequacy criteria are combinations of the all-c-uses and all-p-uses metrics described in Definition 7 and Definition 8. The all-DU test adequacy criterion is a modified version of all-uses that simply requires the coverage of def-use associations without distinguishing between a p-use and a c-use [Hutchins et al., 1994]. In each definition, we use U_v to refer to the universe of live program variable names for method m_v in program under test P.[6]

Definition 7. A test suite T for control flow graph $G_v = (N_v, E_v)$ satisfies the all-c-uses test adequacy criterion if and only if for each association $\langle n_d, n_{c-use}, \mathit{var}_v \rangle$, where $\mathit{var}_v \in U_v$ and $n_d, n_{c-use} \in N_v$, there exists a test $T_f \in \langle T_1, \ldots, T_e \rangle$ to create a complete path π^{var_v} in G_v that covers the association. $\Box$

Definition 8. A test suite T for control flow graph $G_v = (N_v, E_v)$ satisfies the all-p-uses test adequacy criterion if and only if for each association $\langle n_d, (n_{p-use}, t), \mathit{var}_v \rangle$ and $\langle n_d, (n_{p-use}, f), \mathit{var}_v \rangle$ where $\mathit{var}_v \in U_v$ and $n_d, n_{p-use} \in N_v$, there exists a test $T_f \in \langle T_1, \ldots, T_e \rangle$ to create a complete path π^{var_v} in G_v that covers the association. $\Box$

Definition 9. A test suite T for control flow graph $G_v = (N_v, E_v)$ satisfies the all-uses test adequacy criterion if and only if for each association $\langle n_d, n_{c-use}, \mathit{var}_v \rangle$, $\langle n_d, (n_{p-use}, t), \mathit{var}_v \rangle$ and $\langle n_d, (n_{p-use}, f), \mathit{var}_v \rangle$ where $\mathit{var}_v \in U_v$ and $n_d, n_{c-use}, n_{p-use} \in N_v$, there exists a test $T_f \in \langle T_1, \ldots, T_e \rangle$ to create a complete path π^{var_v} in G that covers the association. $\Box$

Definition 10. A test suite T for control flow graph $G_v = (N_v, E_v)$ satisfies the all-du-paths test adequacy criterion if and only if for each association $\langle n_d, n_{c-use}, \mathit{var}_v \rangle$, $\langle n_d, (n_{p-use}, t), \mathit{var}_v \rangle$ and $\langle n_d, (n_{p-use}, f), \mathit{var}_v \rangle$ where $\mathit{var}_v \in U_v$ and $n_d, n_{c-use}, n_{p-use} \in N_v$, the tests in T create a set of complete paths $\Pi_v^{\mathit{var}_v}$ that include all of the execution paths that cover the associations. $\Box$

Our discussion of subsumption for these test adequacy criteria is limited to the traditional understanding of data flow-based testing. A review of the **feasible data flow testing criteria**, a family of test adequacy criteria that only require the coverage of associations that are actually executable, and the subsumption hierarchy associated with these feasible criteria is provided in [Frankl and Weyuker, 1988]. Intuitively, the all-paths criterion subsumes all-du-paths because there might be complete paths within P that do not involve the definition and usage of program variables. Furthermore, all-du-paths subsumes all-uses because all-uses only requires one complete path to cover the required associations and all-du-paths requires all of the complete paths that cover the association. It is clear that all-uses subsumes both all-p-uses and all-c-uses and that there is no subsumption relationship between the all-p-uses and all-c-uses. Because a p-use requires the evaluation of the `true` and `false` branches of a conditional logic statement, it can be shown that all-p-uses subsumes both the all-edges and the all-nodes criterion [Frankl and Weyuker, 1988]. Since structurally-based test adequacy criteria that use control flow and data flow information are theoretically well-founded [Fenton, 1994, Nielson et al., 1999, Parrish and Zweben, 1991, Weyuker, 1986] and the criteria themselves can be organized into often meaningful subsumption hierarchies, it is clear that they will continue to be an important component of future research and practice.

[6]Using the terminology established in Section 3.2, we can observe that $U_v \subseteq U_\lambda$ since U_v only denotes the universe of live variables within m_v and U_λ is the universe of all valid variable names for method data states.

3.4.2 Fault-based

Mutation adequacy is the main fault-based test adequacy criterion. The conception of a mutation adequate test suite is based upon two assumptions called the **competent programmer hypothesis** and the **coupling effect** [DeMillo et al., 1988]. The competent programmer hypothesis assumes that competent programmers create programs that compile and very nearly meet their specification. The coupling effect assumption indicates that test suites that can reveal simple defects in a program under test can also reveal more complicated combinations of simple defects [DeMillo et al., 1988]. Therefore, fault-based test adequacy criterion attempt to ensure that a test suite can reveal all of the defects that are normally introduced into software systems by competent programmers.

Definition 11 describes a test suite T that is **adequate** and Definition 12 defines the notion of **relative adequacy** (or, alternatively, **mutation adequacy**) for test suites [DeMillo and Offutt, 1991]. In these definitions, we use the notation $Q(D)$ and $F(D)$ to mean the "output" of a program Q and the specification F on D, the entire domain of possible program inputs. Furthermore, we use the notation $Q(T_f)$ and $F(T_f)$ to denote the "output" of a program Q and a specification F on a single test T_f that provides a single input to Q and F. If a test suite T is adequate for a program under test P, then T is able to distinguish between all of the incorrect implementations of specification F. If a test suite T is relative-adequate (or, mutation adequate) for a program under test P, then T is able to distinguish between a finite set $\Phi_P = \{\phi_1, \ldots, \phi_s\}$ of incorrect implementations of specification F.

Definition 11. If P is a program to implement specification F on domain D, then a test suite $T \subset D$ is adequate for P and F if ($\forall$ programs Q), $[Q(D) \neq F(D)] \Rightarrow [(\exists T_f \in \langle T_1, \ldots, T_e \rangle)(Q(T_f) \neq F(T_f))]$. $\square$

Definition 12. If P is a program to implement specification F on domain D and Φ is a finite collection of programs, then a test suite $T \subset D$ is adequate for P relative to Φ if ($\forall$ programs $Q \in \Phi$), $[Q(D) \neq F(D)] \Rightarrow [(\exists T_f \in \langle T_1, \ldots, T_e \rangle)(Q(T_f) \neq F(T_f))]$. $\square$

Clearly, an adequate test suite is "stronger" than a relative adequate one. However, through the judicious proposal and application of **mutation operators** that create Φ_P, the finite set of syntactically incorrect programs, we can determine if a test suite can demonstrate that there is a difference between each $\phi_r \in \Phi_P$ and the actual program under test. Under a **strong mutation** test adequacy criterion, a test suite T is strong mutation adequate if it can **kill** each mutant in Φ_P by showing that the output of the mutant and the program under test differ. If a specific mutant $\phi_r \in \Phi_P$ remains alive after the execution of T, this could indicate that T does not have the ability to isolate the specific defect that ϕ_r represents [DeMillo and Offutt, 1991, Hamlet and Maybee, 2001]. Alternatively, it is possible that ϕ_r represents an **equivalent mutant**, or a program that is syntactically different from P while still having the ability to produce the same output as P.

Since strong mutation adequacy is often difficult to fulfill, the **weak mutation** test adequacy criterion only requires that the data states that occur after the execution of the initial source code location and the mutated code location differ [Hamlet and Maybee, 2001, Zhu et al., 1997]. Using the terminology established in our discussion of the fault/failure model in Section 3.2, weak mutation adequacy simply requires the execution of the source code mutation and the infection of the data state of each mutant program $\phi_r \in \Phi_P$. On the other hand, strong mutation adequacy requires the execution of the source code mutation, the infection of the data state of each mutant program, and the propagation of the mutant to the output. Both the strong and weak variants of mutation adequacy require the ability to automatically construct the mutants within Φ_P. While strong mutation adequacy only requires the ability to inspect the output of the program under test, weak mutation adequacy also requires additional instrumentation to reveal the data states that occur after a source code mutation is executed.

Ideally, mutation operators should produce the kinds of mutant programs that software engineers are most likely to create. A **fault model** can be used to describe the types of programming pitfalls that are normally encountered when a software system is implemented in certain types of programming languages. In a procedural programming language like C, it might be useful to include mutation operators that manipulate the relational operators found within a program. Therefore, the conditional logic statement `if(a < b)` in the program under test could require the creation of mutants that replace $<$ with every other relational operator in the set $\Re = \{<=, ==, >=, >, != \}$ [Hamlet and Maybee, 2001]. Other mutation operators might involve the manipulation of scalar and boolean variables and constants [Jezequel et al., 2001]. An arithmetic mutation operator designed to change arithmetic operators could mutate the assignment statement `a = b + 1` with the inverse of the `+` operator and create the mutant `a = b - 1` [Jezequel et al., 2001].

Of course, the inheritance, polymorphism, and encapsulation provided by object-oriented languages makes it more challenging to propose, design, and implement mutation operators for programming languages like Java. Alexander et al. and Kim et al. discuss some traditional "pitfalls" that are found in software systems that are implemented with object-oriented programming languages [Alexander et al., 2000, Kim et al., 2000]. However, these mutation operators lack generality because they are not supported by a fault model that clearly describes the class of potential faults that are related to the unique features of object-oriented programming languages. Kim et al. propose certain mutation operators for the Java programming language after conducting a Hazard and Operability (HAZOP) study [Leveson, 1995] of the Java programming language [Kim et al., 1999]. However, the authors omit the description of operators for some fundamental aspects of the object-oriented programming paradigm, such as certain forms of method overriding, and they do not include operators for select facets of the Java programming language [Kim et al., 1999]. Yet, Offutt et al. have proposed a fault model for object-oriented programming languages that describes common programmer faults related to inheritance and polymorphism [Offutt et al., 2001].

Using the fault model described in [Offutt et al., 2001], Ma et al. include a comprehensive listing of object-oriented mutation operators for the Java programming language [Ma et al., 2002]. The proposed mutation operators fall into the following six types of common object-oriented programming mistakes: (1) information hiding (access control), (2) inheritance, (3) polymorphism, (4), overloading, (5) Java-specific features, and (6) common programming mistakes [Ma et al., 2002]. The Access Modifier Change (AMC) mutation operator is an example of an operator that focuses on information hiding mistakes within object-oriented programs. For example, if a `BankAccount` class contained the declaration of the field `private double balance`, the AMC operator would produce the following mutants: `public double balance`, `protected double balance`, and `double balance`. The Java `this` keyword deletion (JTD) mutation operator removes the usage of `this` inside of the constructor(s) and method(s) provided by a Java class. For example, if the `BankAccount` class contained a `public BankAccount(double balance)` constructor, that used the `this` keyword to disambiguate between the parameter `balance` and the instance variable `balance`, the JTD operator would remove the keyword [Ma et al., 2002].

Figure 6 describes an algorithm that can be used to calculate the mutation adequacy score, $MS(P, T, M_o)$, for program P, test suite T, and a set of mutation operators M_o [DeMillo and Offutt, 1991]. Our description of the *CalculateMutationAdequacy* algorithm measures strong mutation adequacy and thus requires the outputs of P and the automatically generated mutants to differ. However, the algorithm could be revised to compute a weak mutation adequacy score for the test suite. The algorithm in Figure 6 uses a $n \times s$ matrix, $\mathcal{D}$, to store information about the dead mutants and the specific tests that killed the mutants. The s column vectors in the test information matrix $\mathcal{D}$ indicate whether each of the tests within T were able to kill one of the mutants in the set $\Phi_P = \{\phi_1, \ldots, \phi_s\}$. We use the notation $\mathcal{D}[f][r]$ to denote access to the fth row and the rth column within $\mathcal{D}$ and we use a 1 to indicate that a mutant was killed and a 0 to indicate that test T_f left mutant ϕ_r alive. Finally, the algorithm to calculate $MS(P, T, M_o)$ uses $\mathcal{Z}_{n \times s}$ to denote the $n \times s$ zero matrix and $\mathcal{Z}_s$ to denote a single column vector composed of s zeros.

If a specific test T_f is not able to kill the current mutant ϕ_r, it is possible that ϕ_r and P are equivalent. Since the determination of whether ϕ_r is an **equivalent mutant** is generally undecidable [Zhu et al., 1997], it is likely that the execution of the *IsEquivalentMutant* algorithm on line 11 of *CalculateMutationAdequacy* will require human intervention. When a mutant is not killed and it is determined that ϕ_r is an equivalent mutant, we place a 1 in $\mathcal{E}[r]$ to indicate that the current mutant is equivalent to P. The mutation testing information collected in $\mathcal{D}$ and $\mathcal{E}$ is complete once the algorithm has executed every test case against every mutant for every mutation location in program P. Line 13 computes the number of dead mutants, D_{num}, by using the *pos* function to determine if the sum of one of the s column vectors is

```
Algorithm CalculateMutationAdequacy($T, P, M_o$)
(* Calculation of Strong Mutation Adequacy *)
Input: Test Suite $T$;
       Program Under Test $P$;
       Set of Mutation Operators; $M_o$
Output: Mutation Adequacy Score; $MS(P, T, M_o)$
1.  $\mathcal{D} \leftarrow \mathcal{Z}_{n \times s}$
2.  $\mathcal{E} \leftarrow \mathcal{Z}_s$
3.  for $l \in$ ComputeMutationLocations($P$)
4.     do $\Phi_P \leftarrow$ GenerateMutants($l, P, M_o$)
5.        for $\phi_r \in \Phi_P$
6.           do for $T_f \in \langle T_1, \ldots, T_e \rangle$
7.              do $R_f^P \leftarrow$ ExecuteTest($T_f, P$)
8.                 $R_f^{\phi_r} \leftarrow$ ExecuteTest($T_f, \phi_r$)
9.                 if $R_f^P \neq R_f^{\phi_r}$
10.                   do $\mathcal{D}[f][r] \leftarrow 1$
11.                else if IsEquivalentMutant($P, \phi_r$)
12.                        do $\mathcal{E}[r] \leftarrow 1$
13. $D_{num} \leftarrow \sum_{r=1}^{s} pos(\sum_{f=1}^{n} \mathcal{D}[f][r])$
14. $E_{num} \leftarrow \sum_{r=1}^{s} \mathcal{E}[r]$
15. $MS(P, T, M_o) \leftarrow \frac{D_{num}}{(|\Phi_P| - E_{num})}$
16. return $MS(P, T, M_o)$
```

Figure 6: Algorithm for the Computation of Mutation Adequacy.

positive. We define the *pos* function to return 1 if $\sum_{f=1}^{n} \mathcal{D}[f][r] > 0$ and 0 otherwise. Finally, line 14 computes the number of equivalent mutants, E_{num}, and line 15 uses this information to calculate the final mutation adequacy score for program P and test suite T [DeMillo and Offutt, 1991].

While the calculation of mutation test adequacy is conceptually simple, it is computationally expensive. Choi et al. attempted to improve the cost-effectiveness and practicality of measuring mutation adequacy by parallelizing the steps in algorithm *ComputeMutationAdequacy* and scheduling the computation on the hypercube parallel computer architecture [Choi et al., 1989]. Krauser et al. have also investigated a number of techniques that can improve the performance of mutation analysis on a single instruction multiple data (SIMD) parallel computer architecture and evaluated these techniques with a detailed system model that supported a simulation-based empirical analysis [Krauser et al., 1991]. As noted in [Zhu et al., 1997], early attempts to improve the cost-effectiveness and practicality of mutation testing through parallelization had limited impact because they required specialized hardware and highly portable software. Yet, the availability of general purpose distributed computing middleware like Jini and JavaSpaces [Arnold et al., 1999, Edwards, 1999, Freemen et al., 1999] and high-throughput computing frameworks like Condor [Epema et al., 1996], will facilitate performance improvements in the algorithms that measure mutation adequacy.

In another attempt to make mutation adequacy analysis more cost-effective and practical, Offutt et al. have investigated the **N-selective** mutation testing technique that removes the N mutation operators that produce the most mutants of the program under test [Offutt et al., 1996, 1993, Zhu et al., 1997]. Even though all mutation operators make small syntactic changes to the source code of the program under test, some operators are more likely to produce a greater number of mutants. Furthermore, the **semantic impact**, or the change in meaning of the program under test, associated with small syntactic changes can vary from one mutation operator to another [Ma et al., 2002]. N-selective mutation test adequacy attempts to compute a high fidelity mutation adequacy score without executing the mutation operators that create a high number of mutants that do not truly shed light of the defect-revealing potential of the test suite. Ma et al. observe that their AMC operator creates the most mutants out of their set of object-oriented mutation operators [Ma et al., 2002]. Furthermore, Ma et al. also note that the Java `static` modified change (JSC) operator and the AMC operator have a tendency to produce a significant number of equivalent mutants [Ma et al., 2002].

Little is known about the subsumption relationship between different mutation-based test adequacy criterion and between mutation adequacy and other notions of test adequacy. However, Wong has shown that for the class of programs that have at least one variable and multiple definitions and uses of that variable, mutation adequacy is not comparable to the all-c-uses, all-p-uses, and all-uses adequacy criteria [Wong, 1993]. Furthermore, Budd has shown that if mutation analysis uses a mutation operator that replaces each conditional logic statement in a program with the conditionals `if(true)` and `if(false)`, then mutation test adequacy will subsume branch coverage [Budd, 1980]. Wong has also experimentally demonstrated, using an experiment design similar to those described in Section 3.4.4, that mutation test adequacy requires the construction of test suites that have greater fault detecting ability than the tests created to fulfill certain data flow-based test adequacy criteria [Wong, 1993]. It is clear that mutation analysis is a promising practical technique that requires further implementation, empirical analysis, and theoretical study.

3.4.3 Error-based

Error-based test adequacy criterion require a test suite T to demonstrate that the program under test P does not deviate from F, the program's specification, in a certain number of predefined ways. Certain elements of the **category-partition method** proposed by Balcer et al. and Ostrand and Balcer are indicative of error-based test adequacy criterion [Balcer et al., 1989, Ostrand and Balcer, 1988]. The category-partition method requires the analysis of F to create a partition of the input domain of the program under test. By relying upon the guidance of the tester, the category-partition method identifies the parameters and environment conditions, known as **categories**, that impact the behavior of the program under test. Next, the tester is responsible for the decomposition of each category into mutually exclusive **choices** that will be used to describe the partitions of the input within the category [Balcer et al., 1989].

The **test specification language** (TSL) provides a mechanism that allows a tester to write succinct descriptions of the categories and choices that state the input and output of the program under test. It is also possible for the tester to provide a description of the constraints that control the requirement of specific values within the choices and categories. In [Ostrand and Balcer, 1988], the authors describe the categories and choices that might be used for the `find <pattern> <file>` program that is often distributed with the Unix and GNU/Linux operating systems. For example, the specification for `find` might require that `<file>` is a valid file name. Thus, it would be important for the tests within T to ensure that `find` can handle the situations when: (1) there is a valid file associated with the provided name, (2) there is no file with the stated name, and (3) the file name is omitted when the usage of `find` occurs [Ostrand and Balcer, 1988].

Error-based test adequacy criterion judge a test suite to be "stronger" if it covers more of the identified categories and choices. Since there is no general technique to automatically create an error-based test adequacy criterion from F, most error-based adequacy metrics, such as those used in the category-partition method, require human intervention [Zhu et al., 1997]. However, with the complete description of F and the specification of the test adequacy criterion in TSL or some other test case specification language, it is possible to automatically generate a test suite that will fulfill the adequacy criterion. For example, the AETG system of Cohen et al. and the PairTest system of Lei et al. enable the generation of combinatorially balanced test suites from test specifications [Cohen et al., 1996, 1997, Lei and Tai, 1998]. While combinatorial test case generation is not discussed in more detail in this chapter, more information about test data generation algorithms and their relation to test adequacy criterion is provided in Section 3.5.

3.4.4 Comparing Test Adequacy Criteria

Many test adequacy criterion can be related in subsumption hierarchies. However, Ntafos has argued that some test adequacy criteria are incomparable under the subsumption relation [Ntafos, 1988]. Ntafos has also observed that even when it is possible to order test adequacy criterion through subsumption it is likely that this ordering will not provide any direct indication of either the effectiveness of test suites that fulfill the adequacy criteria or the costs associated with testing to the selected criteria [Ntafos, 1988]. Weyuker et al. categorized all comparisons of test adequacy criteria as being either **uniform** or **pointwise** [Weyuker et al., 1991]. A uniform comparison of test adequacy criteria C_α and C_β attempts to relate the requirements of the criteria in light of all possible programs P and all test suites T [Weyuker et al., 1991]. The usage of the subsumption relation can be seen as a type of uniform comparison of test adequacy criteria. However, a pointwise comparison of test adequacy criteria C_α and C_β compares the behavior of the two criteria with respect to a single (or, limited number of) P and T. While pointwise comparisons often provide interesting insights into the effectiveness of selected test adequacy criteria for certain programs, these types of comparisons often do not provide results that can be generalized. However, as noted in [Weyuker et al., 1991] there are also severe limitations associated with uniform comparisons:

> [...] We can conclude that a uniform comparison that guarantees one criterion to be more effective at detecting program defects than another for all programs is no comparison at all. This is a convincing argument against the use of comparisons that attempt to guarantee the relative fault-exposing ability of criteria for all programs.

In light of these concerns about comparing test adequacy criteria, Weyuker et al. conclude their thoughts with the following observation: "We see effectiveness and cost as the two most meaningful bases by which test criteria can be compared; effectiveness is our ultimate concern" [Weyuker et al., 1991]. To this end, Frankl, Weyuker, Weiss, and Hutchins et al. have conducted both analytical and empirical investigations of the effectiveness of test adequacy criteria [Frankl and Weiss, 1993, Frankl and Weyuker, 1993, Hutchins et al., 1994]. Hutchins et al. compared the effectiveness of the all-edges and **all-DU** test adequacy criteria. The all-DU test adequacy criterion is a modified version of all-uses that simply requires the coverage of def-use associations without distinguishing between a p-use and a c-use [Hutchins et al., 1994].

The experimental design of Hutchins et al. used the TSL system described in Section 3.4.3 to automatically generate an **initial test pool** (ITP) that was analyzed to determine the level of achieved adequacy. Next, an **additional test pool** (ATP) was created to ensure that each of the exercisable coverage units within their subject programs was touched by at least 30 test cases [Hutchins et al., 1994]. After the construction of a large test universe from the union of the ITP and the ATP, test sets of specific sizes were randomly selected from the test universe. Furthermore, the eight base programs selected by Hutchins et al. were seeded with defects that the researchers deemed to be representative in terms of their similarity to real-world defects and the "difficulty" associated with the isolation of the faults. An experimental design of this nature enabled Hutchins et al. to examine the relationship between the **fault detection ratio** for a testing technique and the adequacy and size of the resulting test suites. The fault detection ratio is the ratio between the number of test suites that contain a fault-revealing test case and the number of test suites whose adequacy is in a specific interval [Hutchins et al., 1994].

The empirical study conducted by Hutchins et al. reveals some interesting trends. For example, the fault detection ratios for their candidate programs rose sharply as the test adequacy increased above 80 or 90 percent. Furthermore, as the size of a test suite increased, the fault detection ratio also increased. However, the fault detection ratio for test suites that were completely adequate with respect to the all-edges and all-DU criteria varied significantly [Hutchins et al., 1994]. Indeed, Hutchins et al. observe that "the fault detection ratio of test sets with 100% DU coverage varied from .19 to 1.0 with an average of .67 for the 31 faults in the DU class" [Hutchins et al., 1994]. Perhaps more interesting is the following observation from Hutchins et al.:

> [...] Rather, code coverage seems to be a good indicator of test inadequacy. If apparently thorough tests yield only a low coverage level, then there is good reason to continue testing and try to raise the coverage level. The value of doing this can be seen by examining the detection ratios of test sets as their coverage levels approach 100%.

In a comparison of their operation difference (OD) test case selection technique, Harder et al. propose a new **area** and **stacking** approach for comparing test adequacy criteria [Harder et al., 2003].

Indeed, Harder et al. contend that the fault detection ratio is an inappropriate measure of the "efficiency" of test suites generated with respect to specific test adequacy criterion for two reasons. First, since the fault detection ratio vs. test suite size curve relationship is not necessarily linear in nature, there is no guarantee that a doubling in the size of a test suite will always double the ability of the tests to detect faults. Second, and more importantly, the fault detection ratio does not directly reveal which test adequacy criterion is most likely to engender the production of test suites that reveal the most defects [Harder et al., 2003].

Suppose that we were interested in comparing test adequacy criteria C_α and C_β using the area and stacking technique. In order to do so, we could use C_α and C_β to guide the manual and/or automatic generation of test suites T_α and T_β that obtain a specific level of adequacy with respect to the selected criteria. It is likely that the size of the generated test suites will vary for the two criteria and we refer to *size*(T_α) and *size*(T_β) as the *natural size* of the tests for the chosen criteria [Harder et al., 2003]. In an attempt to fairly compare C_α and C_β, Harder et al. advocate the construction of two new test suites T_α^β and T_β^α, where T_α^β denotes a test suite derived from T_α that has been stacked (or, reduced) to include *size*(T_β) tests and T_β^α is a test suite derived from T_β that has been stacked (or, reduced) to include *size*(T_α) test cases. Harder et al. propose stacking as a simple technique that increases or decreases the size of a base test suite by randomly removing tests or adding tests using the generation technique that created the base test suite [Harder et al., 2003].

In a discussion about the size of a test suite, Harder et al. observed that "comparing at any particular size might disadvantage one strategy or the other, and different projects have different testing budgets, so it is necessary to compare the techniques at multiple sizes" [Harder et al., 2003]. Using the base and faulty versions of the candidate programs produced by Hutchins et al., the authors measured the number of faults that were detected for the various natural sizes of the tests produced with respect to certain adequacy criterion. In our example, we could plot the number of revealed defects for the four test suites T_α, T_β, T_α^β and T_β^α at the two sizes of $size(T_\alpha)$ and $size(T_\beta)$. The calculation of the area underneath the two fault-detection vs. test suite size curves can yield a new view of the effectiveness of the test adequacy criteria C_α and C_β. However, the area and stacking technique for comparing test adequacy criteria has not been applied by other researchers and there is a clear need for the comparison of this design to past experimental designs. While the comparison of test adequacy criteria is clearly important, it is also an area of software testing that is fraught with essential and accidental difficulties.

3.5 Test Case Generation

The generation of test cases can be performed in a manual or automated fashion. Frequently, **manual test generation** involves the construction of test cases in a general purpose programming language or a test case specification language. Even though the `KineticTest` class in Figure 4 adheres to the JUnit testing framework, it could have also been specified in a programming language-independent fashion by simply providing the class under test, the method under test, the method input, and the expected output. This specification could then be transformed into a language-dependent form and executed in a specific test execution infrastructure. Alternatively, test cases can be "recorded" or "captured" by simply using the program under test and monitoring the actions that were taken during usage [Steven et al., 2000].

An automated solution to the test data generation problem attempts to automatically create a T that will fulfill selected adequacy criterion C when it is used to test program P. While it is possible for C to be an error-based criterion, automated test data generation is more frequently performed with fault-based and structurally-based test adequacy criteria. There are several different techniques that can be used to automatically generate test data. **Random**, **symbolic**, and **dynamic** test data generation approaches are all alternatives that can be used to construct a T that adequately tests P. A random test data generation approach relies upon a random number generator to simply generate test input values.[7] For complex (and, sometimes quite simple) programs, it is often difficult for random test data generation techniques to produce adequate test suites [Korel, 1996].

[7]Traditionally, random test data generation has been applied to programs that accept numerical inputs. However, these random number generators could be used to produce `String`s if we treat the numbers as ASCII or Unicode values. Furthermore, the random number generator could create complex abstract data types if we assign a semantic meaning to specifi c numerical values.

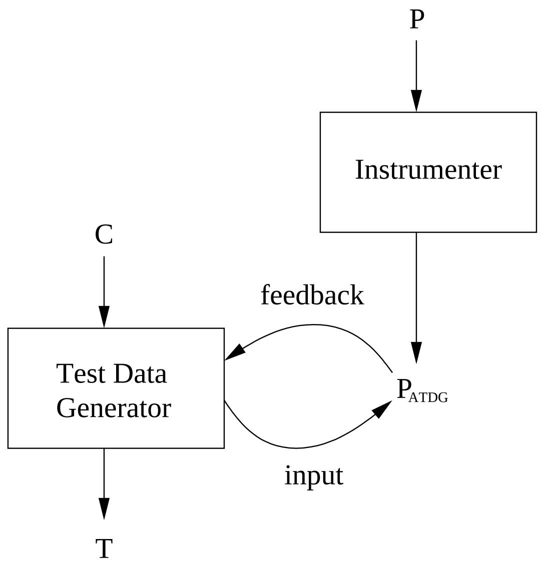


Figure 7: Dynamic Software Test Data Generation.

Symbolic test data generation attempts to express the program under test in an abstract and mathematical fashion. Intuitively, if all of the important aspects of a program can be represented as a system of one or more linear equations, it is possible to use algebraic techniques to determine the solution to these equations [Clarke, 1976, Ramamoorty et al., 1976]. Symbolic test data generation is appealing because it does not require the execution of the program under test. However, this type of symbolic generation has a limited practical value due to the problems associated with iteration constructs and arrays that depends upon other program variables and pointers [Michael et al., 2001]. Alternatively, Howe et al. and Memon et al. have chosen to represent aspects of the program under test with a specification of preconditions and postconditions and then express the desired test cases in terms of initial program states and goal program states [Howe et al., 1997, Memon et al., 2001a]. This type of abstract program representation facilitates the usage of an **artificial intelligence** (AI) **planner** that can automatically produce a test case that causes the program to progress from the start state to the goal state using the formally specified preconditions and postconditions.

Dynamic test data generation is an approach that actually relies upon the execution of P (or, some instrumented version of P) to generate an adequate T [Gupta et al., 1998, 1999, Korel, 1996, Michael et al., 2001]. While any dynamic test data generation approach must incur the overhead associated with actually executing the program under test, this execution can reveal additional insights about which test requirements have not been satisfied. Furthermore, dynamic test data generation techniques can use information gathered from program executions to determine how "close" the generator is to actually creating a test suite that satisfies C. Furthermore, dynamic test generation methods can handle arrays and pointer references because the values of array indices and pointers are know throughout the generation process. Figure 7 provides a view of the dynamic test data generation process that relies upon a program instrumenter to produce P_{ATDG}, a version of P that contains statements that allow P to interact with the test generation subsystem. The test data generator that takes C as input can interact with P_{ATDG} in order to facilitate the automatic generation of test suite T.

Frequently, dynamic test data generation is viewed as a function minimization problem [Ferguson and Korel, 1996, Korel, 1996, Michael et al., 2001]. However, it is important to note that some test data generation techniques that do use functions to represent a program do not focus on minimizing the selected functions [Fisher et al., 2002b]. For the purposes of our discussion, we will examine automated test data generation approaches that attempt to generate T by minimizing functions that describe P. For example, suppose that the selected adequacy criterion requires the execution of the `true` branch of the conditional logic statement `if( mass != 0 )` provided on line 8 of the `computeVelocity` method listed

Decision Example	**Objective Function**
`if(c <= d)`	$\mathcal{F}(x) = \begin{cases} c_i(x) - d_i(x), & \text{if } c_i(x) > d_i(x); \\ 0, & \text{otherwise.} \end{cases}$
`if(c >= d)`	$\mathcal{F}(x) = \begin{cases} d_i(x) - c_i(x), & \text{if } c_i(x) < d_i(x); \\ 0, & \text{otherwise.} \end{cases}$
`if(c == d)`	$\mathcal{F}(x) = \lvert c_i(x) - d_i(x) \rvert$
`if(c != d)`	$\mathcal{F}(x) = -\lvert c_i(x) - d_i(x) \rvert$
`if(b)`	$\mathcal{F}(x) = \begin{cases} 1000, & \text{if } b_i(x) = \textit{false}; \\ 0, & \text{otherwise.} \end{cases}$

Figure 8: General Form of Selected Objective Functions.

in Figure 3. We can ensure the execution of the `true` branch of this conditional logic statement by minimizing the objective function $\mathcal{F}(x) = -|m_8(x)|$ where $m_8(x)$ denotes the value of the variable `mass` on line 8 that was induced by test input value x. Since there is a wide range of program inputs that can lead to the satisfaction of this conditional logic statement, we can actually select any inputs that cause $\mathcal{F}(x)$ to evaluate to a negative output and not directly focus on finding the minimum of the function. Thus, automated test data generation often relies upon constrained function minimization algorithms that search for minimums within certain bounds [Ferguson and Korel, 1996].

Figure 8 provides the general form of the objective function for several different conditional logic forms. Our formulation of an objective function is based upon the "fitness functions" described in [Michael et al., 2001] and the "branch functions" used in [Ferguson and Korel, 1996]. In this figure, we use the notation $c_i(x)$ to denote the value of variable `c` on line i that was induced by test input x. Since dynamic test data generation attempts to minimize objective functions while executing the program under test, the actual value of either $c_i(x)$ or $d_i(x)$ is known during the attempt to generate an adequate T. Yet, the $c_i(x)$ and $d_i(x)$ used in objective functions might be complex functions of the input to the program under test. In this situation, an objective function cannot be minimized directly and it can only be used in a heuristic fashion to guide the test data generator [Michael et al., 2001]. Yet, even when $c_i(x)$ and $d_i(x)$ are complicated functions of test input x, an objective function can still be used to provide "hints" to the test data generator in an attempt to show whether the modification of the test input is improving test suite adequacy.

It is important to note that the formulations of the objective functions for the conditional logic statements `if(c >= d)`, `if(c <= d)`, `if(c == d)`, and `if(e)` are constructed in a fashion that causes the function to reach a minimum and then maintain this value. However, the conditional logic statement `if(c != d)` can create functions that have no specific minimum value and therefore must be minimized in a constrained fashion. Each of the functions in Figure 8 describe an $\mathcal{F}(x)$ that must be minimized in order to take the `true` branch of the condition; the objective functions for the `false` branches could be developed in an analogous fashion. Our discussion of objective functions for conditional logic predicates omits the details associated with dynamic test data generation for conjunctive and disjunctive predicates. In [Fisher et al., 2002b], the authors propose a type of objective function that can handle more complicated predicates. While we omit a discussion of the objective functions for conditional logic pred-

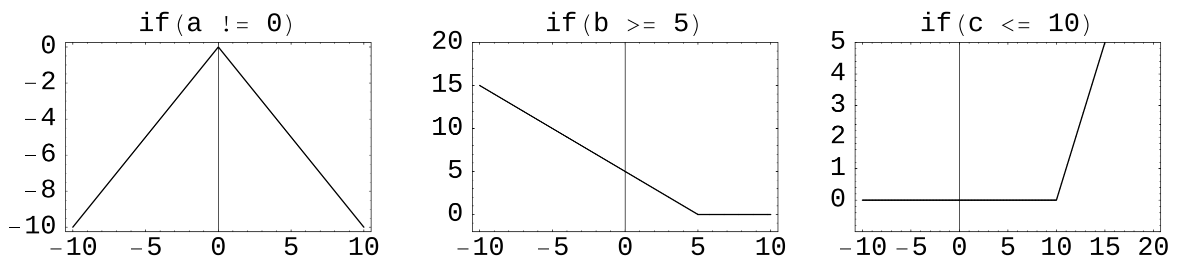


Figure 9: Objective Functions for Selected Conditional Logic Statements.

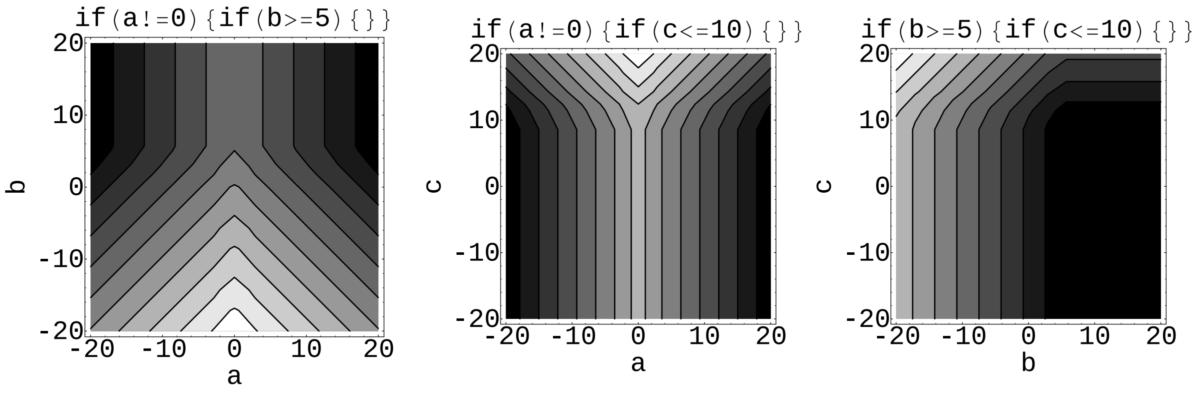


Figure 10: Objective Functions for Nested Conditional Logic Statements.

icates that include the $>$ and $<$ relational operators, [Ferguson and Korel, 1996] describe the form of the functions for conditional logic predicates that use these operators.

Figure 9 depicts the graphs of the objective functions for several different conditional logic statements. These graphs can be viewed as specific instantiations of the general objective functions described in Figure 8. The first graph represents the objective function for the conditional logic statement `if(a != 0)` where variable `a` is a component of test input x. The second and third graphs offer the plots of the objective functions associated with the conditional logic statements `if(b >= 5)` and `if(c <= 10)`, respectively. Of course, P might have multiple conditional logic statements that are nested in an arbitrary fashion. For example, the conditional logic block associated with the statement `if(a != 0)` might have a conditional logic block for `if(b >= 5)` located at an arbitrary position within it's own body. If $\mathcal{F}_a(x)$ corresponds to the objective function for `if(a != 0)` and $\mathcal{F}_b(x)$ represents the objective function for `if(b >= 5)`, then the objective function $\mathcal{F}_a(x) + \mathcal{F}_b(x)$ must be minimized in an attempt to exercise the `true` branches of the nested conditional logic statement `if(a != 0){<...>if(b >= 5){<...>}}` [Michael et al., 2001].

Figure 10 includes the contour plots for the three of the unique combinations of the objective functions that are described in Figure 9 (there are three alternative conditional logic nestings that we do not consider). For simplicity, we assume that the objective functions are direct functions of the test inputs so that $a_i(x) = a$, $b_i(x) = b$, and $c_i(x) = c$ and that the test input x is composed of three values (a, b, c). Since the addition of two objective functions for different program variables creates a three-dimensional function, these contour plots represent "high" areas in the combined objective function with light colors and "low" areas in the combined function with dark colors. Furthermore, each contour plot places the outer conditional logic statement on the x-axis and the inner conditional logic statement on the y-axis.

As expected, the contour plot for the nested conditional logic statement `if(a != 0){<...>if(b >= 5){<...>}}` is at minimum levels in the upper left and right corners of the plot. Since our automated test data generation algorithms attempt to seek minimums in objective functions, either of these

```
....F
Time: 0.026
There was 1 failure:

1) testFour(KineticTest)junit.framework.AssertionFailedError: expected:<20>
   but was:<24>

        at KineticTest.testFour(KineticTest.java:48)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke
         (NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke
         (DelegatingMethodAccessorImpl.java:25)

FAILURES!!!
Tests run: 4,  Failures: 1,  Errors: 0
```

Figure 11: The Results from Executing `KineticTest` in the `junit.textui.TestRunner`.

dark regions could lead to the production of test data that would cause the execution of the `true` branches of both conditional logic statements. While the contour plot associated with the nested conditional logic statement `if(a != 0){<...>if(c <= 5){<...>}}` also contains two dark regions on the left and right sections of the graph, the statement `if(b >= 0){<...>if(c <= 10){<...>}}` only has a single dark region on the right side of the plot.

3.6 Test Execution

The execution of a test suite can occur in a manual or automated fashion. For example, the test case descriptions that are the result of the test selection process could be manually executed against the program under test. However, we will focus on the automated execution of test cases and specifically examine the automated testing issues associated with the JUnit test automation framework [Hightower, 2001, Jeffries, 1999]. JUnit provides a number of `TestRunner`s that can automate the execution of any Java class that `extends` `junit.framework.TestCase`. For example, it is possible to execute the `KineticTest` provided in Figure 4 inside of either the `junit.textui.TestRunner`, `junit.awtui.TestRunner`, or the `junit.swingui.TestRunner`. While each `TestRunner` provides a slightly different interface, they adhere to the same execution and reporting principles. For example, JUnit will simply report “okay” if a test case passes and report a failure (or, error) with a message and a stack trace if the test case does not pass. Figure 11 shows the output resulting from the execution of the `KineticTest` provided in Figure 4 of Section 3.2.

The JUnit test automation framework is composed of a number of Java classes. Version 3.7 of the JUnit testing framework organizes its classes into nine separate Java packages. Since JUnit is currently released under the open source Common Public License 1.0, it is possible to download the source code from `http://www.junit.org` in order to learn more about the design and implementation choices made by Kent Beck and Erich Gamma, the creators of the framework. In this chapter, we will highlight some of the interesting design and usage issues associated with JUnit. More details about the intricacies of JUnit can be found in [Gamma and Beck, 2004, Jackson, 2003].

The `junit.framework.TestCase` class adheres to the `Command` design pattern and thus provides the `run` method that describes the default manner in which tests can be executed. As shown in Figure 4, a programmer can write a new collection of tests by creating a subclass of `TestCase`. Unless a `TestCase` subclass provides a new implementation of the `run` method, the JUnit framework is designed to call the default `setUp`, `runTest` and `tearDown` methods. The `setUp` and `tearDown` methods are simply responsible for creating the state of the class(es) under test and then “cleaning up” after a single test has executed. That is, JUnit provides a mechanism to facilitate the creation of independent test suites, as defined in Definition 3 of Section 2.1.

JUnit uses the `Composite` design pattern to enable the collection of `TestCase`s into a single `TestSuite`, as described by Definition 1 and Definition 2 in Section 2.1. The `Test` interface in JUnit has two subclasses: `TestCase` and `TestSuite`. Like `TestCase`, a `TestSuite` also has a `run`

```
import junit.framework.*;
public class BankAccountTest extends TestCase
{
  private BankAccount account = new BankAccount(1000);
  private static double NO_DELTA = 0.0;
  public BankAccountTest(String name)
  {
    super(name);
  }
  public static Test suite()
  {
    return new TestSuite(BankAccountTest.class);
  }

  public void testValidWithdraw()
  {
    double expected = 500.00;
    account.withdraw(500);
    double actual = account.getBalance();
    assertEquals(expected, actual, NO_DELTA);
  }
  public void testInvalidWithdraw()
  {
    try
    {
        account.withdraw(1500);
        fail(”Should have thrown OverdraftException”);
    }
    catch(OverdraftException e)
    {
        // test is considered to be successful
    }
  }
}
```

Figure 12: The `BankAccountTest` with Valid and Invalid Invocations of `withdraw`.

method. However, `TestSuite` is designed to contain 1 to e `TestCase`s and it's `run` method calls the `run` method of each of the instances of `TestCase` that it contains [Jackson, 2003]. A `TestCase` can describe the tests that it contains by providing a `suite` method. JUnit provides an interesting "shorthand" that enables a subclass of `TestCase` to indicate that it would simply like to execute all of the tests that it defines. The statement `return new TestSuite(KineticTest.class)` on line 10 of the `suite` method in Figure 4 requires the JUnit framework to use the Java reflection facilities to determine, at run-time, the methods within `KineticTest` that start with the required "`test`" prefix.

The `run` method in the `Test` superclass has the following signature: `public void run(TestResult result)` [Gamma and Beck, 2004]. Using the terminology established in [Beck, 1997, Gamma and Beck, 2004], `result` is known as a "collecting parameter" because it enables the collection of information about whether the tests in the test suite passed or caused a **failure** or an **error** to occur. Indeed, JUnit distinguishes between a test that fails and a test that raises an error. JUnit test cases include assertions about the expected output of a certain method under test or the state of the class under test and a failure occurs when these assertions are not satisfied. On the other hand, the JUnit framework automatically records that an error occurred when an unanticipated subclass of `java.lang.Exception` is thrown by the class under test. In the context of the terminology established in Section 2.1, JUnit's errors and failures both reveal faults in the application under test.

JUnit also facilitates the testing of the expected "exceptional behavior" of a class under test. For example, suppose that a `BankAccount` class provides a `withdraw(double amount)` method that raises an `OverdraftException` whenever the provided `amount` is greater than the `balance` encapsulated by the `BankAccount` instance. Figure 12 provides the `BankAccountTest` class that tests the normal and exceptional behavior of a `BankAccount` class. In this subclass of `TestCase`, the `testInvalidWithdraw` method is designed to fail when the `withdraw` method does not throw the `OverdraftException`. However, the `testValidWithdraw` method will only throw an exception if the `assertEquals(expected, actual, NO_DELTA)` is violated. In this test, the third param-

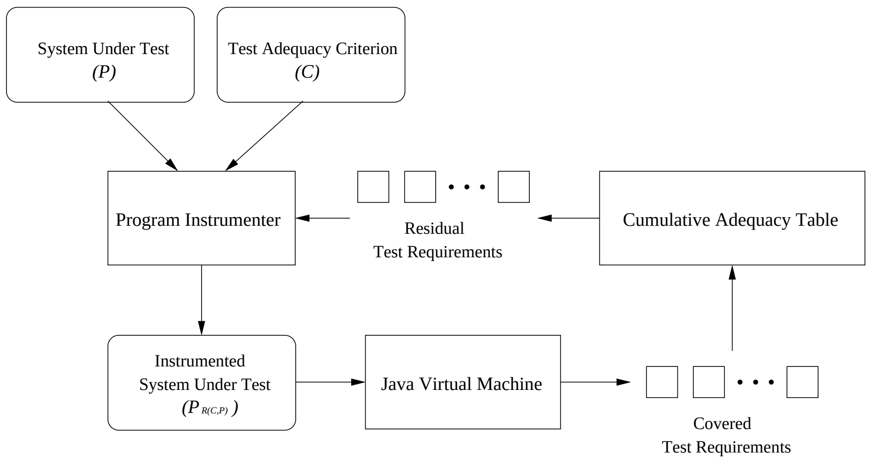


Figure 13: The Iterative Process of Residual Test Coverage Monitoring. *Source:* Christina Pavlopoulou and Michal Young. Residual test coverage monitoring. In *Proceedings of the 21st International Conference on Software Engineering*, pages 277–284. IEEE Computer Society Press, 1999. Permission granted.

eter in the call to `assertEquals` indicates that the test will not tolerate any small difference in the `double` parameters `expected` and `actual`.

3.7 Test Adequacy Evaluation

It is often useful to determine the adequacy of an existing test suite. For example, an automated test data generation algorithm might be configured to terminate when the generated test suite reaches a certain level of adequacy. If test suites are developed in a manual fashion, it is important to measure the adequacy of these tests to determine if the program under test is being tested "thoroughly." In our discussion of test adequacy evaluation, we use $R(C, P)$ to denote the set of test requirements (or, alternatively, the set of test descriptions) for a given test adequacy criterion C and a program under test P. If the all-nodes criterion was selected to measure to adequacy of a T used to test the `computeVelocity` method in Figure 3, then we would have $R(C, P) = \{enter_{cv}, n_6, n_7, n_8, n_{10}, n_{11}, n_{12}, n_{16}, n_{18}, exit_{cv}\}$. Alternatively, if C was the all-uses test adequacy criterion, then $R(C, P)$ would contain all of the def-c-use and def-p-use associations within `computeVelocity`. For example, line 11 of `computeVelocity` contains a definition of the variable `velocity` and lines 12 contains a computation-use of `velocity` and this def-use association would be include in $R(C, P)$.

Normally, the adequacy of test suite T is evaluated by instrumenting the program under test to produce $P_{R(C,P)}$, a version of P that can report which test requirements are covered during the execution of T. Pavlopoulou and Young have proposed, designed, and implemented a **residual test adequacy evaluator** that can instrument the program under test and calculate the adequacy of the test suites used during development [Pavlopoulou and Young, 1999]. Figure 13 provides a high-level depiction of this test adequacy evaluation system for Java programs. The residual coverage tool described by these authors can also measure the coverage of test requirements after a software system has been deployed and it is being used in the field. Finally, this test coverage monitoring tool provides the ability to incrementally remove the test coverage probes placed in the program under test after the associated test requirements have been exercised [Pavlopoulou and Young, 1999]. Pavlopoulou and Young report that the removal of the probes used to monitor covered test requirements often dramatically reduces the overhead associated with test adequacy evaluation [Pavlopoulou and Young, 1999].

3.8 Regression Testing

After a software system experiences changes in the form of bug fixes or additional functionality, a software maintenance activity known as regression testing can be used to determine if these changes introduced defects. As described in Section 2.2 and depicted in Figure 2, the regression testing process applies all of the other software testing stages whenever the program under test changes. The creation, maintenance, and execution of a regression test suite helps to ensure that the evolution of an application does not result in lower quality software. The industry experiences noted by Onoma et al. indicate that regression testing often has a strong positive influence on software quality [Onoma et al., 1998]. Indeed, the importance of regression testing is well understood. However, as noted by Beizer, Leung, and White, many software development teams might choose to omit some or all of the regression testing tasks because they often account for as much as one-half the cost of software maintenance [Beizer, 1990, Leung and White, 1989]. Moreover, the high costs of regression testing are often directly associated with the execution of the test suite. Other industry reports from Rothermel et al. show that the complete regression testing of a 20,000 line software system required seven weeks of continuous execution [Rothermel et al., 1999]. Since some of the most well-studied software failures, such as the Ariane-5 rocket and the 1990 AT&T outage, can be blamed on the failure to test changes in a software system [Hamlet and Maybee, 2001], many techniques have been developed to support efficient regression testing.

Several different methods have been developed in an attempt to reduce the cost of regression testing. Regression **test selection** approaches attempt to reduce the cost of regression testing by selecting some appropriate subset of the existing test suite [Ball, 1998, Rothermel and Harrold, 1997, Vokolos and Frankl, 1997]. Test selection techniques normally use the source code of a program to determine which tests should be executed during the regression testing stage [Rothermel and Harrold, 1996]. Regression **test prioritization** techniques attempt to order a regression test suite so that those tests with the highest priority, according to some established criterion, are executed earlier in the regression testing process than those with lower priority [Elbaum et al., 2000, Rothermel et al., 1999]. By prioritizing the execution of a regression test suite, these methods hope to reveal important defects in a software system earlier in the regression testing process. Regression **test distribution** is another alternative that can make regression testing more practical by more fully utilizing the computing resources that are normally available to a testing team [Kapfhammer, 2001].

Throughout our discussion of regression testing, we will continue to use the notation described in Section 2.1 and extend it with additional notation used in [Rothermel and Harrold, 1996]. Therefore, we will use P' to denote a modified version of program under test P. Problem 1 characterizes the regression testing problem in a fashion initially proposed in [Rothermel and Harrold, 1994]. It is important to note that any attempt to solve the regression testing problem while attempting to make regression testing more cost-effective could use regression test selection, prioritization, and distribution techniques in conjunction with or in isolation from one another. In Section 3.8.1 through Section 3.8.3, we iteratively construct a regression testing solution that can use test selection, prioritization, and distribution techniques.

Problem 1. Given a program P, its modified version P', and a test suite T that was used to previously test P, find a way to utilize T to gain sufficient confidence in the correctness of P'. $\Box$

3.8.1 Selective Regression Testing

Selective retest techniques attempt to reduce the cost of regression testing by identifying the portions of P' that must be exercised by the regression test suite. Intuitively, it might not be necessary to re-execute test cases that test source code locations in P' that are the same as the source locations in P. Any selective regression testing approach must ensure that it selects all of the test cases that are defect-revealing for P', if there are any defects within P'. Selective retesting is distinctly different from a retest-all approach that conservatively executes every test in an existing regression test suite. Figure 14 uses the *RTS* algorithm to express the steps that are commonly associated with a regression testing solution that uses a test selection approach [Rothermel and Harrold, 1996, 1998]:

Each one of the steps in the *RTS* algorithm addresses a separate facet of the regression testing problem. Line 1 attempts to select a subset of T that can still be used to effectively test P'. Line 3

```
Algorithm RTS($T, P, P'$)
(∗ Regression Testing with Selection ∗)
Input: Initial Regression Test Suite $T$;
       Initial Program Under Test $P$;
       Modified Program Under Test $P'$
Output: Final Regression Test Suite $T_L$
1.  $T' \leftarrow$ SelectTests($T, P, P'$)
2.  $R_1 \leftarrow$ ExecuteTests($T', P'$)
3.  $T'' \leftarrow$ CreateAdditionalTests($T', P, P', R_1$)
4.  $R_2 \leftarrow$ ExecuteTests($T'', P'$)
5.  $T_L \leftarrow$ CreateFinalTests($T, T', R_1, T'', R_2, P'$)
6.  return $T_L$
```

Figure 14: Regression Testing with Selection.

tries to identify portions of P' that have not been sufficiently tested and then seeks to create these new regression tests. Line 2 and line 4 focus on the efficient execution of the regression test suite and the examination of the testing results, denoted R_1 and R_2, for incorrect results. Finally, line 5 highlights the need to analyze the results of all previous test executions, the test suites themselves, and the modified program in order to produce the final test suite. When the *RTS* algorithm terminates, modified program P' becomes the new program under test and T_L is now treated as the test suite that will be used during any further regression testing that occurs after the new P changes. Traditionally, regression test selection mechanisms limit themselves to the problem described in line 1. Furthermore, algorithms designed to identify T' must conform to the **controlled regression testing assumption**. This assumption states that the only valid changes that can be made to P in order to produce P' are those changes that impact the source code of P [Rothermel and Harrold, 1997].

3.8.2 Regression Test Prioritization

Regression test prioritization approaches assist with regression testing in a fashion that is distinctly different from test selection methods. Test case prioritization techniques allow testers to order the execution of a regression test suite in an attempt to increase the probability that the suite might detect a fault at earlier testing stages [Elbaum et al., 2000, Rothermel et al., 1999, 2001a, Wong et al., 1997]. Figure 15 uses the formalization of Elbaum et al. and Rothermel and Harrold to characterize the typical steps taken by a technique that employs both selection and prioritization to solve the regression testing problem [Elbaum et al., 2000, Rothermel and Harrold, 1996].

The expression of the *RTSP* algorithm in Figure 15 is intended to indicate that a tester can use regression test selection and prioritization techniques in either a collaborative or independent fashion. Line 1 allows the tester to select T' such that it is a proper subset of T or such that it actually contains every test that T contains. Furthermore, line 2 could produce T'_r such that it contains an execution ordering for the regression tests that is the same or different than the ordering provided by T'. The other steps in the *RTSP* algorithm are similar to those outlined in Figure 14, the regression testing solution that relied only upon test selection.

The test case prioritization approaches developed by Elbaum et al. restrict themselves to the problem described in line 2. Testers might desire to prioritize the execution of a regression test suite such that code coverage initially increases at a faster rate. Or, testers might choose to increase the rate at which high-risk faults are first detected by a test suite. Alternatively, the execution of a test suite might be prioritized in an attempt to ensure that the defects normally introduced by competent programmers are discovered earlier in the testing process. Current techniques that prioritize a regression test suite by the fault-exposing-potential of a test case also rely upon the usage of mutation analysis [Elbaum et al., 2000, Rothermel et al., 2001a], as previously described in Section 3.4.2. When a regression test suite for a given program P is subjected to a mutation analysis, a finite set of syntactically different versions of P

```
**Algorithm** *RTSP*$(T, P, P')$
($*$ Regression Testing with Selection and Prioritization $*$)
**Input:** Initial Regression Test Suite $T$;
    Initial Program Under Test $P$;
    Modified Program Under Test $P'$
**Output:** Final Regression Test Suite $T_L$
1.  $T' \leftarrow$ SelectTests$(T, P, P')$
2.  $T'_r \leftarrow$ PermuteTests$(T', P, P')$
3.  $R_1 \leftarrow$ ExecuteTests$(T'_r, P')$
4.  $T'' \leftarrow$ CreateAdditionalTests$(T'_r, P, P', R_1)$
5.  $R_2 \leftarrow$ ExecuteTests$(T'', P')$
6.  $T_L \leftarrow$ CreateFinalTests$(T, T'_r, R_1, T'', R_2, P')$
7.  **return** $T_L$
```

Figure 15: Regression Testing with Selection and Prioritization.

are produced. Then, the entire regression test suite is executed for each one of the mutated versions of P in order to determine if the tests can detect the fault that each mutant represents. Even though mutation analysis has proven to be a viable regression test prioritization technique, its practicality is limited because it is so computationally intensive [Elbaum et al., 2000, Rothermel et al., 2001a].

Regression test suite prioritization algorithms are motivated by the empirical investigations of test adequacy criteria, as discussed in Section 3.4.4, which indicate that tests that are highly adequate are often more likely to reveal program defects. In an empirical evaluation of regression test suite prioritization, techniques that create a prioritized test suite can be evaluated based upon the weighted average of the percentage of faults detected over the life of the test suite, or the *APFD* [Elbaum et al., 2003]. For example, suppose that a regression test suite T was prioritized by its code coverage ability in order to produce T_r^c and was also prioritized by its fault exposing potential, thereby creating T_r^{fep}. If the faults in the program under test were known and the APFD of T_r^{fep} was greater than the APFD of T_r^c, then it is likely that we would prefer the usage of mutation adequacy over code coverage to prioritize the execution of T. Yet, Elbaum et al. caution against this interpretation when they observe that "to assume that a higher APFD implies a better technique, independent of cost factors, is an oversimplification that may lead to inaccurate choices among techniques" [Elbaum et al., 2003].

Since the APFD metric was used in early studies of regression test suite prioritization techniques and because it can still be used as a basis for more comprehensive prioritization approaches that use cost-benefit thresholds [Elbaum et al., 2003], it is important to investigate it in more detail. If we use the notation established in Section 2.1 and we have a test suite T and a total of g faults within program under test P, then Equation (18) defines the $APFD(T, P)$ [Elbaum et al., 2003]. We use *reveal*(i, T) to denote the position within T of the first test that reveals fault i.

$$APFD(T, P) = 1 - \frac{\sum_{i=1}^{g} \textit{reveal}(i, T)}{ng} + \frac{1}{2n} \tag{18}$$

For example, suppose that we have the test suite T with test sequence $\langle T_1, \ldots, T_5 \rangle$ and we know that the tests detect faults $f_1, \ldots, f_5$ in P according to Figure 16. Next, assume that *PermuteTests*(T', P, P') creates a T'_{r_1} with test sequence $\langle T_1, T_2, T_3, T_4, T_5 \rangle$, thus preserving the ordering of T. In this situation, we now have $APFD(T'_{r_1}, P) = 1 - .4 + .1 = .7$. If *PermuteTests*$(T', P, P')$ does change the order of T to produce T'_{r_2} with test sequence $\langle T_3, T_4, T_1, T_2, T_5 \rangle$, then we have $APFD(T'_{r_2}, P) = 1 - .2 + .1 = .9$. In this example, T'_{r_2} has a greater weighted average of the percentage of faults detected over its life than T'_{r_1}. This is due to the fact that the tests which are able to detect all of the faults in P, T_3 and T_4, are executed first in T'_{r_2}. Therefore, if the prioritization technique used to produce T'_{r_2} is not significantly more expensive than the one used to create T'_{r_1}, then it is likely a wise choice to rely upon the second prioritization algorithm for our chosen P and T.

Test Case	Faults				
	f_1	f_2	f_3	f_4	f_5
T_1			$\times$	$\times$	
T_2	$\times$	$\times$			
T_3	$\times$	$\times$	$\times$		
T_4			$\times$	$\times$	$\times$
T_5		$\times$	$\times$		

Figure 16: The Faults Detected by a Regression Test Suite $T = \langle T_1, \ldots, T_5 \rangle$.

3.8.3 Distributed Regression Testing

Any technique that attempts to distribute the execution of a regression test suite will rely upon the available computational resources during line 2, line 3, and line 5 of algorithm *RTSP* in Figure 15. That is, when tests are being selected, prioritized, or executed, distributed regression testing relies upon all of the available testing machines to perform the selection, prioritization, and execution in a distributed fashion. If the changes that are applied to P to produce P' involve the program's environment and this violates the controlled regression testing assumption, a distribution mechanism can be used to increase regression testing cost-effectiveness. When the computational requirements of test case prioritization are particularly daunting, a distributed test execution approach can be used to make prioritizations based upon coverage-levels or fault-exposing-potential more practical [Kapfhammer, 2001]. In situations where test selection and/or prioritization are possible, the distributed execution of a regression test suite can be used to further enhance the cost-effectiveness of the regression testing process. When only a single test machine is available for regression testing, the distribution mechanism can be disabled and the other testing approaches can be used to solve the regression testing problem.

3.9 Recent Software Testing Innovations

The software testing and analysis research community is actively proposing, implementing, and analyzing new software testing techniques. Recent innovations have been both theoretical and practical in nature. In this section, we summarize a selection of recent testing approaches that do not explicitly fit into the process model proposed in Section 2.2. Yet, it is important to note that many of these techniques have relationship(s) to the "traditional" phases described by our model.

3.9.1 Robustness Testing

A software system is considered to be **robust** if it can handle inappropriate inputs in a graceful fashion. Robustness testing is a type of software testing that attempts to ensure that a software system performs in an acceptable fashion when it is provided with anomalous input or placed in an inappropriate execution environment. Robustness testing is directly related to the process of hardware and software **fault injection**. For example, the FUZZ system randomly injects data into selected operating system kernel and utility programs in order to facilitate an empirical examination of operating system robustness [Miller et al., 1990]. Initial studies indicate that it was possible to crash between 25% and 33% of the utility programs that were associated with 4.3 BSD, SunOS 3.2, SunOS 4.0, SCO Unix, AOS Unix, and AIX 1.1 Unix [Miller et al., 1990]. Subsequent studies that incorporated additional operating systems like AIX 3.2, Solaris 2.3, IRIX 5.1.1.2, NEXTSTEP 3.2 and Slackware Linux 2.1.0 indicate that there was a noticeable improvement in operating system utility robustness during the intervening years between the first and second studies [Miller et al., 1998]. Interestingly, the usage of FUZZ on the utilities associated with a GNU/Linux operating system showed that these tools exhibited significantly higher levels of robustness than the commercial operating systems included in the study [Miller et al., 1998].

Other fault injection systems, like FTAPE, rely upon the usage of computer hardware to inject meaningful faults into a software system [Tsai and Iyer, 1995]. Yet, most recent fault injection systems, such as Ballista, do not require any special hardware in order to perform robustness testing. Ballista can perform robustness testing on the Portable Operating System Interface (POSIX) API in order to assess the robustness of an entire operating system interface [Koopman and DeVale, 2000]. Instead of randomly supplying interfaces with test data, Ballista builds test cases that are based upon the data types that are associated with procedure parameters. Ballista associates each of the possible parameter data types with a finite set of test values that can be combined to generate test inputs for operations.

A Ballista test case can be automatically executed in an infrastructure that supports the testing of operating systems such as AIX 4.41, Free BSD 2.2.5, HP-UX 10.20, Linux 2.0.18, and SunOS 5.5 [Koopman and DeVale, 2000]. Finally, if a test causes the operating system to perform in an anomalous fashion, the test records the severity of the test results based upon the **CRASH scale** that describes *C*atastrophic, *R*estart, *A*bort, *S*ilent, and *H*indering failures. According to [Biyani and Santhanam, 1997], [Krop et al., 1998], and [Koopman and DeVale, 2000], a catastrophic failure occurs when a utility program (or an operating system API) causes the system to hang or crash while a restart failure happens when the procedure under test never returns control to the test operation in a specific robustness test. Furthermore, abort failures are traditionally associated with "core dumps" and silent failures occur when the operating system does not provide any indication of the fact that the robustness testing subject just performed in an anomalous fashion. Finally, hindering failures happen when the procedure under test returns an error code that does not properly describe the exceptional condition that arose due to robustness testing. An empirical analysis of Ballista's ability to detect robustness failures indicates that a relatively small number of POSIX functions in the candidate operating systems did not ever register on the CRASH scale [Krop et al., 1998].

Several robustness testing systems have also been developed for the Windows NT operating system [Ghosh et al., 1998, 1999, Tsai and Singh, 2000]. For example, Ghosh et al. discuss the fault injection simulation tool (FIST) and a wrapping technique that can improve the robustness of Windows NT applications. In [Ghosh et al., 1999], the authors describe the usage of FIST to isolate a situation in which Microsoft Office 97 performs in a non-robust fashion and then propose a wrapping technique that can enable the Office application to correctly handle anomalous input. Furthermore, an empirical analysis that relied upon the application of a fault injection tool called the Random and Intelligent Data Design Library Environment (RIDDLE) revealed that GNU utilities ported to Windows NT appear to be less robust than the same utilities on the Unix operating system [Ghosh et al., 1998]. Finally, the dependability test suite (DTS) is another fault injection tool that can be used to determine the robustness of Windows NT applications such as Microsoft IIS Server and SQL Server [Tsai and Singh, 2000]. While robustness testing and software fault injection frequently focus their testing efforts at the operating system level, Haddox et al. have developed techniques that use fault injection to test and analyze software **commercial off the shelf**, or **COTS**, components of a finer level of granularity than a complete operating system API or an entire application [Haddox et al., 2001, 2002].

3.9.2 Testing Spreadsheets

Our discussion of software testing has generally focused on the testing and analysis of programs that are written in either procedural or object-oriented programming languages. However, Rothermel et al. and Fisher et al. have focused on the testing and analysis of programs written in spreadsheet languages [Fisher et al., 2002a,b, Rothermel et al., 1997, 2000, 2001b]. The form-based visual programming paradigm, of which spreadsheet programs are a noteworthy example, is an important mode of software development. Indeed, Frederick Brooks makes the following observation about spreadsheets and databases: "These powerful tools, so obvious in retrospect and yet so late in appearing, lend themselves to a myriad of uses, some quite unorthodox" [Jr., 1995]. However, until recently, the testing and analysis of spreadsheet programs has been an area of research that has seen little investigation. Rothermel et al. echo this sentiment in the following observation [Rothermel et al., 2000]:

> Spreadsheet languages have rarely been studied in terms of their software engineering properties. This is a serious omission, because these languages are being used to create production software upon which real decisions are being made. Further, research shows that many spreadsheets created with these languages contain faults. For these reasons, it is important

to provide support for mechanisms, such as testing, that can help spreadsheet programmers determine the reliability of values produced by their spreadsheets.

Rothermel et al. have described some of the differences between form-based programming languages and imperative programming language paradigms, noting that these "programs" are traditionally composed of cells that have formulas embedded within them [Rothermel et al., 1997]. These authors have proposed the **cell relation graph** (CRG) as an appropriate model for a form-based program that is loosely related to the control-flow graph of an imperative program [Rothermel et al., 1997]. The CRG includes information about the control flow within the formulas that are embedded within cells and the dependencies between the program's cells. Rothermel et al. have also examined the usefulness of different understandings of test adequacy for spreadsheet programs, such as traditional node and edge-based criteria defined for a program's CRG. However, Rothermel et al. conclude that a data flow-based test adequacy criterion is most appropriate for form-based programs because it models the definition and usage of the cells within a program. Furthermore, since form-based programs do not contain constructs such as array indexing operations and pointer accesses, it is often much easier to perform data flow analysis on the CRG of a spreadsheet program than it would be to perform a data flow analysis on the ICFG of an imperative program [Rothermel et al., 1997].

In [Rothermel et al., 2001b], the authors provide all of the algorithms associated with a testing methodology for the Forms/3 spreadsheet language. In this chapter, we will focus on the issues associated with automated test data generation [Fisher et al., 2002b] and test reuse [Fisher et al., 2002a] for programs written in spreadsheet languages. The "What You See Is What You Test" (WYSIWYT) spreadsheet testing methodology proposed by Fisher et al. and Rothermel et al. includes a test data generation mechanism for the **output-influencing-all-du-pairs** (or, oi-all-du-pairs) test adequacy criterion that is based upon the CRG representation of a spreadsheet [Fisher et al., 2002b, Rothermel et al., 2001b]. This test adequacy criterion is similar to the standard all-uses and all-DUs criteria, except that it requires the coverage of the def-use associations in a spreadsheet's CRG that influence the cells that contain user-visible output. The goal-oriented test data generation approach implemented by the authors is similar to Ferguson and Korel's chaining approach [Ferguson and Korel, 1996] because it represents a spreadsheet as a collection of branch functions. The constrained linear search procedure described in [Fisher et al., 2002b] attempts to exercise uncovered output influencing def-use associations by causing the branch functions associated with the selected CRG path to take on positive values.

Test suite reuse is also an important facet of spreadsheet testing because end-users often share spreadsheets and production spreadsheets must be revalidated when new versions of commercial spreadsheet engines are released [Fisher et al., 2002a]. Current spreadsheet reuse algorithms are similar to the regression test suite selection techniques presented in Section 3.8.1. When a spreadsheet user modifies a spreadsheet application, the reuse algorithms must determine which portion of the test suite must be executed in order to validate the program in a manner that is cost-effective and practical. Since the WYSIWYT tools are designed to be used by a spreadsheet developer in a highly interactive fashion, the timely retesting of a spreadsheet often can only occur in the test suite is reused in an intelligent fashion. In [Fisher et al., 2002a], the authors propose specific test reuse actions that should occur when the spreadsheet developer deletes a cell, changes the formula within a spreadsheet cell, or inserts a new cell into a spreadsheet. For example, if the programmer changes the formula within a specific spreadsheet cell, the test reuse algorithms must select all of the existing test(s) that validate the output of the modified cell and any other impacted cells. Alternatively, if the formula changes for a certain cell change cause the formula to rely upon new and/or different spreadsheet input cells, some test cases might be rendered obsolete [Fisher et al., 2002a].

3.9.3 Database-Driven Application Testing

Even simple software applications have complicated and ever-changing operating environments that increase the number of interfaces and the interface interactions that must be tested. Device drivers, operating systems, and databases are all aspects of a software system's environment that are often ignored during testing [Whittaker, 2000, Whittaker and Voas, 2000]. Yet, relatively little research has specifically focused on the testing and analysis of applications that interact with databases. Chan and Cheung have proposed a technique that can test database-driven applications that are written in a general purpose programming language, like Java, C, or C++, and include embedded structured query language (SQL) statements that are designed to interact with a relational database [Chan and Cheung, 1999a,b]. In their approach, Chan and Cheung transform the embedded SQL statements within a database-driven application into general purpose programming language constructs. In [Chan and Cheung, 1999a], the authors provide C code segments that describe the selection, projection, union, difference, and cartesian product operators that form the relational algebra and thus heavily influence the structured query language. Once the embedded SQL statements within the program under test have been transformed into general purpose programming language constructs, it is possible to apply traditional test adequacy criteria, as described in Section 3.4, to the problem of testing programs that interact with one or more relational databases.

Chays et al. and Chays and Deng have described the challenges associated with testing database-driven applications and proposed the AGENDA tool suite as a solution to a number of these challenges [Chays et al., 2000, 2002, Chays and Deng, 2003]. In fact, Chays et al. observe that measuring the "correctness" of database-driven applications might involve the following activities: (1) determining whether the program behaves according to specification, (2) deciding whether the relational database schema is correctly designed, (3) showing that the database is secure, (4) measuring the accuracy of the data within the database, and (5) ensuring that the database management system correctly performs the SQL operations required by the application itself [Chays et al., 2000]. In [Chays et al., 2000], the authors propose a partially automatable software testing methodology, inspired by the category-partition method described in Section 3.4.3, that addresses the first understanding of database-driven application correctness. When provided with the relational schema of the database(s) used by the program under test and a description of the categories and choices for the attributes required by the relational tables, the AGENDA tool can generate meaningful test databases [Chays et al., 2002, Chays and Deng, 2003]. AGENDA also provides a number of database testing heuristics, such as "determine the impact of using attribute boundary values" or "determine the impact of `null` attribute values" that can enable the tester to gain insights into the behavior of a program when it interacts with a database that contains "interesting" states [Chays et al., 2000, 2002, Chays and Deng, 2003]. Deng et al. have recently extended AGENDA to support the testing of database transaction concurrency by using a dataflow analysis to determine database transaction schedules that might reveal program faults [Deng et al., 2003].

While the AGENDA tool suite does provide innovative techniques for populating the relational database used by a database-driven application, it does not explicitly test the interactions between the program and the database. As noted by Daou et al. and Kapfhammer and Soffa, a database interaction point in a program can be viewed as an interaction with different entities of a relational database, depending upon the granularity with which we view the interaction [Daou et al., 2001, Kapfhammer and Soffa, 2003]. That is, we can view a SQL statement's interaction with a database at the level of databases, relations, records, attributes, or attribute values. In [Kapfhammer and Soffa, 2003], the authors propose an approach that can enumerate all of the relational database entities that a database-driven program interacts with and then create a **database interaction control flow graph** (DICFG) that specifically models the actions of the program and its interactions with a database [Kapfhammer and Soffa, 2003]. As an example, the `lockAccount` method provided in Figure 17 could be a part of a database-driven ATM application. Lines 6 and 7 of this program contain a database interaction point where the `lockAccount` method sends a SQL update statement to a relational database. Figure 18 offers a database interaction control flow graph for the `lockAccount` method that represents the operation's interaction with the relational database at the level of the database and attribute interactions.[8]

After proposing a new representation for database-driven applications, Kapfhammer and Soffa also describe a family of test adequacy criteria that can facilitate the measurement of the test suite quality for programs that interact with relational database. The **all-database-DUs**, **all-relation-DUs**, **all-record-DUs**, **all-attribute-DUs**, and **all-attribute-value-DUs** test adequacy criteria that are extensions of the traditional all-DUs proposed by Hutchins et al. and discussed in Section 3.4.4. Definition 13 defines the all-relation-DUs test adequacy criterion. In this definition, we use R_l to denote the set of all the database

[8]In this example, we assume that the program interacts with a single relational database called `Bank`. Furthermore, we assume that the `Bank` database contains two relations, `Account` and `UserInfo`. Finally, we require the `Account` relation to contain the attributes `id`, `acct_name`, `balance`, and `card_number`.

```
public boolean lockAccount(int card_number)
  throws SQLException
{
  boolean completed = false;
  String qu_lock =
    "UPDATE UserInfo SET acct_lock=1 WHERE card_number=" +
       card_number + ";";
  Statement update_lock = m_connect.createStatement();
  int result_lock = update_lock.executeUpdate(qu_lock);
  if( result_lock == 1)
  {
     completed = true;
  }
    return completed;
}
```

Figure 17: The `lockAccount` in an ATM Application.

relations that are interacted with by a method in a database-driven application. While Kapfhammer and Soffa define the all-relation-DUs and other related test adequacy criteria in the context of the database interaction control flow graph for a single method, the criteria could be defined for a "database enhanced" version of the class control flow graph or the interprocedural control flow graph [Kapfhammer and Soffa, 2003].

Definition 13. A test suite T for database interaction control flow graph $G_{DB} = (N_{DB}, E_{DB})$ satisfies the all-relation-DUs test adequacy criterion if and only if for each association $\langle n_d, n_{use}, var_{DB} \rangle$, where $var_{DB} \in R_l$ and $n_d, n_{use} \in N_{DB}$, there exists a test in T to create a complete path $\pi^{var_{DB}}$ in G_{DB} that covers the association. $\Box$

3.9.4 Testing Graphical User Interfaces

The graphical user interface (GUI) is an important component of many software systems. While past estimates indicated that an average of 48% of an application's source code was devoted to the interface [Myers and Rosson, 1992], current reports reveal that the GUI represents 60% of the overall source of a program [Memon, 2002]. While past research has examined user interface (UI) usability and widget layout [Sears, 1993], interactive system performance [Endo et al., 1996], and GUI creation framework performance [Howell et al., 2003], relatively little research has focused on the testing and analysis of graphical user interfaces. Memon et al. have conducted innovative research that proposes program representations, test adequacy criteria, and automated test data generation algorithms that are specifically tailored for programs with graphical user interfaces [Memon et al., 2001a,b, Memon, 2002]. As noted in the following observation from [Memon et al., 1999], the testing of GUIs is quite challenging.

> In particular, the testing of GUIs is more complex than testing conventional software, for not only does the underlying software have to be tested but the GUI itself must be exercised and tested to check for bugs in the GUI implementation. Even when tools are used to generate GUIs automatically, they are not bug free, and these bugs may manifest themselves in the generated GUI, leading to software failures.

To complicate matters further, the space of possible GUI states is extremely large. Memon et al. chose to represent a GUI as a series of operators that have preconditions and postconditions related to the state of the GUI [Memon et al., 1999, 2001a]. This representation classifies the GUI events into the categories of menu-open events, unrestricted-focus events, restricted-focus events, and system-interaction events [Memon et al., 2001a]. Menu-open events are normally associated with the usage of the pull-down menus in a GUI and are interesting because they do not involve interaction with the underlying application. While unrestricted-focus events simply expand the interaction options available to a GUI user, restricted focus events require the attention of the user before additional interactions can occur. Finally, system interaction events require the GUI to interact with the actual application [Memon et al., 2001a]. In order to perform automated test data generation, Memon et al. rely upon artificial intelligence planners that can

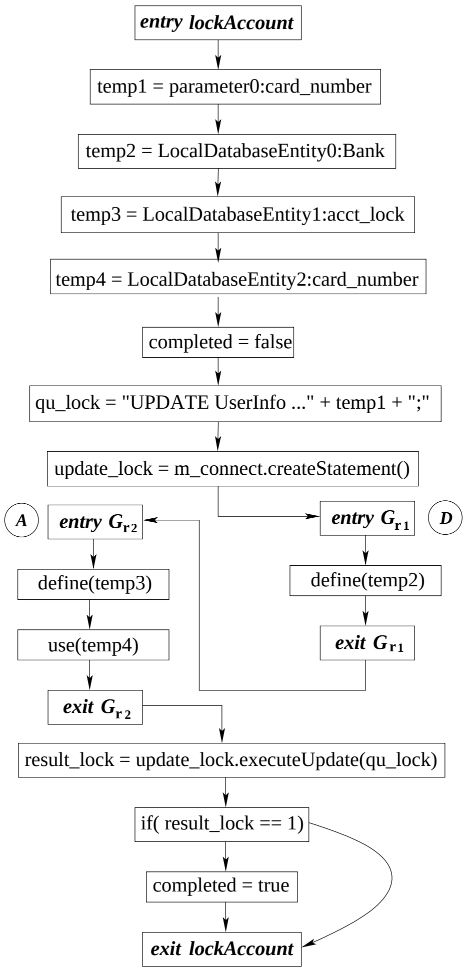

entry lockAccount
temp1 = parameter0:card_number
temp2 = LocalDatabaseEntity0:Bank
temp3 = LocalDatabaseEntity1:acct_lock
temp4 = LocalDatabaseEntity2:card_number
completed = false
qu_lock = "UPDATE UserInfo ..." + temp1 + ";"
update_lock = m_connect.createStatement()
A
entry G_{r2}
D
entry G_{r1}
define(temp3)
define(temp2)
use(temp4)
exit G_{r1}
exit G_{r2}
result_lock = update_lock.executeUpdate(qu_lock)
if(result_lock == 1)
completed = true
exit lockAccount

Figure 18: A DICFG for `lockAccount`. *Source:* Gregory M. Kapfhammer and Mary Lou Soffa. A Family of Test Adequacy Criteria for Database-Driven Applications. In *Proceedings of the 9th European Software Engineering Conference and the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM Press. Permission granted.

use the provided GUI events and operations to automatically produce tests that cause the GUI to progress from a specified initial GUI state to a desired goal state.

In an attempt to formally describe test adequacy criteria for GUI applications, Memon et al. propose the event-flow graph as an appropriate representation for the possible interactions that can occur within a graphical user interface component [Memon et al., 2001b]. Furthermore, the integration tree shows the interactions between all of the GUI components that comprise a complete graphical interface. Using this representation, Memon et al. define intra-component and inter-component test adequacy criteria based upon GUI event sequences. The simplest intra-component test adequacy criterion, **event-interaction coverage**, requires a test suite to ensure that after a certain GUI event e has been performed, all events that directly interact with e are also performed [Memon et al., 2001b]. The **length-n event sequence** test adequacy criterion extends the simple event-interaction coverage by requiring a context of n events to occur before GUI event e actually occurs. Similarly, Memon et al. propose inter-component test adequacy criteria that generalize the intra-component criteria and must be calculated by using the GUI's integration tree [Memon et al., 2001b]

4 Conclusion

Testing is an important technique for the improvement and measurement of a software system's quality. Any approach to testing software faces essential and accidental difficulties. Indeed, as noted by Edsger Dijkstra the construction of the needed test programs is a "major intellectual effort" [Dijkstra, 1968]. While software testing is not a "silver bullet" that can guarantee the production of high quality applications, theoretical and empirical investigations have shown that the rigorous, consistent, and intelligent application of testing techniques can improve software quality. Software testing normally involves the stages of test case specification, test case generation, test execution, test adequacy evaluation, and regression testing. Each of these stages in our model of the software testing process plays an important role in the production of programs that meet their intended specification. The body of theoretical and practical knowledge about software testing continues to grow as research expands the applicability of existing techniques and proposes new testing techniques for an ever-widening range of programming languages and application domains.

5 Defining Terms

Software Verification: The process of ensuring that a program meets its intended specification.

Software Testing: The process of assessing the functionality and correctness of a program through execution or analysis.

Failure: The external, incorrect behavior of a program.

Fault: A collection of program source statements that cause a program failure.

Error: A mistake made by a programmer during the implementation of a software system.

Test Case Specification: The process of analyzing the program under test, in light of a chosen test adequacy criterion, in order to produce a list of tests that must be provided in order to create a completely adequate test suite.

Test Case Generation: The manual or automatic process of creating test cases for the program under test. Automatic test case generation can be viewed as an attempt to satisfy the constraints imposed by the selected test adequacy criteria.

Test Adequacy Evaluation: The measurement of the quality of an existing test suite for a specific test adequacy criterion and a selected program under test.

Regression Testing: An important software maintenance activity that attempts to ensure that the addition of new functionality and/or the removal of program faults does not negatively impact the correctness of the program under test.

PIE Model: A model proposed by Voas which states that a fault will only manifest itself in a failure when it is executed, it infects the program data state, and finally propagates to the output.

Coincidental Correctness: A situation when a fault in a program does not manifest itself in a failure even though the fault has been executed and it has infected the data state of the program.

Interprocedural Control Flow Graph: A graph-based representation of the static control flow for an entire program. In object-oriented programming languages, the interprocedural control flow graph is simply a collection of the intraprocedural control flow graphs for each of the methods within the program under test.

Subsumption: A relationship between two test adequacy criterion. Informally, if test adequacy criterion C_α subsumes C_β, then C_α is considered to be "stronger" than C_β.

Complete Path: A path in a control flow graph that starts at the program graph's entry node and ends at its exit node.

All-nodes/Statement Coverage: A test adequacy criterion that requires the execution of all the statements within the program under test.

All-edges/Branch Coverage: A test adequacy criterion that requires the execution of all the branches within the program under test.

Multiple Condition Coverage: A test adequacy criterion that requires a test suite to account for every permutation of the boolean variables in every branch of a program.

Condition: An expression in a conditional logic predicate that evaluates to `true` or `false` while not having any other boolean valued expressions within it.

Condition-decision Coverage: A test adequacy criterion that requires a test suite to cover all of the edges within a program's control flow graph and to ensure that each condition evaluates to `true` and `false` at least one time.

All-uses: A test adequacy criterion, used as the basis for definition-use testing, that requires the coverage of all the definition-c-use and definition-p-use associations within the program under test.

Competent Programmer Hypothesis: An assumption that competent programmers create programs that compile and very nearly meet their specification.

Coupling Effect: An assumption that test suites that can reveal simple defects in a program can also reveal more complicated combinations of simple defects.

Mutation Adequacy/Relative Adequacy: A test adequacy criterion, based upon the competent programmer hypothesis and the coupling effect assumption, that requires a test suite to differentiate between the program under test and a set of programs that contain common programmer errors.

Strong Mutation Adequacy: A test adequacy criterion that requires that the mutant program and the program under test produce different output. This adequacy criterion requires the execution, infection, and propagation of the mutated source locations within the mutant program.

Weak Mutation Adequacy: A test adequacy criterion that requires that the mutant program and the program under test produce different data states after the mutant is executed. This test adequacy criterion requires the execution and infection of the mutated source locations within the mutant program.

Mutation Operator: A technique that modifies the program under test in order to produce a mutant that represents a faulty program that might be created by a competent programmer.

Equivalent Mutant: A mutant that is not distinguishable from the program under test. Determining whether a mutant is equivalent is generally undecidable. When mutation operators produce equivalent mutants, the calculation of mutation adequacy scores often requires human intervention.

N-selective Mutation Testing: A mutation testing technique that attempts to compute a high fidelity mutation adequacy score without executing the mutation operators that create the highest number of mutants and do not truly shed light on the defect-revealing potential of the test suite.

Category-partition Method: A partially automatable software testing technique that enables the generation of test cases that attempt to ensure that a program meets its specification.

Fault Detection Ratio: In the empirical evaluation of test adequacy criteria, the ratio between the number of test suites whose adequacy is in a specific interval and the number of test suites that contain a fault-revealing test case.

Residual Test Adequacy Evaluator: A test evaluation tool that can instrument the program under test in order to determine the adequacy of a provided test suite. A tool of this nature inserts probes into the program under test in order to measure adequacy and can remove these probes once certain test requirements have been covered.

Regression Test Suite Selection: A technique that attempts to reduce the cost of regression testing by selecting some appropriate subset of an existing test suite for execution.

Regression Test Suite Prioritization: A technique that attempts to order a regression test suite so that the test cases that are most likely to reveal defects are executed earlier in the regression testing process.

Regression Test Suite Distribution: A technique that attempts to make regression testing more cost-effective and practical by using all of the available computational resources during test suite selection, prioritization, and execution.

Robustness Testing/Fault Injection: A software testing technique that attempts to determine how a software system handles inappropriate inputs.

Commercial off the Shelf Component: A software component that is purchased and integrated into a system. Commercial off the shelf components do not often provide source code access.

References

Roger T. Alexander, James M. Bieman, and John Viega. Coping with Java programming stress. *IEEE Computer*, 33 (4):30–38, April 2000.

Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath. *The Jini Specification*. Addison-Wesley, Inc., Reading, MA, 1999.

M. Balcer, W. Hasling, and T. Ostrand. Automatic generation of test scripts from formal test specifications. In *Proceedings of the ACM SIGSOFT Third Symposium on Software Testing, Analysis, and Verification*, pages 210–218. ACM Press, 1989.

T. Ball. The limit of control flow analysis for regression test selection. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 134–142. ACM Press, March 1998.

Kent Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1997.

Boris Beizer. *Software Testing Techniques*. Van Nostrong Reinhold, New York, NY, 1990.

Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, Boston, MA, 1999.

R. Biyani and P. Santhanam. TOFU: Test optimizer for functional usage. *Software Engineering Technical Brief*, 2(1), 1997.

Jonathan P. Bowen and Michael G. Hinchley. Ten commandments of formal methods. *IEEE Computer*, 28(4):56–63, April 1995.

Bill Brykczynski. A survey of software inspection checklists. *ACM SIGSOFT Software Engineering Notes*, 24(1):82, 1999.

T.A. Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Department of Computer Science, Yale University, New Haven, CT, 1980.

Man-yee Chan and Shing-chi Cheung. Applying white box testing to database applications. Technical Report HKUST-CS9901, Hong Kong University of Science and Technology, Department of Computer Science, February 1999a.

Man-yee Chan and Shing-chi Cheung. Testing database applications with SQL semantics. In *Proceedings of the 2nd International Symposium on Cooperative Database Systems for Advanced Applications*, pages 363–374, March 1999b.

David Chays, Saikat Dan, Phyllis G. Frankl, Filippos I. Vokolos, and Elaine J. Weyuker. A framework for testing database applications. In *Proceedings of the 7th International Symposium on Software Testing and Analysis*, pages 147–157, August 2000.

David Chays and Yuetang Deng. Demonstration of AGENDA tool set for testing relational database applications. In *Proceedings of the International Conference on Software Engineering*, pages 802–803, May 2003.

David Chays, Yuetang Deng, Phyllis G. Frankl, Saikat Dan, Filippos I. Vokolos, and Elaine J. Weyuker. AGENDA: A test generator for relational database applications. Technical Report TR-CIS-2002-04, Department of Computer and Information Sciences, Polytechnic University, Brooklyn, NY, August 2002.

B. Choi, A. Mathur, and B. Pattison. PMothra: Scheduling mutants for execution on a hypercube. In *Proceedings of the Third ACM SIGSOFT Symposium on Software Testing, Analysis, and Verfi ciation*, pages 58–65, December 1989.

L.A. Clarke. A system to generate test data symbolically. *IEEE Transactions on Software Engineering*, 2(3):215–222, September 1976.

Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A comparison of data fbw path selection criteria. In *Proceedings of the 8th International Conference on Software Engineering*, pages 244–251. IEEE Computer Society Press, 1985.

David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, 13(5):83–87, September 1996.

David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–443, July 1997.

Bassel Daou, Ramzi A. Haraty, and Nash'at Mansour. Regression testing of database applications. In *Proceedings of the 2001 ACM Symposium on Applied Computing*, pages 285–289. ACM Press, 2001.

R.A. DeMillo, D.S. Guindi, W.M. McCracken, A.J. Offutt, and K.N. King. An extended overview of the Mothra software testing environment. In *Proceedings of the ACM SIGSOFT Second Symposium on Software Testing,Analysis, and Verfi ciation*, pages 142–151, July 1988.

R.A. DeMillo, R. J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

R.A. DeMillo and A.J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

Yuetang Deng, Phyllis Frankl, and Zhongqiang Chen. Testing database transaction concurrency. In *Proceedings of the 18th International Conference on Automated Software Engineering*, Montreal, Canada, October 2003.

Edsger W. Dijkstra. The structure of the THE multiprogramming system. *Communications of the ACM*, 11(5): 341–346, 1968.

Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A demand-driven analyzer for data fbw testing at the integration level. In *Proceedings of the 18th International Conference on Software Engineering*, pages 575–584. IEEE Computer Society Press, 1996.

W. Keith Edwards. *Core Jini*. Prentice Hall PTR, Upper Saddle River, NJ, 1999.

Sebastian Elbaum, Alexey G. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 102–112. ACM Press, August 2000.

Sebastian Elbaum, Gregg Rothermel, Satya Kanduri, and Alexey G. Malishevsky. Selecting a cost-effective test case prioritization technique. Technical Report 03-01-01, Department of Computer Science and Engineering, University of Nebraska – Lincoln, January 2003.

Yasuhiro Endo, Zheng Wang, J. Bradley Chen, and Margo Seltzer. Using latency to evaluate interactive system performance. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, pages 185–199. ACM Press, 1996.

D. Epema, M. Livny, R.V. Dantzig, X. Evers, and J. Pruyne. A worldwide fbck of Condors: Load sharing among workstation clusters. *Journal on Future Generations of Computer Systems*, 12(1):53–65, December 1996.

M. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3): 182–211, 1976.

N. Fenton. Software measurement: A necessary scientifi c basis. *IEEE Transactions on Software Engineering*, 20(3): 199–206, March 1994.

Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, 1996.

M. Fisher, D. Jin, G. Rothermel, and M. Burnett. Test reuse in the spreadsheet paradigm. In *Proceedings of the 13th International Symposium on Software Reliability Engineering*, Annapolis, MD, November 2002a.

Marc Fisher, Mingming Cao, Gregg Rothermel, Curtis R. Cook, and Margaret M. Burnett. Automated test case generation for spreadsheets. In *Proceedings of the 24th International Conference on Software Engineering*, pages 141–153. ACM Press, 2002b.

Phyllis G. Frankl and Elaine J. Weyuker. An applicable family of data fbw testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

Pyhllis G. Frankl and Stuart Weiss. An experimental comparison of the effectiveness of branch testing and data fbw testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, August 1993.

Pyhllis G. Frankl and Elaine J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Transactions on Software Engineering*, 19(3):202–213, March 1993.

Eric Freemen, Susanne Hupfer, and Ken Arnold. *JavaSpaces: Principles, Patterns, and Practice*. Addison-Wesley, Inc., Reading, MA, 1999.

Erich Gamma and Kent Beck. JUnit: A cook's tour. 2004. http://www.junit.org/.

Matthew Geller. Test data as an aid in proving program correctness. *Communications of the ACM*, 21(5):368–375, 1978.

Anup K. Ghosh, Matt Schmid, and Frank Hill. Wrapping Windows NT software for robustness. In *Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, June 1999.

Anup K. Ghosh, Matt Schmid, and Viren Shah. Testing the robustness of Windows NT software. In *Proceedings of the Ninth International Symposium on Software Reliability Engineering*, pages 231–235. IEEE Computer Society, November 1998.

Neelam Gupta, Aditya A. Mathur, and Mary Lou Soffa. Automated test data generation using an iterative relaxation method. In *Proceedings of the 5th ACM SIGSOFT Symposium on Foundations of Software Engineering*. ACM Press, November 1998.

Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa. UNA based iterative test data generation and its evaluation. In *Proceedings of the 14th International Conference on Automated Software Engineering*, pages 224–232, October 1999.

Jennifer Haddox, Gregory M. Kapfhammer, and C.C. Michael. An approach for understanding and testing third-party software components. In *48th Reliability and Maintainability Symposium*, January 2002.

Jennifer Haddox, Gregory M. Kapfhammer, C.C. Michael, and Michael Schatz. Testing commercial-off-the-shelf components with software wrappers. In *Proceedings of the 18th International Conference on Testing Computer Software*, Washington, D.C., June 2001.

Dick Hamlet. Foundations of software testing: dependability theory. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 128–139. ACM Press, 1994.

Dick Hamlet and Joe Maybee. *The Engineering of Software*. Addison Wesley, Boston, MA, 2001.

Michael Harder, Jeff Mellen, and Michael D. Ernst. Improving test suites via operational abstraction. In *Proceedings of the 24th International Conference on Software Engineering*, pages 60–71. IEEE Computer Society Press, 2003.

Mary Jean Harrold and Gregg Rothermel. Performing data flow testing on classes. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 154–163. ACM Press, 1994.

Mary Jean Harrold and Gregg Rothermel. Aristotle: A system for research on and developement of program-analysis-based tools. Technical Report OSU-CISRC-3/97-TR17, The Ohio State University, Department of Computer and Information Science, March 1995.

Mary Jean Harrold and Gregg Rothermel. A coherent family of analyzable graphical representations for object-oriented software. Technical Report Technical Report OSU-CISRC-11/96-TR60, Department of Computer and Information Sciences, Ohio State University, November 1996.

Mary Jean Harrold and Mary Lou Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, 1994.

Richard Hightower. *Java Tools for Extreme Programming: Mastering Open Source Tools, Including Ant, JUnit, and Cactus*. John Wiley and Sons, Inc., New York, NY, 2001.

Adele E. Howe, Anneliese von Mayrhauser, and Richard T. Mraz. Test case generation as an AI planning problem. *Automated Software Engineering: An International Journal*, 4(1):77–106, January 1997.

Christopher J. Howell, Gregory M. Kapfhammer, and Robert S. Roos. An examination of the run-time performance of GUI creation frameworks. In *Proceedings of the 2nd ACM SIGAPP International Conference on the Principles and Practice of Programming in Java*, Kilkenny City, Ireland, June 2003.

Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, pages 191–200. IEEE Computer Society Press, 1994.

IEEE. *IEEE Standard Glossary of Software Engineering Terminology*. ANSI/IEEE Std 610.12-1990, 1996.

Daniel Jackson. Lecture 17: Case study: JUnit. 2003. http://ocw.mit.edu/6/6.170/f01/lecture-notes/index.html.

Pankaj Jalote. *Fault Tolerance in Distributed Systems*. PTR Prentice Hall, Upper Saddle River, New Jersey, 1998.

Ronald E. Jeffries. Extreme testing. *Software Testing and Quality Engineering*, March/April 1999.

Jean-Marc Jezequel, Daniel Deveaux, and Yves Le Traon. Reliable objects: Lightweight testing for OO languages. *IEEE Software*, 18(4):76–83, July/August 2001.

Frederick P. Brooks Jr. *The Mythical Man-Month*. Addison-Wesley, Reading, Massachusetts, 1995.

Cem Kaner, Jack Falk, and Hung Quoc Hguyen. *Testing Computer Software*. International Thompson Computer Press, London, UK, 1993.

Gregory M. Kapfhammer. Automatically and transparently distributing the execution of regression test suites. In *Proceedings of the 18th International Conference on Testing Computer Software*, Washinton, D.C., June 2001.

Gregory M. Kapfhammer and Mary Lou Soffa. A family of test adequacy criteria for database-driven applications. In *Proceedings of the 9th European Software Engineering Conference and the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering*. ACM Press, 2003.

Sunwoo Kim, John A. Clark, and John A. McDermid. The rigorous generation of Java mutation operators using HAZOP. In *Proceedings of the 12th International Conference on Software and Systems Engineering and their Applications*, December 1999.

Sunwoo Kim, John A. Clark, and John A. McDermid. Class mutation: Mutation testing for object-oriented programs. In *Proceedings of the Object-Oriented Software Systems, Net.ObjectDays Conference*, October 2000.

Philip Koopman and John DeVale. The exception handling effectiveness of POSIX operating systems. *IEEE Transactions on Software Engineering*, 26(9):837–848, September 2000.

Bogdan Korel. Automated test data generation for programs with procedures. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis*, pages 209–215. ACM Press, 1996.

E.W. Krauser, A.P. Mathur, and V.J. Rego. High performance software testing on SIMD machines. *IEEE Transactions on Software Engineering*, 17(5):403–423, May 1991.

Nathan P. Krop, Philip J. Koopman, and Daniel P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *Proceedings of the 28th Fault Tolerant Computing Symposium*, pages 230–239, June 1998.

Oliver Laitenberger and Colin Atkinson. Generalizing perspective-based inspection to handle object-oriented development artifacts. In *Proceedings of the 21st international conference on Software engineering*, pages 494–503. IEEE Computer Society Press, 1999.

Yu Lei and K.C. Tai. In-parameter-order: A test generation strategy for pairwise testing. In *Proceedings of the High-Assurance Systems Engineering Symposium*, pages 254–261, November 1998.

H. K. N. Leung and L. J. White. Insights into regression testing. In *Proceedings of the International Conference on Software Maintenance*, pages 60–69. IEEE Computer Society Press, October 1989.

Nancy G. Leveson. *Safeware: System safety and computers*. Addison-Wesley Publishing Co., Reading, MA, September 1995.

Yu-Seung Ma, Yong-Rae Kwon, and Jeff Offutt. Inter-class mutation operators for Java. In *Proceedings of the Twelfth International Symposium on Software Reliability Engineering*, November 2002.

Brian Marick. When should a test be automated? In *Proceedings of the 11th International Quality Week Conference*, San Francisco, CA, May, 26–29 1998.

Brian Marick. How to misuse code coverage. In *Proceedings of the 16th Interational Conference on Testing Computer Software*, June 1999.

Atif M. Memon. *A Comprehensive Framework for Testing Graphical User Interfaces*. PhD thesis, University of Pittsburgh, Department of Computer Science, 2001.

Atif M. Memon. GUI testing: Pitfalls and process. *IEEE Computer*, 35(8):90–91, August 2002.

Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Using a goal-driven approach to generate test cases for GUIs. In *Proceedings of the 21st International Conference on Software Engineering*, pages 257–266. IEEE Computer Society Press, 1999.

Atif M. Memon, Martha E. Pollock, and Mary Lou Soffa. Heirarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering*, 27(2):144–155, February 2001a.

Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. Coverage criteria for GUI testing. In *Proceedings of the 8th European Software Engineering Conference and the 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 256–267. ACM Press, 2001b.

Christoph C. Michael, Gary McGraw, and Michael Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, December 2001.

B. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of operating system utilities. *Communications of the ACM*, 33:32–44, December 1990.

B. Miller, D. Koski, C. Lee, V. Maganty, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examiniation of the reliability of UNIX utilities and services. Technical Report 1268, University of Wisconsin-Madison, May 1998.

L. J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, 1990.

Brad A. Myers and Mary Beth Rosson. Survey on user interface programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 195–202. ACM Press, 1992.

Richard Neapolitan and Kumarss Naimipour. *Foundations of Algorithms*. Jones and Bartlett Publishers, Boston, Massachusetts, 1998.

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, Berlin, Germany, 1999.

Simeon C. Ntafos. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, 14(6):868–874, June 1988.

A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of suffi cient mutant operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118, 1996.

A.J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation testing. In *Proceedings of the 15th International Conference on Software Engineering*, pages 100–107, May 1993.

Jeff Offutt, Roger Alexander, Ye Wu, Quansheng Xiao, and Chuck Hutchinson. A fault model for subtype inheritance and polymorphism. In *Proceedings of the Twelfth International Symposium on Software Reliability Engineering*, pages 84–95, November 2001.

Akira K. Onoma, Wei-Tek Tsai, Mustafa Poonawala, and Hiroshi Suganuma. Regression testing in an industrial environment. *Communications of the ACM*, 41(5):81–86, 1998.

T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Communications of the ACM*, 31(6):676–686, 1988.

Allen Parrish and Stuart H. Zweben. Software test data adequacy properties. *IEEE Transactions on Software Engineering*, 17(6):565–581, June 1991.

Adam S. Paul. SAGE: A static metric for testability under the PIE model. Technical Report 96-5, Allegheny College, Department of Computer Science, 1996.

Christina Pavlopoulou and Michal Young. Residual test coverage monitoring. In *Proceedings of the 21st International Conference on Software Engineering*, pages 277–284. IEEE Computer Society Press, 1999.

Bret Pettichord. Seven steps to test automation success. In *Proceedings of the International Conference on Software Testing, Analysis, and Review*, San Jose, CA, November 1999.

C.V. Ramamoorty, S.F. Ho, and W.T. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, 2(4):293–300, December 1976.

Sandra Rapps and Elaine J. Weyuker. Data fbw analysis techniques for test data selection. In *Proceedings of the 6th International Conference on Software Engineering*, pages 272–278. IEEE Computer Society Press, 1982.

Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data fbw information. *IEEE Transactions on Software Engineering*, 11(4), April 1985.

D. Richardson, O. O'Malley, and C. Tittle. Approaches to specifi cation-based testing. In *Proceedings of the ACM SIGSOFT Third Symposium on Software Testing, Analysis, and Verifi cation*, pages 86–96. ACM Press, 1989.

G. Rothermel and M. J. Harrold. A framework for evaluating regression test selection techniques. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 201–210. IEEE Computer Society Press, May 1994.

G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, August 1996.

G. Rothermel and M. J. Harrold. A safe, effi cient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.

G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering*, 24(6):401–419, June 1998.

G. Rothermel, L. Li, and M. Burnett. Testing strategies for form-based visual programs. In *Proceedings of the 8th International Symposium on Software Reliability Engineering*, pages 96–107, Albuquerque, NM, November 1997.

G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *Proceedings of the International Conference on Software Maintenance*, pages 179–188, August 1999.

G. Rothermel, Roland H. Untch, Chengyun Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, October 2001a.

Gregg Rothermel, Margaret Burnett, Lixin Li, Christopher Dupuis, and Andrei Sheretov. A methodology for testing spreadsheets. *ACM Transactions on Software Engineering and Methodology*, 10(1):110–147, January 2001b.

Karen J. Rothermel, Curtis R. Cook, Margaret M. Burnett, Justin Schonfeld, T. R. G. Green, and Gregg Rothermel. WYSIWYT testing in the spreadsheet paradigm: an empirical evaluation. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 230–239. ACM Press, 2000.

Andrew Sears. Layout appropriateness: A metric for evaluating user interface widget layout. *IEEE Transactions on Software Engineering*, 19(7):707–719, 1993.

Forrest Shull, Ioana Rus, and Victor Basili. Improving software inspections by using reading techniques. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 726–727. IEEE Computer Society, 2001.

Ian Sommerville. *Software Engineering*. Addison-Wesley, 6th edition, August 2000.

John Steven, Pravir Chandra, Bob Fleck, and Andy Podgurski. jRapture: A capture/replay tool for observation-based testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 158–167. ACM Press, 2000.

T. Tsai and R. Iyer. Measuring fault tolerance with the FTAPE fault injection tool. In *Proceedings of the 8th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, pages 26–40, 1995.

Timothy K. Tsai and Navjot Singh. Reliability testing of applications on Windows NT. In *Proceedings of the International Conference on Dependable Systems and Networks*, New York City, USA, June 2000.

Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

Jeffrey M. Voas. PIE: a dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18(8):717–735, 1992.

F. Vokolos and P. Frankl. Pythia: A regression test selection tool based on textual differencing. In *Third International Conference of Reliability, Quality, and Safety of Software Intensive Systems*, May 1997.

Elaine Weyuker. Axiomatizing software test data adequacy. *IEEE Transactions on Software Engineering*, (12): 1128–1138, December 1986.

Elaine J. Weyuker, Stewart N. Weiss, and Dick Hamlet. Comparison of program testing strategies. In *Proceedings of the Symposium on Testing, Analysis, and Verifi cation*, pages 1–10. ACM Press, 1991.

James A. Whittaker. What is software testing? and why is it so hard? *IEEE Software*, 17(1):70–76, January/February 2000.

James A. Whittaker and Jeffrey Voas. Toward a more reliable theory of software reliability. *IEEE Computer*, 32(12): 36–42, December 2000.

W.E. Wong. *On Mutation and Data Flow*. PhD thesis, Department of Computer Science, Purdue University, West Lafayette, IN, December 1993.

W.E. Wong, J.R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proceedings of the 8th International Symposium on Software Reliability Engineering*, pages 230–238, November 1997.

Michael Young and Richard N. Taylor. Rethinking the taxonomy of fault detection techniques. In *Proceedings of the 11th International Conference on Software Engineering*, pages 53–62. ACM Press, 1989.

Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.

6 Further Information

Software testing and analysis is an active research area. The ACM/IEEE International Conference on Software Engineering, the ACM SIGSOFT Symposium on the Foundations of Software Engineering, the ACM SIGSOFT International Symposium on Software Testing and Analysis, and the ACM SIGAPP Symposium on Applied Computing's Software Engineering Track are all important forums for new research in the areas of software engineering and software testing and analysis. Other important conferences include: IEEE Automated Software Engineering, IEEE International Conference on Software Maintenance, IEEE International Symposium on Software Reliability Engineering, the IEEE/NASA Software Engineering Workshop, and the IEEE Computer Software and Applications Conference.

There are also several magazines and journals that provide archives for important software engineering and software testing research. The *IEEE Transactions on Software Engineering* and the *ACM Transactions on Software Engineering and Methodology* are two noteworthy journals that often publish software testing papers. Other journals include: *Software Testing, Verification, and Reliability*, *Software: Practice and Experience*, *Software Quality Journal*, *Automated Software Engineering: An International Journal*, and *Empirical Software Engineering: An International Journal*. Magazines that publish software testing articles include *Communications of the ACM*, *IEEE Software*, *IEEE Computer*, and *Better Software* (formerly known as *Software Testing and Quality Engineering*). ACM SIGSOFT also sponsors the bi-monthly newsletter called *Software Engineering Notes*.

7 Acknowledgements

Robert D. Cupper was supportive of the writing of this chapter from its inception. Some of my undergraduate students, including James A. Clause, Richard C. Geary, and Matthew J. Rummel, read several drafts of this chapter and provided insightful comments. Timothy K. Tsai and Jennifer Haddox-Schatz both read an intermediate draft of the chapter and pointed out unclear passages and definitions. Mary Lou Soffa has often helped me to refine my understanding of software testing and indirectly influenced many portions of this chapter. My wife Jessica M. Kapfhammer and my daughter Madelyn M. Kapfhammer provided loving support and encouragement during the writing of this chapter and they continue to demonstrate that they truly are God's gift to me. The most important acknowledgement that I can make begins with the one that was made by Frederick P. Brooks, Jr. in his preface to the 20th anniversary edition of *The Mythical Man-Month*. ***Soli Deo Gloria*** **– To God Alone be Glory.** ***Sola Scriptura*** **– By the Scriptures Alone.** ***Solus Christus*** **– By Christ Alone.** ***Sola Gratia*** **– By Grace Alone.** ***Sola Fide*** **– By Faith Alone.**