Chapter 11

Test Selection, Minimization, and Prioritization for Regression Testing

The purpose of this chapter is to introduce techniques for the selection, minimization, and prioritization of tests for regression testing. The source T from which tests are to be selected is likely derived using a combination of black-box and white-box techniques and used for system or component testing. However, when this system or component is modified, for whatever reason, one might be able to retest it using only a subset of T and ensure that despite the changes the existing unchanged code continues to function as desired. A sample of techniques for the selection and prioritization of this subset are presented in this chapter.

The word *regress* means to return to a previous, usually worse, state. Regression testing refers to that portion of the test cycle in which a program P' is tested to ensure that not only does the newly added or modified code behaves correctly, but also that code carried over unchanged from the previous version P, continues to behave correctly. Thus regression testing is useful, and needed, whenever a new version of a program is obtained by modifying an existing version.

Regression testing is sometimes referred to as "program revalidation." The term "corrective regression testing" refers to regression testing of a program obtained by making corrections to the previous versions. Another term "progressive regression testing" refers to regression testing of a program obtained by adding new features. A typical regression testing scenario often includes both corrective and progressive regression testing. In any case, techniques described in this chapter are applicable to both types of regression testing.

To understand the process of regression testing, let us examine a development cycle exhibited in Figure 11.1. The figure shows a highly simplified develop-test-release process for

program P, referred to as Version 1. While P is in use, there might be a need to add new features, remove errors reported by the users, and rewrite some code to improve performance. Such modifications lead to P', referred to as Version 2. This modified version must be tested for any new functionality (step 5 in the figure). However, when making modifications to P the developers might mistakenly add or remove code that causes the existing and unchanged functionality from P to stop behaving as desired. One performs regression testing (step 6) to ensure that any malfunction of the existing code is detected and repaired prior to the release of P'.

Version 1	**Version 2**
1. Develop P	4. Modify P to P′
2. Test P	5. Test P′ for new functionality
3. Release P	6. Perform regression testing on P′ to ensure that the code carried over from P behaves correctly.
	7. Release P′

Figure 11.1: Two phases of product development and maintenance. Version 1 (P) is developed, tested, and released in the first phase. In the next phase, Version 2 (P′) is obtained by modifying Version 1.

It should be obvious from the above description that regression testing can be applied in each phase of software development. For example, during unit testing, when a given unit such as a class, is modified by adding new methods, one needs to perform regression testing to ensure that methods not modified continue to work as required. Certainly, in cases where the developer can prove through suitable arguments that the methods added can have no effect on the existing methods, regression testing is redundant and will likely be not performed.

Regression testing is also needed when a subsystem is modified to generate a new version of an application. When one or more components of an application are modified the entire application must also be subject to regression testing. In some cases regression testing might be needed when the underlying hardware changes. In this case regression testing is performed despite any change in the software.

In the remainder of this chapter you will find various techniques for regression testing. It is important to note that some techniques introduced in this chapter, while sophisticated, might not be applicable in certain environments while absolutely necessary in others. Hence, it is not only important to understand the technique for regression testing but also its strengths and limitations.

11.1. Regression test process

A regression test process is exhibited in Figure 11.2. The process assumes that P' is available for regression testing. There is usually a long series of tasks that lead to P' from P. These tasks, not shown in Figure 11.2, include creation of one or more modification requests and the actual modification of the design and the code. A modification request might lead to a simple error fix, or to a complex redesign and coding of a component of P. In any case, regression testing is recommended after P has been modified and any newly added functionality tested and found correct.

The tasks in Figure 11.2 are shown as if they occur in the given sequence. This is not necessarily true and other sequencings are possible. Several of the tasks shown can be completed while P is being modified to P'. It is important to note that except in some cases, for test selection, all tasks shown in the figure are performed in almost all phases of testing and are not specific to regression testing.

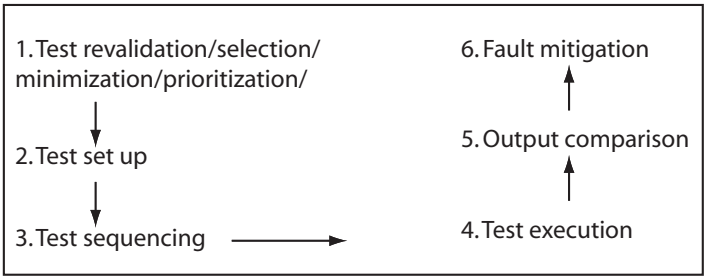

Figure 11.2: A subset of tasks in regression testing.

11.1.1. Test revalidation/selection/minimization/prioritization

While it would be ideal to test P' against all tests developed for P, this might not be possible for several reasons. For example, there might not be sufficient time available to run all tests. Also, some tests for P might become invalid for P' due to one or more reasons such as a change in the input data and its format for one or more features. In yet another scenario, the inputs specified in some tests might remain valid for P' but the expected output might not. These are some reasons that necessitate step 1 in Figure 11.2.

Test revalidation refers to the task of checking which tests for P remain valid for P'. Revalidation is necessary to ensure that only tests that are applicable to P' are used during regression testing.

Test selection can be interpreted in several ways. Validated tests might be redundant in that they do not traverse any of the modified portions in P'. The identification of tests that traverse modified portions of P' is often referred to as test selection and sometimes as the *regression test selection* (RTS) problem. However, note that both test minimization and prioritization described next are also techniques for test selection.

Test minimization discards tests seemingly redundant with respect to some criteria. For example, if t_1 and t_2 test function f in P then one might decide to discard t_2 in favor of t_1. The

purpose of minimization is to reduce the number of tests to execute for regression testing.

Test prioritization refers to the task of prioritizing tests based on some criteria. A set of prioritized tests becomes useful when only a subset of tests can be executed due to resource constraints. Test selection can be achieved by selecting a few tests from a prioritized list. However, several other methods for test selection are available as discussed later in this chapter. Revalidation, followed by selection, minimization, and prioritization is one possible sequence to execute these tasks.

EXAMPLE 11.1. A Web service is a program that can be used by another program over the Web. Consider a Web service named ZC, short for ZipCode. The initial version of ZC provides two services: ZtoC and ZtoA. Service ZtoC inputs a zip code and returns a list of cities and the corresponding state while ZtoA inputs a zip code and returns the corresponding area code. We assume that while the ZipCode service can be used over the Web from wherever an internet connection is available, it serves only the United States.

Let us suppose that ZC has been modified to ZC' as follows. First, a user can select from a list of countries and supply the zip code to obtain the corresponding city in that country. This modification is made only to the ZtoC function while ZtoA remains unchanged. Note that the term "zip code" is not universal. For example, in India, the equivalent term is "pin code" which is 6-digits long as compared to the 5-digit zip code used in the United States. Second, a new service named ZtoT has been added which inputs a country and a zip code and returns the corresponding time zone.

Consider the following two tests (only inputs specified) used for testing ZC.

t_1 : $< service = ZtoC, zip = 47906 >$
t_2 : $< service = ZtoA, zip = 47906 >$

A simple examination of the two tests reveals that test t_1 is not valid for ZC' as it does not list the required country field. Test t_2 is valid as we have made no change to ZtoA. Thus we need to either discard t_1 and replace it by a new test for the modified ZtoC, or simply modify t_1 appropriately. We prefer to modify and hence our validated regression test suite for ZC' is

t_1 : $< country = USA, service = ZtoC, zip = 47906 >$
t_2 : $< service = ZtoA, zip = 47906 >$.

Note that testing ZC' requires additional tests to test the ZtoT service. However, we need only the two tests listed above for regression testing. To keep this example short, we have listed only a few tests for ZC. In practice one would develop a much larger suite of tests for ZC which will then be the source of regression tests for ZC'. ■

11.1.2. Test setup

Test set up refers to the process by which the application under test is placed in its intended, or simulated, environment ready to receive data and able to transfer any desired output information. This process could be as simple as double clicking on the application icon to launch it for testing and as complex as setting up the entire special purpose hardware and monitoring equipment and initializing the environment before the test could begin. Test set up becomes

even more challenging when testing embedded software such as that found in printers, cell phones, Automated Teller Machines, medical devices, and automobile engine controllers.

Note that test set up is not special to regression testing, it is also necessary during other stages of testing such as during integration or system testing. Often test set up requires the use of simulators that allow the replacement of a "real device" to be controlled by the software with its simulated version. For example, a heart simulator is used while testing a commonly used heart control device known as the pacemaker. The simulator allows the pacemaker software to be tested without having to install it inside a human body.

The test set up process and the set up itself, are highly dependent on the application under test and its hardware and software environment. For example, the test set up process and the set up for an automobile engine control software is quite different from that of a cell phone. In the former one needs an engine simulator, or the actual automobile engine to be controlled, while in the latter one needs a test driver that can simulate the constantly changing environment.

11.1.3. Test sequencing

The sequence in which tests are input to an application may or may not be of concern. Test sequencing often becomes important for an application with an internal state and that is continuously running. Banking software, web service, engine controller, are examples of such applications. Sequencing requires grouping and sequencing tests to be run together. The following example illustrates the importance of test sequencing.

EXAMPLE 11.2. Consider a simplified banking application referred to as SATM. Application SATM maintains account balances and offers users the following functionality: login, deposit, withdraw, and exit. Data for each account is maintained in a secure database.

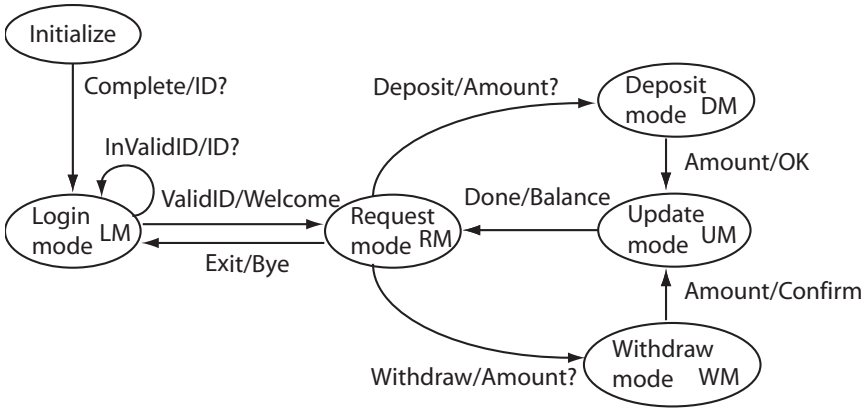

Figure 11.3: State transition in a simplified banking application. Transitions are labeled as X/Y, where X indicates an input and Y the expected output. "Complete" is an internal input indicating that the application moves to the next state upon completion of operations in its current state.

Figure 11.3 exhibits the behavior of SATM as a finite state machine. Note that the machine

has six distinct states, some referred to as modes in the figure. These are labeled as Initialize, LM, RM, DM, UM, and WM. When launched, the SATM performs initialization operations, generates an "ID?" message, and moves to the LM state. If the user enters a valid ID, SATM moves to the RM state else it remains in the LM state and again requests for an ID.

While in the RM state the application expects a service request. Upon receiving a Deposit request it enters the DM state and asks for an amount to be deposited. Upon receiving an amount it generates a confirmatory message and moves to the UM state where it updates the account balance and gets back to the RM state. A similar behavior is shown for the Withdraw request. SATM exits the RM state upon receiving an Exit request.

Let us now consider a set of three tests designed to test the Login, Deposit, Withdraw and Exit features of SATM. The tests are given in the following table in the form of a test matrix. Each test requires that the application be launched fresh and the user (tester in this case) log in. We assume that the user with ID=1 begins with an account balance of 0. Test t_1 checks the login module and the Exit feature, t_2 the Deposit module, and t_3 the Withdraw module. As you might have guessed, these tests are not sufficient for a thorough test of SATM, but they suffice to illustrate the need for test sequencing as explained next.

Test	**Input sequence**	**Expected output sequence**	**Purpose**
t_1	ID=1, Request= Exit	Welcome, Bye	Test Login module.
t_2	ID=1, Request= Deposit, Amount=50	D?, Welcome, Amount?, OK, Done, 50	Test Deposit module.
t_3	ID=1, Request= Withdraw, Amount=30	ID?, Welcome, Amount?, 30, Done, 20	Test Withdraw module.

Now suppose that the Withdraw module has been modified to implement a change in withdrawal policy, e.g. "No more than $300 can be withdrawn on any single day." We now have the modified SATM' to be tested for the new functionality as well as to check if none of the existing functionality has broken. What tests should be rerun ?

Assuming that no other module of SATM has been modified, one might propose that tests t_1 and t_2 need not be rerun. This is a risky proposition unless some formal technique is used to prove that indeed the changes made to the Withdraw module cannot affect the behavior of the remaining modules.

Let us assume that the testers are convinced that the changes in SATM will not affect any module other than Withdraw. Does this mean that we can run only t_3 as a regression test ? The answer is in the negative. Recall our assumption that testing of SATM begins with an account balance of 0 for the user with ID=1. Under this assumption, when run as the first test, t_3 will likely fail because the expected output will not match the output generated by SATM' (see Exercise 11.1).

The argument above leads us to conclude that we need to run test t_3 after having run t_2. Running t_2 ensures that SATM' is brought to the state in which we expect test t_3 to be successful.

Note that the finite state machine shown in Figure 11.3 ignores the values of internal vari-

ables and data bases used by SATM and SATM'. During regression as well as many other types of testing, test sequencing is often necessary to bring the application to a state where the values of internal variables, and contents of the data bases used, correspond to the intention at the time of designing the tests. It is advisable that such intentions (or assumptions) be documented along with each test. ■

11.1.4. Test execution

Once the testing infrastructure has been set up, tests selected, revalidated, and sequenced, it is time to execute them. This task is often automated using a generic or a special purpose tool. General purpose tools are available to run regression tests for applications such as web service (see Section 11.9). However, most embedded systems, due to their unique hardware requirements, often require special purpose tools that input a test suite and automatically run the application against it.

The importance of a tool for test execution cannot be overemphasized. Commercial applications tend to be large and the size of the regression test suite usually increases as new versions arrive. Manual execution of regression tests might become impractical and error prone.

11.1.5. Output comparison

Each test needs verification. This is also done automatically with the help of the test execution tool that compares the generated output with the expected output. However, this might not be a simple process, especially in embedded systems. In such systems often it is the internal state of the application, or the state of the hardware controlled by the software application, that must be checked. This is one reason why generic tools that offer an oracle might not be appropriate for test verification.

One of the goals for test execution is to measure an application's performance. For example, one might want to know how many requests per second can be processed by a web service. In this case performance, and not functional correctness, is of interest. The test execution tool must have special features to allow such measurements.

11.2. Regression test selection: the problem

Let us examine the regression testing problem with respect to Figure 11.4. Let P denote Version X that has been tested using test set T against specification S. Let P' be generated by modifying P. The behavior of P' must conform to specification S'. Specifications S and S' could be the same and P' is the result of modifying P to remove faults. S' could also be different from S in that S' contains all features in S and a few more, or that one of the features in S has been redefined in S'.

The regression testing problem is to find a test set T_r on which P' is to be tested to ensure that code that implements functionality carried over from P works correctly. As shown in Figure 11.4, often T_r is a subset of T used for testing P.

In addition to the regression testing, P' must also be tested to ensure that the newly added code behaves correctly. This is done using a newly developed test set T_d. Thus P' is tested

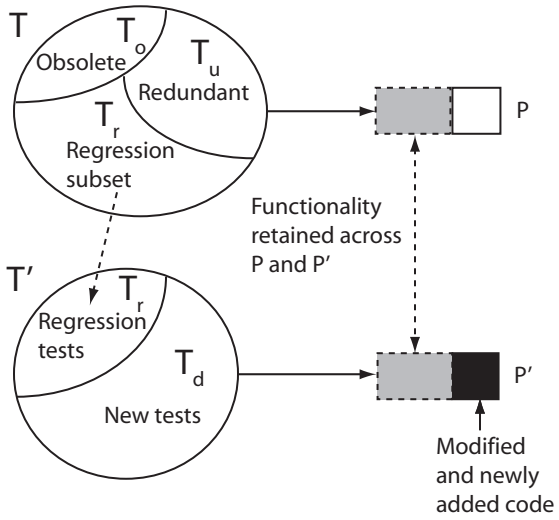

Figure 11.4: Regressing testing as a test selection problem. A subset T_r of set T is selected for retesting the functionality of P that remains unchanged in P′.

against $T' = T_r \cup T_d$ where T_r is the regression test suite and T_d the development test suite intended to test any new functionality in P'. Note that we have subdivided T into three categories: redundant tests (T_u), obsolete tests (T_o), and regression tests (T_r). While P is executed against the entire T, P' is executed only against the regression set T_r and the development set T_d. Tests in T that cause P to terminate prematurely or enter into an infinite loop, might be included in T_o or in T_r depending on their purpose.

In summary, the regression test selection problem (RTS) problem is stated as follows: Find a minimal T_r such that $\forall\, t \in T_r$ and $t' \in T_u \cup T_r, P(t) = P'(t) \Rightarrow P(t') = P'(t')$. In other words the RTS problem is to find a minimal subset T_r of non-obsolete tests from T such that if P' passes tests in T_r then it will also pass tests in T_u. Notice that determination of T_r requires that we know the set of obsolete tests T_o. Obsolete tests are those no longer valid for P' for some reason.

Identification of obsolete tests is a largely manual activity. As mentioned earlier, this activity is often referred to as *test case revalidation*. A test case valid for P might be invalid for P' because the input, output, or both input and output components of the test are rendered invalid by the modification made to P. Such a test case either becomes obsolete and is discarded while testing P' or is corrected and becomes a part of T_r or T_u.

Note that the notion of "correctness" in the above discussion is with respect to the correct functional behavior. It is possible that a solution to the RTS problem ignores a test case t on which P' fails to meet its performance requirement whereas P does. Algorithms for test selection in the remainder of this chapter ignore the performance requirement.