FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

Software Testing and Analysis: Process, Principles, and Techniques

https://hemanthrajhemu.github.io

https://hemanthrajhemu.github.io

Chapter 17

Test Execution

Whereas test design, even when supported by tools, requires insight and ingenuity in similar measure to other facets of software design, test execution must be sufficiently automated for frequent reexecution without little human involvement. This chapter describes approaches for creating the run-time support for generating and managing test data, creating scaffolding for test execution, and automatically distinguishing between correct and incorrect test case executions.

Required Background

- Chapter 7
  Reasoning about program correctness is closely related to test oracles that recognize incorrect behavior at run-time.

- Chapters 9 and 10
  Basic concepts introduced in these chapters are essential background for understanding the distinction between designing a test case specification and executing a test case.

- Chapters 11 through 16
  These chapters provide more context and concrete examples for understanding the material presented here.

17.1 Overview

Designing tests is creative; executing them should be as mechanical as compiling the latest version of the product, and indeed a product build is not complete until it has passed a suite of test cases. In many organizations, a complete build-and-test cycle occurs nightly, with a report of success or problems ready each morning.

The purpose of run-time support for testing is to enable frequent hands-free reexecution of a test suite. A large suite of test data may be generated automatically from a

https://hemanthrajhemu.github.io

more compact and abstract set of test case specifications. For unit and integration testing, and sometimes for system testing as well, the software under test may be combined with additional "scaffolding" code to provide a suitable test environment, which might, for example, include simulations of other software and hardware resources. Executing a large number of test cases is of little use unless the observed behaviors are classified as passing or failing. The human eye is a slow, expensive, and unreliable instrument for judging test outcomes, so test scaffolding typically includes automated test oracles. The test environment often includes additional support for selecting test cases (e.g., rotating nightly through portions of a large test suite over the course of a week) and for summarizing and reporting results.

17.2 From Test Case Specifications to Test Cases

If the test case specifications produced in test design already include concrete input values and expected results, as for example in the category-partition method, then producing a complete test case may be as simple as filling a template with those values. A more general test case specification (e.g., one that calls for "a sorted sequence, length greater than 2, with items in ascending order with no duplicates") may designate many possible concrete test cases, and it may be desirable to generate just one instance or many. There is no clear, sharp line between test case design and test case generation. A rule of thumb is that, while test case design involves judgment and creativity, test case generation should be a mechanical step.

Automatic generation of concrete test cases from more abstract test case specifications reduces the impact of small interface changes in the course of development. Corresponding changes to the test suite are still required with each program change, but changes to test case specifications are likely to be smaller and more localized than changes to the concrete test cases.

Instantiating test cases that satisfy several constraints may be simple if the constraints are independent (e.g., a constraint on each of several input parameter values), but becomes more difficult to automate when multiple constraints apply to the same item. Some well-formed sets of constraints have no solution at all ("an even, positive integer that is not the sum of two primes"). Constraints that appear to be independent may not be. For example, a test case specification that constrains both program input and output imposes a conjunction of two constraints on output (it conforms to the given output constraint *and* it is produced by the given input).

General test case specifications that may require considerable computation to produce test data often arise in model-based testing. For example, if a test case calls for program execution corresponding to a certain traversal of transitions in a finite state machine model, the test data must trigger that traversal, which may be quite complex if the model includes computations and semantic constraints (e.g., a protocol model in Promela; see Chapter 8). Fortunately, model-based testing is closely tied to model analysis techniques that can be adapted as test data generation methods. For example, finite state verification techniques typically have facilities for generating counter-examples to asserted properties. If one can express the negation of a test case specification, then treating it as a property to be verified will result in a counter-example from which a

https://hemanthrajhemu.github.io

concrete test case can be generated.

17.3 Scaffolding

During much of development, only a portion of the full system is available for testing. In modern development methodologies, the partially developed system is likely to consist of one or more runnable programs and may even be considered a version or prototype of the final system from very early in construction, so it is possible at least to execute each new portion of the software as it is constructed, but the external interfaces of the evolving system may not be ideal for testing; often additional code must be added. For example, even if the actual subsystem for placing an order with a supplier is available and fully operational, it is probably not desirable to place a thousand supply orders each night as part of an automatic test run. More likely a portion of the order placement software will be "stubbed out" for most test executions.

Code developed to facilitate testing is called *scaffolding*, by analogy to the temporary structures erected around a building during construction or maintenance. Scaffolding may include test drivers (substituting for a main or calling program), test harnesses (substituting for parts of the deployment environment), and stubs (substituting for functionality called or used by the software under test), in addition to program instrumentation and support for recording and managing test execution. A common estimate is that half of the code developed in a software project is scaffolding of some kind, but the amount of scaffolding that must be constructed with a software project can vary widely, and depends both on the application domain and the architectural design and build plan, which can reduce cost by exposing appropriate interfaces and providing necessary functionality in a rational order.

test driver
test harness
stub

The purposes of scaffolding are to provide controllability to execute test cases and observability to judge the outcome of test execution. Sometimes scaffolding is required to simply make a module executable, but even in incremental development with immediate integration of each module, scaffolding for controllability and observability may be required because the external interfaces of the system may not provide sufficient control to drive the module under test through test cases, or sufficient observability of the effect. It may be desirable to substitute a separate test "driver" program for the full system, in order to provide more direct control of an interface or to remove dependence on other subsystems.

Consider, for example, an interactive program that is normally driven through a graphical user interface. Assume that each night the program goes through a fully automated and unattended cycle of integration, compilation, and test execution. It is necessary to perform some testing through the interactive interface, but it is neither necessary nor efficient to execute all test cases that way. Small driver programs, independent of the graphical user interface, can drive each module through large test suites in a short time.

When testability is considered in software architectural design, it often happens that interfaces exposed for use in scaffolding have other uses. For example, the interfaces needed to drive an interactive program without its graphical user interface are likely to serve also as the interface for a scripting facility. A similar phenomenon appears at a

https://hemanthrajhemu.github.io

finer grain. For example, introducing a Java interface to isolate the public functionality of a class and hide methods introduced for testing the implementation has a cost, but also potential side benefits such as making it easier to support multiple implementations of the interface.

17.4 Generic versus Specific Scaffolding

The simplest form of scaffolding is a driver program that runs a single, specific test case. If, for example, a test case specification calls for executing method calls in a particular sequence, this is easy to accomplish by writing the code to make the method calls in that sequence. Writing hundreds or thousands of such test-specific drivers, on the other hand, may be cumbersome and a disincentive to thorough testing. At the very least one will want to factor out some of the common driver code into reusable modules. Sometimes it is worthwhile to write more generic test drivers that essentially interpret test case specifications.

At least some level of generic scaffolding support can be used across a fairly wide class of applications. Such support typically includes, in addition to a standard interface for executing a set of test cases, basic support for logging test execution and results. Figure 17.1 illustrates use of generic test scaffolding in the JFlex lexical analyzer generator.

Fully generic scaffolding may suffice for small numbers of hand-written test cases. For larger test suites, and particularly for those that are generated systematically (e.g., using the combinatorial techniques described in Chapter 11 or deriving test case specifications from a model as described in Chapter 14), writing each test case by hand is impractical. Note, however, that the Java code expressing each test case in Figure 17.1 follows a simple pattern, and it would not be difficult to write a small program to convert a large collection of input, output pairs into procedures following the same pattern. A large suite of automatically generated test cases and a smaller set of hand-written test cases can share the same underlying generic test scaffolding.

Scaffolding to replace portions of the system is somewhat more demanding, and again both generic and application-specific approaches are possible. The simplest kind

mock

of stub, sometimes called a *mock*, can be generated automatically by analysis of the source code. A mock is limited to checking expected invocations and producing precomputed results that are part of the test case specification or were recorded in a prior execution. Depending on system build order and the relation of unit testing to integration in a particular process, isolating the module under test is sometimes considered an advantage of creating mocks, as compared to depending on other parts of the system that have already been constructed.

The balance of quality, scope, and cost for a substantial piece of scaffolding software — say, a network traffic generator for a distributed system or a test harness for a compiler — is essentially similar to the development of any other substantial piece of software, including similar considerations regarding specialization to a single project or investing more effort to construct a component that can be used in several projects.

The balance is altered in favor of simplicity and quick construction for the many small pieces of scaffolding that are typically produced during development to support

https://hemanthrajhemu.github.io

```
public final class IntCharSet {
...
  public void add(Interval intervall) {
...
  }
```

```
package JFlex.tests;

import JFlex.IntCharSet;
import JFlex.Interval;
import junit.framework.TestCase;
...
public class CharClassesTest extends TestCase {
...
  public void testAdd1() {
    IntCharSet set = new IntCharSet(new Interval('a','h'));
    set.add(new Interval('o','z'));
    set.add(new Interval('A','Z'));
    set.add(new Interval('h','o'));
    assertEquals("{ ['A'-'Z']['a'-'z'] }", set.toString());
  }

  public void testAdd2() {
    IntCharSet set = new IntCharSet(new Interval('a','h'));
    set.add(new Interval('o','z'));
    set.add(new Interval('A','Z'));
    set.add(new Interval('i','n'));
    assertEquals("{ ['A'-'Z']['a'-'z'] }", set.toString());
  }
...
}
```

*Figure 17.1: Excerpt of JFlex 1.4.1 source code (a widely used open-source scanner generator) and accompanying JUnit test cases. JUnit is typical of basic test scaffolding libraries, providing support for test execution, logging, and simple result checking (*assertEquals *in the example). The illustrated version of JUnit uses Java reflection to find and execute test case methods; later versions of JUnit use Java annotation (metadata) facilities, and other tools use source code preprocessors or generators.*

https://hemanthrajhemu.github.io

unit and small-scale integration testing. For example, a database query may be replaced by a stub that provides only a fixed set of responses to particular query strings.

17.5 Test Oracles

It is little use to execute a test suite automatically if execution results must be manually inspected to apply a pass/fail criterion. Relying on human intervention to judge test outcomes is not merely expensive, but also unreliable. Even the most conscientious and hard-working person cannot maintain the level of attention required to identify one failure in a hundred program executions, little more one or ten thousand. That is a job for a computer.

Δ test oracle

Software that applies a pass/fail criterion to a program execution is called a *test oracle*, often shortened to *oracle*. In addition to rapidly classifying a large number of test case executions, automated test oracles make it possible to classify behaviors that exceed human capacity in other ways, such as checking real-time response against latency requirements or dealing with voluminous output data in a machine-readable rather than human-readable form.

Ideally, a test oracle would classify every execution of a correct program as passing and would detect every program failure. In practice, the pass/fail criterion is usually imperfect. A test oracle may apply a pass/fail criterion that reflects only part of the actual program specification, or is an approximation, and therefore passes some program executions it ought to fail. Several partial test oracles (perhaps applied with different parts of the test suite) may be more cost-effective than one that is more comprehensive. A test oracle may also give false alarms, failing an execution that it ought to pass. False alarms in test execution are highly undesirable, not only because of the direct expense of manually checking them, but because they make it likely that real failures will be overlooked. Nevertheless sometimes the best we can obtain is an oracle that detects deviations from expectation that may or may not be actual failures.

comparison-based oracle

One approach to judging correctness — but not the only one — compares the actual output or behavior of a program with predicted output or behavior. A test case with a comparison-based oracle relies on predicted output that is either precomputed as part of the test case specification or can be derived in some way independent of the program under test. Precomputing expected test results is reasonable for a small number of relatively simple test cases, and is still preferable to manual inspection of program results because the expense of producing (and debugging) predicted results is incurred once and amortized over many executions of the test case.

Support for comparison-based test oracles is often included in a test harness program or testing framework. A harness typically takes two inputs: (1) the input to the program under test (or can be mechanically transformed to a well-formed input), and (2) the predicted output. Frameworks for writing test cases as program code likewise provide support for comparison-based oracles. The assertEquals method of *JUnit*, illustrated in Figure 17.1, is a simple example of comparison-based oracle support.

Comparison-based oracles are useful mainly for small, simple test cases, but sometimes expected outputs can also be produced for complex test cases and large test suites. Capture-replay testing, a special case of this in which the predicted output or behavior

https://hemanthrajhemu.github.io

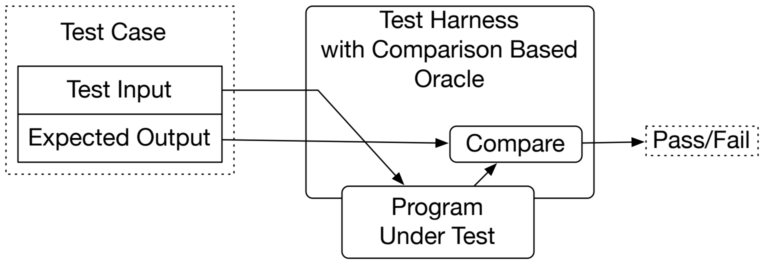

Figure 17.2: A test harness with a comparison-based test oracle processes test cases consisting of (program input, predicted output) pairs.

is preserved from an earlier execution, is discussed in this chapter. A related approach is to capture the output of a trusted alternate version of the program under test. For example, one may produce output from a trusted implementation that is for some reason unsuited for production use; it may too slow or may depend on a component that is not available in the production environment. It is not even necessary that the alternative implementation be *more* reliable than the program under test, as long as it is sufficiently different that the failures of the real and alternate version are likely to be independent, and both are sufficiently reliable that not too much time is wasted determining which one has failed a particular test case on which they disagree.

A third approach to producing complex (input, output) pairs is sometimes possible: It may be easier to produce program input corresponding to a given output than vice versa. For example, it is simpler to scramble a sorted array than to sort a scrambled array.

A common misperception is that a test oracle always requires predicted program output to compare to the output produced in a test execution. In fact, it is often possible to judge output or behavior without predicting it. For example, if a program is required to find a bus route from station A to station B, a test oracle need not independently compute the route to ascertain that it is in fact a valid route that starts at A and ends at B.

Oracles that check results without reference to a predicted output are often partial, in the sense that they can detect some violations of the actual specification but not others. They check necessary but not sufficient conditions for correctness. For example, if the specification calls for finding the optimum bus route according to some metric, a validity check is only a partial oracle because it does not check optimality. Similarly, checking that a sort routine produces sorted output is simple and cheap, but it is only a partial oracle because the output is also required to be a permutation of the input. A cheap partial oracle that can be used for a large number of test cases is often combined with a more expensive comparison-based oracle that can be used with a smaller set of test cases for which predicted output has been obtained.

partial oracle

Ideally, a single expression of a specification would serve both as a work assignment and as a source from which useful test oracles were automatically derived. Spec-

https://hemanthrajhemu.github.io

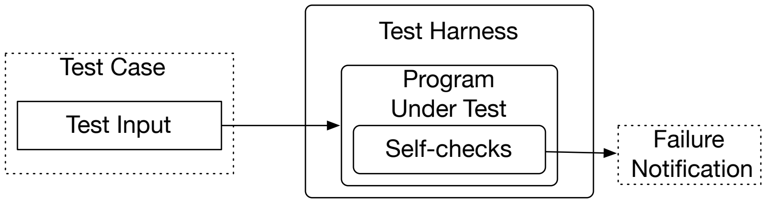

Figure 17.3: When self-checks are embedded in the program, test cases need not include predicted outputs.

ifications are often incomplete, and their informality typically makes automatic derivation of test oracles impossible. The idea is nonetheless a powerful one, and wherever formal or semiformal specifications (including design models) are available, it is worthwhile to consider whether test oracles can be derived from them. Some of the effort of formalization will be incurred either early, in writing specifications, or later when oracles are derived from them, and earlier is usually preferable. Model-based testing, in which test cases and test oracles are both derived from design models are discussed in Chapter 14.

17.6 Self-Checks as Oracles

A program or module specification describes *all* correct program behaviors, so an oracle based on a specification need not be paired with a particular test case. Instead, the oracle can be incorporated into the program under test, so that it checks its own work (see Figure 17.3). Typically these self-checks are in the form of assertions, similar to assertions used in symbolic execution and program verification (see Chapter 7), but designed to be checked during execution.

Self-check assertions may be left in the production version of a system, where they provide much better diagnostic information than the uncontrolled application crash the customer may otherwise report. If this is not acceptable — for instance, if the cost of a runtime assertion check is too high — most tools for assertion processing also provide controls for activating and deactivating assertions. It is generally considered good design practice to make assertions and self-checks be free of side-effects on program state. Side-effect free assertions are essential when assertions may be deactivated, because otherwise suppressing assertion checking can introduce program failures that appear only when one is *not* testing.

Self-checks in the form of assertions embedded in program code are useful primarily for checking module and subsystem-level specifications, rather than overall program behavior. Devising program assertions that correspond in a natural way to specifications (formal or informal) poses two main challenges: bridging the gap between concrete execution values and abstractions used in specification, and dealing in a reasonable way with quantification over collections of values.

Test execution necessarily deals with concrete values, while abstract models are

https://hemanthrajhemu.github.io

indispensable in both formal and informal specifications. Chapter 7 (page 110) describes the role of abstraction functions and structural invariants in specifying concrete operational behavior based on an abstract model of the internal state of a module. The intended effect of an operation is described in terms of a precondition (state before the operation) and postcondition (state after the operation), relating the concrete state to the abstract model. Consider again a specification of the get method of java.util.Map from Chapter 7, with pre- and postconditions expressed as the Hoare triple

$$(\!|\langle \mathrm{k}, \mathrm{v}\rangle \in \phi(\mathrm{dict})|\!)$$
o = dict.get(k)
$$(\!|\mathrm{o} = \mathrm{v}|\!)$$

ϕ is an abstraction function that constructs the abstract model type (sets of key, value pairs) from the concrete data structure. ϕ is a logical association that need not be implemented when reasoning about program correctness. To create a test oracle, it is useful to have an actual implementation of ϕ. For this example, we might implement a special observer method that creates a simple textual representation of the set of (key, value) pairs. Assertions used as test oracles can then correspond directly to the specification. Besides simplifying implementation of oracles by implementing this mapping once and using it in several assertions, structuring test oracles to mirror a correctness argument is rewarded when a later change to the program invalidates some part of that argument (e.g., by changing the treatment of duplicates or using a different data structure in the implementation).

In addition to an abstraction function, reasoning about the correctness of internal structures usually involves structural invariants, that is, properties of the data structure that are preserved by all operations. Structural invariants are good candidates for self checks implemented as assertions. They pertain directly to the concrete data structure implementation, and can be implemented within the module that encapsulates that data structure. For example, if a dictionary structure is implemented as a red-black tree or an AVL tree, the balance property is an invariant of the structure that can be checked by an assertion within the module. Figure 17.4 illustrates an invariant check found in the source code of the Eclipse programming invariant.

There is a natural tension between expressiveness that makes it easier to write and understand specifications, and limits on expressiveness to obtain efficient implementations. It is not much of a stretch to say that programming languages are just formal specification languages in which expressiveness has been purposely limited to ensure that specifications can be executed with predictable and satisfactory performance. An important way in which specifications used for human communication and reasoning about programs are more expressive and less constrained than programming languages is that they freely quantify over collections of values. For example, a specification of database consistency might state that account identifiers are unique; that is, *for all* account records in the database, there *does not exist* another account record with the same identifier.

It is sometimes straightforward to translate quantification in a specification statement into iteration in a program assertion. In fact, some run-time assertion checking

https://hemanthrajhemu.github.io

```
package org.eclipse.jdt.internal.ui.text;
import java.text.CharacterIterator;
import org.eclipse.jface.text.Assert;
/**
* A <code>CharSequence</code> based implementation of
* <code>CharacterIterator</code>.
* @since 3.0
*/
public class SequenceCharacterIterator implements CharacterIterator {
...
        private void invariant() {
                Assert.isTrue(fIndex >= fFirst);
                Assert.isTrue(fIndex <= fLast);
        }
...
        public SequenceCharacterIterator(CharSequence sequence, int first, int last)
            throws IllegalArgumentException {
                if (sequence == null)
                        throw new NullPointerException();
                if (first < 0 || first > last)
                        throw new IllegalArgumentException();
                if (last > sequence.length())
                        throw new IllegalArgumentException();
                fSequence= sequence;
                fFirst= first;
                fLast= last;
                fIndex= first;
                invariant();
        }
...
        public char setIndex(int position) {
                if (position >= getBeginIndex() && position <= getEndIndex())
                        fIndex= position;
                else
                        throw new IllegalArgumentException();

                invariant();
                return current();
        }
...
}
```

Figure 17.4: A structural invariant checked by run-time assertions. Excerpted from the Eclipse programming environment, version 3. © 2000, 2005 IBM Corporation; used under terms of the Eclipse Public License v1.0.

https://hemanthrajhemu.github.io

systems provide quantifiers that are simply interpreted as loops. This approach can work when collections are small and quantifiers are not too deeply nested, particularly in combination with facilities for selectively disabling assertion checking so that the performance cost is incurred only when testing. Treating quantifiers as loops does not scale well to large collections and cannot be applied at all when a specification quantifies over an infinite collection.[1] For example, it is perfectly reasonable for a specification to state that the route found by a trip-planning application is the shortest among all possible routes between two points, but it is not reasonable for the route planning program to check its work by iterating through all possible routes.

The problem of quantification over large sets of values is a variation on the basic problem of program testing, which is that we cannot exhaustively check all program behaviors. Instead, we select a tiny fraction of possible program behaviors or inputs as representatives. The same tactic is applicable to quantification in specifications. If we cannot fully evaluate the specified property, we can at least select some elements to check (though at present we know of no program assertion packages that support sampling of quantifiers). For example, although we cannot afford to enumerate all possible paths between two points in a large map, we may be able to compare to a sample of other paths found by the same procedure. As with test design, good samples require some insight into the problem, such as recognizing that if the shortest path from A to C passes through B, it should be the concatenation of the shortest path from A to B and the shortest path from B to C.

A final implementation problem for self-checks is that asserted properties sometimes involve values that are either not kept in the program at all (so-called ghost variables) or values that have been replaced ("before" values). A specification of noninterference between threads in a concurrent program may use ghost variables to track entry and exit of threads from a critical section. The postcondition of an in-place sort operation will state that the new value is sorted and a permutation of the input value. This permutation relation refers to both the "before" and "after" values of the object to be sorted. A run-time assertion system must manage ghost variables and retained "before" values and must ensure that they have no side-effects outside assertion checking.

17.7 Capture and Replay

Sometimes it is difficult to either devise a precise description of expected behavior or adequately characterize correct behavior for effective self-checks. For example, while many properties of a program with a graphical interface may be specified in a manner suitable for comparison-based or self-check oracles, some properties are likely to require a person to interact with the program and judge its behavior. If one cannot completely avoid human involvement in test case execution, one can at least avoid

[1] It may seem unreasonable for a program specification to quantify over an infinite collection, but in fact it can arise quite naturally when quantifiers are combined with negation. If we say "there is no integer greater than 1 that divides k evenly," we have combined negation with "there exists" to form a statement logically equivalent to universal ("for all") quantification over the integers. We may be clever enough to realize that it suffices to check integers between 2 and $\sqrt{k}$, but that is no longer a direct translation of the specification statement.

https://hemanthrajhemu.github.io

unnecessary repetition of this cost and opportunity for error. The principle is simple. The first time such a test case is executed, the oracle function is carried out by a human, and the interaction sequence is captured. Provided the execution was judged (by the human tester) to be correct, the captured log now forms an (input, predicted output) pair for subsequent automated retesting.

The savings from automated retesting with a captured log depends on how many build-and-test cycles we can continue to use it in, before it is invalidated by some change to the program. Distinguishing between significant and insignificant variations from predicted behavior, in order to prolong the effective lifetime of a captured log, is a major challenge for capture/replay testing. Capturing events at a more abstract level suppresses insignificant changes. For example, if we log only the actual pixels of windows and menus, then changing even a typeface or background color can invalidate an entire suite of execution logs.

Mapping from concrete state to an abstract model of interaction sequences is sometimes possible but is generally quite limited. A more fruitful approach is capturing input and output behavior at multiple levels of abstraction within the implementation. We have noted the usefulness of a layer in which abstract input events (e.g., selection of an object) are captured in place of concrete events (left mouse button depressed with mouse positioned at 235, 718). Typically, there is a similar abstract layer in graphical output, and much of the capture/replay testing can work at this level. Small changes to a program can still invalidate a large number of execution logs, but it is much more likely that an insignificant detail can either be ignored in comparisons or, even better, the abstract input and output can be systematically transformed to reflect the intended change.

Further amplification of the value of a captured log can be obtained by varying the logged events to obtain additional test cases. Creating meaningful and well-formed variations also depends on the abstraction level of the log. For example, it is simpler to vary textual content recorded in a log than to make an equivalent change to a recorded bitmap representation of that text.

Open Research Issues

Tools to generate some kinds of scaffolding from program code have been constructed, as have tools to generate some kinds of test oracles from design and specification documents. Fuller support for creating test scaffolding might bring these together, combining information derivable from program code itself with information from design and specification to create at least test harnesses and oracles. Program transformation and program analysis techniques have advanced quickly in the last decade, suggesting that a higher level of automation than in the past should now be attainable.

Further Reading

Techniques for automatically deriving test oracles from formal specifications have been described for a wide variety of specification notations. Good starting points in this lit-

https://hemanthrajhemu.github.io

erature include Peters and Parnas [PP98] on automatic extraction of test oracles from a specification structured as tables; Gannon et al. [GMH81] and Bernot et al. [BGM91] on derivation of test oracles from algebraic specifications; Doong and Frankl [DF94] on an approach related to algebraic specifications but adapted to object-oriented programs; Bochmann and Petrenko [vBP94] on derivation of test oracles from finite state models, particularly (but not only) for communication protocols; and Richardson et al. [RAO92] on a general approach to deriving test oracles from multiple specification languages, including a form of temporal logic and the Z modeling language.

Rosenblum [Ros95] describes a system for writing test oracles in the form of program assertions and assesses their value. Memon and Soffa [MS03] assesses the impact of test oracles and automation for interactive graphical user interface (GUI) programs. Ostrand et al. [OAFG98] describe capture/replay testing for GUI programs.

Mocks for simulating the environment of a module are described by Saff and Ernst [SE04]. Husted and Massol [HM03] is a guide to the popular JUnit testing framework. Documentation for JUnit and several similar frameworks for various languages and systems are also widely available on the Web.

Related Topics

Readers interested primarily in test automation or in automation of other aspects of analysis and test may wish to continue reading with Chapter 23.

Exercises

17.1. Voluminous output can be a barrier to naive implementations of comparison-based oracles. For example, sometimes we wish to show that some abstraction of program behavior is preserved by a software change. The naive approach is to store a detailed execution log of the original version as predicted output, and compare that to a detailed execution log of the modified version. Unfortunately, a detailed log of a single execution is quite lengthy, and maintaining detailed logs of many test case executions may be impractical. Suggest more efficient approaches to implementing comparison-based test oracles when it is not possible to store the whole output.

17.2. We have described as an ideal but usually unachievable goal that test oracles could be derived automatically from the same specification statement used to record and communicate the intended behavior of a program or module. To what extent does the "test first" approach of extreme programming (XP) achieve this goal? Discuss advantages and limitations of using test cases as a specification statement.

17.3. Often we can choose between on-line self-checks (recognizing failures as they occur) and producing a log of events or states for off-line checking. What considerations might motivate one choice or the other?

https://hemanthrajhemu.github.io

https://hemanthrajhemu.github.io