be too narrow, to avoid useless waste of resources, nor too wide, to avoid scattering resources without obtaining useful data. Focusing on specific interactions is usually more effective than attempting to assess usability of a whole program at once. For example, the Chipmunk usability test team independently assesses interations for catalog browsing, order definition and purchase, and repair service.

The larger the population sample, the more precise the results, but the cost of very large samples is prohibitive; selecting a small but representative sample is therefore critical. A good practice is to identify homogeneous classes of users and select a set of representatives from each class. Classes of users depend on the kind of application to be tested, and may be categorized by role, social characteristics, age, etc. A typical compromise between cost and accuracy for a well designed test session is five users from a unique class of homogeneous users, four users from each of two classes, or three users for each of three or more classes. Questionnaires should be prepared for the selected users to verify their membership in their respective classes. Some approaches also assign a weight to each class, according to their importance to the business. For example, Chipmunk can identify three main classes of users: individual, business, and education customers. Each of the main classes is further divided. Individual customers are distinguished by education level; business customers by role; and academic customers by size of the institution. Altogether, six putatively homogenous classes are obtained: Individual customers with and without at least a bachelor degree, managers and staff of commercial customers, and customers at small and large education institutions.

Users are asked to execute a planned set of actions that are identified as typical uses of the tested feature. For example, the Chipmunk usability assessment team may may ask users to configure a product, modify the configuration to take advantage of some special offers, and place an order with overnight delivery.

Users should perform tasks independently, without help or influence from the testing staff. User actions are recorded, and comments and impressions are collected with a post-activity questionnaire. Activity monitoring can be very simple, e.g., recording sequences of mouse clicks to perform each action. More sophisticated monitoring can include recording mouse or eye movements. Timing should also be recorded and may sometimes be used for driving the sessions, e.g., fixing a maximum time for the session or for each set of actions.

An important aspect of usability is accessibility to all users, including those with disabilities. Accessibility testing is legally required in some application domains, e.g., some governments impose specific accessibility rules for web applications of public institutions. The set of Web Content Accessibility Guidelines (WCAG) defined by the World Wide Web Consortium are becoming an important standard reference. The WCAG guidelines are summarized in the sidebar on page 428.

22.5 Regression Testing

When building a new version of a system, e.g., by removing faults, changing or adding functionality, porting the system to a new platform, or extending interoperability, we may also change existing functionality in unintended ways. Sometimes even small

Web Content Accessibility Guidelines (WCAG)[a]

1. Provide equivalent alternatives to auditory and visual content that convey essentially the same function or purpose.
2. Ensure that text and graphics are understandable when viewed without color.
3. Mark up documents with the proper structural elements, controlling presentation with style sheets rather than presentation elements and attributes.
4. Use markup that facilitates pronunciation or interpretation of abbreviated or foreign text.
5. Ensure that tables have necessary markup to be transformed by accessible browsers and other user agents.
6. Ensure that pages are accessible even when newer technologies are not supported or are turned off.
7. Ensure that moving, blinking, scrolling, or auto-updating objects or pages may be paused or stopped.
8. Ensure that the user interface, including embedded user interface elements, follows principles of accessible design: device-independent access to functionality, keyboard operability, self-voicing, etc.
9. Use features that enable activation of page elements via a variety of input devices.
10. Use interim accessibility so that assisting technologies and older browsers will operate correctly.
11. Where technologies outside of W3C specifications is used (e.g, Flash), provide alternative versions to ensure accessibility to standard user agents and assistive technologies (e.g., screen readers).
12. Provide context and orientation information to help users understand complex pages or elements.
13. Provide clear and consistent navigation mechanisms to increase the likelihood that a person will find what they are looking for at a site.
14. Ensure that documents are clear and simple, so they may be more easily understood.

[a]Excerpted and adapted from *Web Content Accessibility Guidelines 1.0*, W3C Recommendation 5-May 1999; used by permission. The current version is distributed by W3C at `http://www.w3.org/TR/WAI-WEBCONTENT`.

changes can produce unforeseen effects that lead to new failures. For example, a guard added to an array to fix an overflow problem may cause a failure when the array is used in other contexts, or porting the software to a new platform may expose a latent fault in creating and modifying temporary files.

When a new version of software no longer correctly provides functionality that should be preserved, we say that the new version *regresses* with respect to former versions. The *non-regression* of new versions, i.e., the preservation of functionality, is a basic quality requirement. Disciplined design and development techniques, including precise specification and modularity that encapsulates independent design decisions, improves the likelihood of achieving non-regression. Testing activities that focus on regression problems are called *(non) regression testing*. Usually "non" is omitted and we commonly say *regression testing*.

A simple approach to regression testing consists of re-executing all test cases designed for previous versions. Even this simple *retest all* approach may present non-trivial problems and costs: Former test cases may not be re-executable on the new version without modification, and re-running all test cases may be too expensive and unnecessary. A good quality test suite must be maintained across system versions.

retest all

Changes in the new software version may impact the format of inputs and outputs, and test cases may not be executable without corresponding changes. Even simple modifications of the data structures, e.g., the addition of a field or small change of data types, may invalidate former test cases, or outputs comparable with the new ones. Moreover, some test cases may be *obsolete*, i.e., they may test features of the software that have been modified, substituted, or removed from the new version.

test case!maintenance

Scaffolding that interprets test case specifications, rather than fully concrete test data, can reduce the impact of input and output format changes on regression testing, as discussed in Chapter 17. Test case specifications and oracles that capture essential correctness properties, abstracting from arbitrary details of behavior, likewise reduce the likelihood that a large portion of a regression test suite will be invalidated by a minor change.

High-quality test suites can be maintained across versions by identifying and removing obsolete test cases, and revealing and suitably marking redundant test cases. Redundant cases differ from obsolete, being executable, but not important with respect to the considered testing criteria. For example, test cases that cover the same path are mutually redundant with respect to structural criteria, while test cases that match the same partition are mutually redundant with respect to functional criteria. Redundant test cases may be introduced in the test suites due to concurrent work of different test designers or to changes in the code. Redundant test cases do not reduce the overall effectiveness of tests, but impact on the cost-benefits trade-off: they are unlikely to reveal faults, but augment the costs of test execution and maintenance. Obsolete test cases are removed because not useful any more, while redundant test cases are kept, because that may become helpful in successive versions of the software.

Good test documentations is particularly important. As we will see in Chapter 24, test specifications define the features to be tested, the corresponding test cases, the inputs and expected outputs as well as the execution conditions for all cases, while reporting documents indicate the results of the test executions, the open faults and their relation to the test cases. This information is essential for tracking faults and for identifying test cases to be re-executed after fault removal.

22.6 Regression Test Selection Techniques

Even when we can identify and eliminate obsolete test cases, the number of tests to be re-executed may be large, especially for legacy software. Executing all test cases for large software products may require many hours or days of execution, and may depend on scarce resources such as an expensive hardware test harness. For example, some mass market software systems must be tested for compatibility with hundreds of different hardware configurations and thousands of drivers. Many test cases may have been designed to exercise parts of the software that cannot be affected by the changes in the version under test. Test cases designed to check the behavior of the file management system of an operating system is unlikely to provide useful information when re-executed after changes of the window manager. The cost of re-executing a test suite can be reduced by selecting a subset of test cases to be re-executed, omitting irrelevant test cases or prioritizing execution of subsets of the test suite by their relation to changes.

Test case prioritization orders frequency of test case execution, executing all of them eventually but reducing the frequency of those deemed least likely to reveal faults by some criterion. Alternate execution is a variant on prioritization for environments with frequent releases and small incremental changes; it selects a subset of regression test cases for each software version. Prioritization can be based on the specification and code-based regression test selection techniques described below. In addition, test histories and fault-proneness models can be incorporated in prioritization schemes. For example, a test case that has previously revealed a fault in a module that has recently undergone change would receive a very high priority, while a test case that has never failed (yet) would receive a lower priority, particularly if it primarily concerns a feature that was not the focus of recent changes.

Regression test selection techniques are based on either code or specifications. Code-based selection techniques select a test case for execution if it exercises a portion of the code that has been modified. Specification-based criteria select a test case for execution if it is relevant to a portion of the specification that has been changed. Code-based regression test techniques can be supported by relatively simple tools. They work even when specifications are not properly maintained. However, like code-based test techniques in general, they do not scale well from unit testing to integration and system testing. In contrast, specification based criteria scale well and are easier to apply to changes that cut across several modules. However, they are more challenging to automate and require carefully structured and well-maintained specifications.

Among code-based test selection techniques, control-based techniques rely on a record of program elements executed by each test case, which may be gathered from an instrumented version of the program. The structure of the new and old versions of the program are compared, and test cases that exercise added, modified, or deleted elements are selected for re-execution. Different criteria are obtained depending on the program model on which the version comparison is based, e.g., control flow or data flow graph models.

Control flow (CFG) regression techniques are based on the differences between the control flow graphs of the new and old versions of the software. Let us consider, for example, the C function cgi_decode from Chapter 12. Figure 22.1 shows the original function as presented in Chapter 12, while Figure 22.2 shows a revison of the program. We refer to these two versions as 1.0 and 2.0, respectively. Version 2.0 adds code to fix a fault in interpreting hexadecimal sequences `'%xy'`. The fault was revealed by testing version 1.0 with input terminated by an erroneous subsequence `'%x'`, causing version 1.0 to read past the end of the input buffer and possibly overflow the output buffer. Version 2.0 contains a new branch to map the unterminated sequence to a question mark.

control flow (CFG) regression test

Let us consider all structural test cases derived for cgi_decode in Chapter 12, and assume we have recorded the paths exercised by the different test cases as shown in Figure 22.3. Recording paths executed by test cases can be done automatically with modest space and time overhead, since what must be captured is only the set of program elements exercised rather than the full history.

CFG regression testing techniques compare annotated control flow graphs of the two program versions to identify a subset of test cases that traverse modified parts of the graphs. The graph nodes are annotated with corresponding program statements, so that comparison of the annotated CFGs detect not only new or missing nodes and arcs, but also nodes whose changed annotations correspond to small, but possibly relevant changes in statements.

The CFG for version 2.0 of cgi_decode is given in Figure 22.4. Differences between version 2.0 and 1.0 are indicated in grey. In the example, we have new nodes, arcs and paths. In general, some nodes or arcs may be missing, e.g., when part of the program is removed in the new version, and some other nodes may differ only in the annotations, e.g., when we modify a condition in the new version.

CFG criteria select all test cases that exercise paths through changed portions of the CFG, including CFG structure changes and node annotations. In the example, we would select all test cases that pass through node D and proceed towards node G and all test cases that reach node L, i.e., all test cases except $TC1$. In this example, the criterion is not very effective in reducing the size of the test suite because modified statements affect almost all paths.

If we consider only the corrective modification (nodes X and Y), the criterion is more effective. The modification affects only the paths that traverse the edge between D and G, so the CFG regression testing criterion would select only test cases traversing those nodes, i.e., $TC2$, $TC3$, $TC4$, $TC5$, $TC8$ and $TC9$. In this case the size of the test suite to be reexecuted includes $\frac{2}{3}$ of the test cases of the original test suite.

In general the CFG regression testing criterion is effective only when the changes affect a relatively small subset of the paths of the original program, as in the latter case. It becomes almost useless when the changes affect most paths, as in version 2.0.

Data flow (DF) regression testing techniques select test cases for new and modified pairs of definitions with uses (DU pairs, cf. Sections 6.1, page 77 and 13.2, page 238). DF regression selection techniques reexecute test cases that, when executed on the original program, exercised DU pairs that are deleted or modified in the revised program. Test cases that executed a conditional statement whose predicate has been altered are also selected, since the changed predicate could alter some old definition

data flow (DF) regression test

```
#include "hex_values.h"
/**  Translate a string from the CGI encoding to plain ascii text.
 *    '+' becomes space, %xx becomes byte with hex value xx,
 *    other alphanumeric characters map to themselves.
 *   Returns 0 for success, positive for erroneous input
 *        1 = bad hexadecimal digit
 */
int cgi_decode(char *encoded, char *decoded) {
  char *eptr = encoded;
  char *dptr = decoded;
  int ok=0;
  while (*eptr) {
    char c;
    c = *eptr;
    if (c == '+') {          /* Case 1: '+' maps to blank */
      *dptr = ' ';
    } else if (c == '%') { /* Case 2: '%xx' is hex for character xx */
      int  digit_high = Hex_Values[*(++eptr)]; /* note illegal => -1 */
      int  digit_low  = Hex_Values[*(++eptr)];
      if ( digit_high == -1 || digit_low == -1 ) {
        /* *dptr='?'; */
        ok=1; /* Bad return code */
      } else {
        *dptr = 16* digit_high + digit_low;
      }
    } else {  /* Case 3: Other characters map to themselves */
      *dptr = *eptr;
    }
    ++dptr;
    ++eptr;
  }
  *dptr = '\0';                    /* Null terminator for string */
  return ok;
}
```

Figure 22.1: C function cgi_decode version 1.0. The C function cgi_decode translates a cgi-encoded string to a plain ASCII string, reversing the encoding applied by the common gateway interface of most web servers. Repeated from Figure 12.1 in Chapter 12 at page 215

```
#include "hex_values.h"
/**   Translate a string from the CGI encoding to plain ascii text.
 *      '+' becomes space, %xx becomes byte with hex value xx,
 *      other alphanumeric characters map to themselves, illegal to '?'.
 *     Returns 0 for success, positive for erroneous input
 *            1 = bad hex digit, non-ascii char, or premature end.
 */
int cgi_decode(char *encoded, char *decoded) {
  char *eptr = encoded;
  char *dptr = decoded;
  int ok=0;
  while (*eptr) {
    char c;
    c = *eptr;
    if (c == '+') {          /* Case 1: '+' maps to blank */
      *dptr = ' ';
    } else if (c == '%') { /* Case 2: '%xx' is hex for character xx */
      if (! ( *(eptr + 1)   && *(eptr + 2) )) {   /* \%xx must precede EOL */
        ok = 1; return;
      }
      /* OK, we know the xx are there, now decode them */
      int   digit_high = Hex_Values[*(++eptr)]; /* note illegal => -1 */
      int   digit_low   = Hex_Values[*(++eptr)];
      if (     digit_high == -1 || digit_low == -1 ) {
        /* *dptr='?'; */
        ok=1; /* Bad return code */
      } else {
        *dptr = 16* digit_high + digit_low;
      }
    } else {   /* Case 3: Other characters map to themselves */
      *dptr = *eptr;
    }
    if (! isascii(*dptr)) { /* Produce only legal ascii */
      *dptr = '?';
      ok = 1;
    }
    ++dptr;
    ++eptr;
  }
  *dptr = '\0';        /* Null terminator for string */
  return ok;
}
```

Figure 22.2: Version 2.0 of function cgi_decode adds a control on hexadecimal escape sequences to reveal incorrect escape sequences at the end of the input string, and a new branch to deal with non-ASCII characters.

Id	Test case	Path
TC1	“ ”	A B M
TC2	“test+case%1Dadequacy”	A B C D F L ... B M
TC3	“adequate+test%0Dexecution%7U”	A B C D F L ... B M
TC4	“%3D”	A B C D G H L B M
TC5	“%A”	A B C D G I L B M
TC6	“a+b”	A B C D F L B C E L B C D F L B M
TC7	“test”	A B C D F L B C D F L B C D F L B C D F L B M
TC8	“+%0D+%4J”	A B C E L B C D G I L ... B M
TC9	“first+test%9Ktest%K9”	A B C D F L ... B M

Figure 22.3: Paths covered by the structural test cases derived for version 1.0 of function cgi_decode. Paths are given referring to the nodes of the control flow graph of Figure 22.4.

use associations. Figure 22.5 shows the new definitions and uses introduced by modifications to cgi_decode.[1] These new definitions and uses introduce new DU pairs and remove others.

In contrast to code-based techniques, specification-based test selection techniques do not require recording the control flow paths executed by tests. Regression test cases can be identified from correspondence between test cases and specification items. For example, when using category partition, test cases correspond to sets of sets of choices, while in finite state machine model-based approaches, test cases cover states and transitions. Where test case specifications and test data are generated automatically from a specification or model, generation can simply be repeated each time the specification or model changes.

Code-based regression test selection criteria can be adapted for model-based regression test selection. Consider for example the control flow graph derived from the *process shipping order* specification in Chapter 11. We add the following item to that specification:

Restricted countries: A set of restricted destination countries is maintained, based on current trade restrictions. If shipping address contains a restricted destination country, only credit card payments are accepted for that order, and shipping proceeds only after approval by a designated company officer responsible for checking that the goods ordered may be legally exported to that country.

The new requirement can be added to the flow graph model of the specification as illustrated in Figure 22.6

[1]When dealing with arrays, we follow the criteria discussed in Chapter 13: A change of an array value is a definition of the array and a use of the index. A use of an array value is a use of both the array and the index.

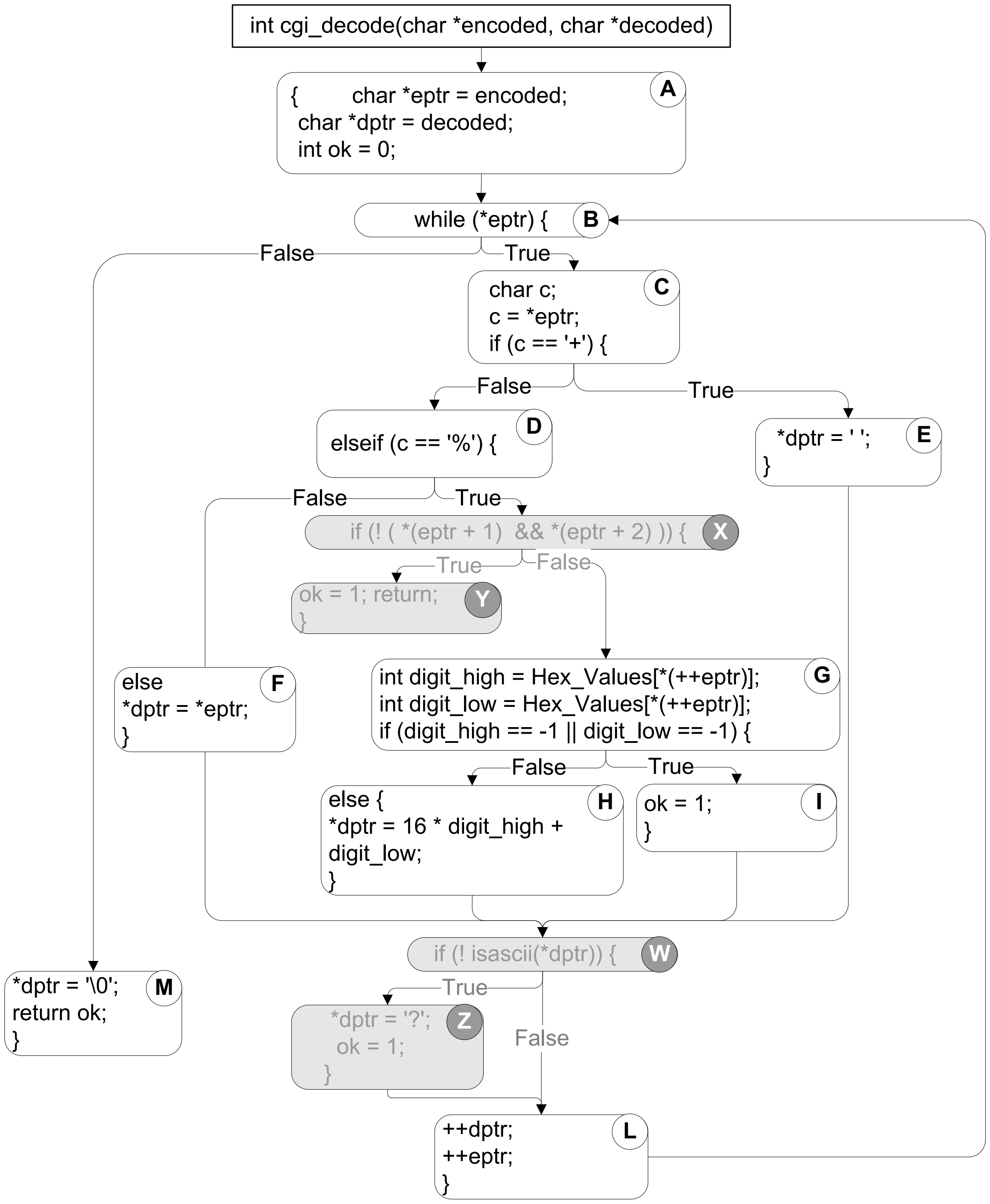

Figure 22.4: The control flow graph of function cgi_decode version 2.0. Grey background indicates the changes from the former version

Variable	Definitions	Uses
*eptr		X
eptr		X
dptr	Z	W
dptr		Z W
ok	Y Z	

Figure 22.5: Definitions and uses introduced introduced by changes in cgi_decode. Labels refer to the nodes in the control flow graph of Figure 22.4

We can identify regression test cases with the CFG criterion that selects all cases that correspond to international shipping addresses, i.e., test cases *TC-1* and *TC-5* from the table below. The table corresponds to the functional test cases derived using to the method described in Chapter 14 on page 261.

Case	Too small	Ship where	Ship method	Cust type	Pay method	Same addr	CC valid
TC-1	No	Int	Air	Bus	CC	No	Yes
TC-2	No	Dom	Land	–	–	–	–
TC-3	Yes	–	–	–	–	–	–
TC-4	No	Dom	Air	–	–	–	–
TC-5	No	Int	Land	–	–	–	–
TC-6	No	–	–	Edu	Inv	–	–
TC-7	No	–	–	–	CC	Yes	–
TC-8	No	–	–	–	CC	–	No (abort)
TC-9	No	–	–	–	CC	–	No (no abort)

Models derived for testing can be used not only for selecting regression test cases, but also for generating test cases for the new code. In the example above, we can use the model not only to identify the test cases that should be reused, but also to generate new test cases for the new functionality, following the approaches described in Chapter 11.

22.7 Test Case Prioritization and Selective Execution

Regression testing criteria may select a large portion of a test suite. When a regression test suite is too large, we must further reduce the set of test cases to be executed.

Random sampling is a simple way to reduce the size of the regression test suite. Better approaches prioritize test cases to reflect their predicted usefulness. In a continuous cycle of retesting as the product evolves, high priority test cases are selected more often than low priority test cases. With a good selection strategy, all test cases are executed sooner or later, but the varying periods result in an efficient rotation in which the cases most likely to reveal faults are executed most frequently.

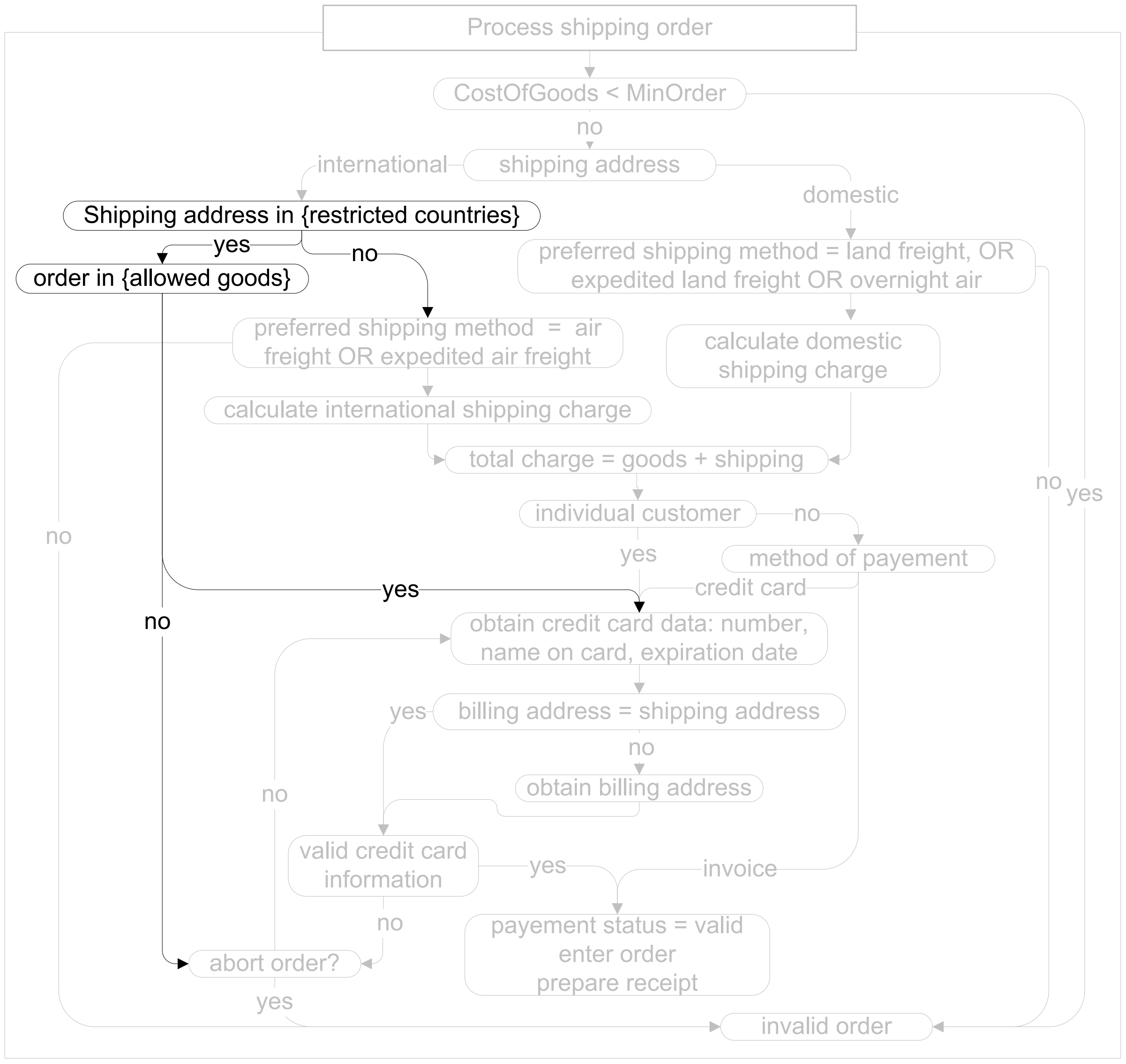
Process shipping order
CostOfGoods < MinOrder
no
international
shipping address
domestic
Shipping address in {restricted countries}
yes
no
order in {allowed goods}
preferred shipping method = land freight, OR expedited land freight OR overnight air
preferred shipping method = air freight OR expedited air freight
calculate domestic shipping charge
calculate international shipping charge
total charge = goods + shipping
no
yes
individual customer
no
no
yes
method of payement
yes
credit card
no
obtain credit card data: number, name on card, expiration date
yes
billing address = shipping address
no
obtain billing address
no
valid credit card information
yes
invoice
no
payement status = valid enter order prepare receipt
abort order?
yes
invalid order

Figure 22.6: A flow graph model of the specification of the shipping order functionality presented in Chapter 11, augmented with the “restricted country” requirement. The changes in the flow graph are indicated in black

Priorities can be assigned in many ways. A simple priority scheme assigns priority according to the execution history: Recently executed test cases are given low priority, while test cases that have not been recently executed are given high priority. In the extreme, heavily weighting execution history approximates round robin selection.

execution history priority schema

Other history-based priority schemes predict fault detection effectiveness: Test cases that have revealed faults in recent versions are given high priority. Faults are not evenly distributed, but tend to accumulate in particular parts of the code or around particular functionality. Test cases that exercised faulty parts of the program in the past often exercise faulty portions of subsequent revisions.

fault revealing priority schema

Structural coverage leads to a set of priority schemes based on the elements covered by a test case. We can give high priority to test cases that exercise elements that have not recently been exercised. Both the number of elements covered, and the "age" of each element (time since that element was covered by a test case) can contribute to the prioritization.

structural priority schema

Structural priority schemes produce several criteria depending on which elements we consider: statements, conditions, decisions, functions, files, etc. The choice of the element of interest is usually driven by the testing level. Fine grain elements such as statements and conditions are typically used in unit testing, while in integration or system testing one can consider coarser grain elements such as methods, features, files, etc.

Open research issues

System requirements include many non-functional behavioral properties. While there is an active research community in reliability testing, in general assessment of non-functional properties is not as well-studied as testing for correctness. Moreover, as trends in software develop, new problems for test and analysis follow emphasis on particular non-functional properties. A prominent example of this over the last several years, and with much left to do, is test and analysis to assess and improve security.

Selective regression test selection based on analysis of source code is now well-studied. There remains need and opportunity for improvement in techniques that give up the safety guarantee (selecting all test cases that *might* be affected by a software change) to obtain more significant test suite reductions. Specification-based regression test selection is a promising avenue of research, particularly as more systems incorporate components without full source code.

Increasingly ubiquitous network access is blurring the once-clear lines between alpha and beta testing and opening possibilities for gathering much more information from execution of deployed software. We expect to see advances in approaches to gathering information (both from failures and from normal execution) as well as exploiting potentially large amounts of gathered information. Privacy and confidentiality are an important research challenge in post-deployment monitoring.