<ch3 toc ch5>

Chapter 4
TRANSACTION-FLOW TESTING

1. SYNOPSIS

Transaction flows are introduced as a representation of a system's processing. The methods that were applied to control flowgraphs are then used for functional testing. Transaction flows and transaction-flow testing are to the independent system tester what control flows and path testing are to the programmer.

2. GENERALIZATIONS

The control flowgraph discussed in Chapter 3 is the most often used model in test design. It is but one of an infinite number of possible models based on the same components—links and nodes. The control flowgraph was introduced as a structural model. We now use the same conceptual components and methods over a different kind of flowgraph, the **transaction flowgraph**—this time, though, to create a behavioral model of the program that leads to functional testing. The transaction flowgraph is, if you will, a model of the structure of the system's behavior (i.e., functionality).

We can either focus on how software is built (i.e., structure) or on how it behaves (i.e., function). A structural focus leads us to structural test techniques, whereas a functional (behavioral) focus leads us to functional test methods. In either case, for either point of view, the **graph**—that is, a representation based on circles (nodes) and arrows (links)—is a powerful conceptual tool. There are many different ways to represent software. Peters (PETE76) summarizes the most useful ones. Most of these representations can be converted to some kind of flowgraph.

Path testing (Chapter 3) is fundamental because we'll now see how the entire elaborate mechanism of path testing can be used in an analogous form as a basis for system testing. The point of all this is to demonstrate that testing consists of defining useful graph models and covering them.

Question—What do you do when you see a graph?

Answer—COVER IT!

3. TRANSACTION FLOWS

3.1. Definitions

A **transaction** is a unit of work seen from a system user's point of view. A transaction consists of a sequence of operations, some of which are performed by a system, persons, or devices that are outside of the system. Transactions begin with **birth—**that is, they are created as a result of some external act. At the conclusion of the transaction's processing, the transaction is no longer in the system, except perhaps in the form of historical records. A transaction for an online information retrieval system might consist of the following steps or **tasks:**

1. Accept input (tentative birth).
2. Validate input (birth).

3. Transmit acknowledgment to requester.
4. Do input processing.
5. Search file.
6. Request directions from user.
7. Accept input.
8. Validate input.
9. Process request.
10. Update file.
11. Transmit output.
12. Record transaction in log and cleanup (death).

The user sees this scenario as a single transaction. From the system's point of view, the transaction consists of twelve steps and ten different kinds of subsidiary tasks.

Most online systems process many kinds of transactions. For example, an automatic bank teller machine can be used for withdrawals, deposits, bill payments, and money transfers. Furthermore, these operations can be done for a checking account, savings account, vacation account, Christmas club, and so on. Although the sequence of operations may differ from transaction to transaction, most transactions have common operations. For example, the automatic teller machine begins every transaction by validating the user's card and password number. Tasks in a transaction flowgraph correspond to processing steps in a control flowgraph. As with control flows, there can be conditional and unconditional branches, and junctions.

3.2. Example

Figure 4.1 shows part of a transaction flow. A PC is used as a terminal controller for several dumb terminals. The terminals are used to process orders for parts, say. The order forms are complicated. The user specifies the wanted action, and the terminal controller requests the appropriate form from a remotely located central computer. The forms may be several pages long and may have many fields on each page. A compressed version of the form is transmitted by the central computer to minimize communication line usage. The form is then translated by the terminal control PC for display by the dumb terminal. The terminal controller only transmits the answers (i.e., the contents of the blanks) back to the central computer. As each page of the form is filled out, the terminal controller transmits the answers to the central computer, which either accepts or rejects them. If the answers are invalid, a diagnostic code is transmitted by the central computer to the terminal controller, which in turn translates the code and informs the user at the terminal. Finally, the system allows the user to review the filled-out form.

Decision D1 in Figure 4.1 is not part of the process as such; it is really a characteristic of the kinds of transactions that the terminal controller handles. However, there is a decision like this somewhere in the program. Process P1 probably consists of several subsidiary processes that are completed by the transmission of the request for an order from the central computer. The next step is "process P2," which involves no real processing. What the terminal controller does here depends on the software's structure. Typically, transactions for some other terminal will be processed. Process P3 is a real processing step, as is P4. Process P6 is another wait for input. This has no direct correspondence to a program step. Decisions D2 and D4 depend on the structure of the form. Which branch is taken at decision D3 is determined by the user's behavior. The system does not necessarily have actual decisions corresponding to D1, D2, D3, D4, or D5; D1 and D5 for example, might be implemented by interrupts caused by special keys on the terminal or, if the terminal was itself a PC, by function keys.

The most general case of a transaction flow, then, represents by a flowgraph a scenario between people

and computers. In a more restricted example, the transaction flow can represent an internal sequence of events that may occur in processing a transaction.

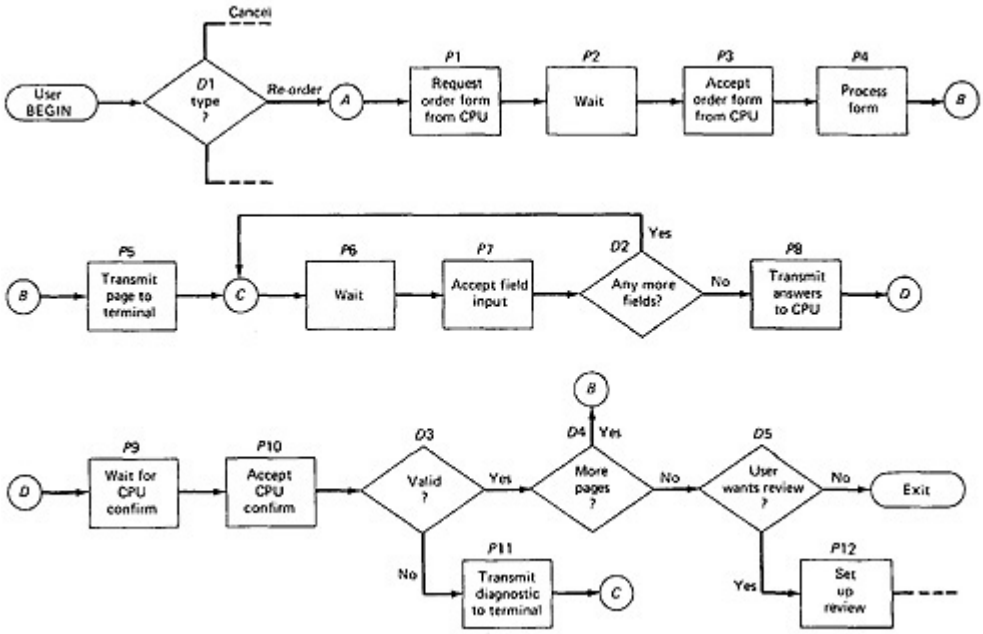

Figure 4.1. Example of a Transaction Flow.

3.3. Usage

Transaction flows are indispensable for specifying requirements of complicated systems, especially online systems. A big system such as an air traffic control or airline reservation system, has not hundreds, but thousands of different transaction flows. The flows are represented by relatively simple flowgraphs, many of which have a single straight-through path. Loops are infrequent compared to control flowgraphs. The most common loop is used to request a retry after user input errors. An ATM system, for example, allows the user to try, say three times, and will take the card away the fourth time.

3.4. Implementation

The implementation of a transaction flow is usually implicit in the design of the system's control structure and associated data base. That is, there is no direct, one-to-one correspondence between the "processes" and "decisions" of the transaction flows and corresponding program component. A transaction flow is a representation of a path taken by a transaction through a succession of processing modules. Think of each transaction as represented by a **token**—such as a transaction-control block that is passed from routine to routine as it progresses through its flow. The transaction flowgraph is a pictorial representation of what happens to the tokens; it is *not* the control structure of the program that manipulates those tokens.

Figure 4.2 shows another transaction flow and the corresponding implementation of a program that creates that flow. This transaction goes through input processing, which classifies it as to type, and then passes through process A, followed by B. The result of process B may force the transaction to pass back to process A. The transaction then goes to process C, then to either D or E, and finally to output processing.

Figure 4.2b is a diagrammatic representation of a software architecture that might implement this and many other transactions. The system is controlled by an executive/scheduler/dispatcher/operating

system—call it what you will. In this diagram the boxes represent processes and the links represent processing queues. The transaction enters (that is, it is created by) an input processing module in response to inputs received, for example, at a terminal. The transaction is "created" by the act of filling out a transaction-control block and placing that token on an input queue. The scheduler then examines the transaction and places it on the work queue for process A, but process A will not necessarily be activated immediately. When a process has finished working on the transaction, it places the transaction-control block back on a scheduler queue. The scheduler then examines the transaction control block and routes it to the next process based on information stored in the block. The scheduler contains tables or code that routes the transaction to its next process. In systems that handle hundreds of transaction types, this information is usually stored in tables rather than as explicit code. Alternatively, the dispatcher may contain no transaction control data or code; the information could be implemented as code in each transaction processing module.

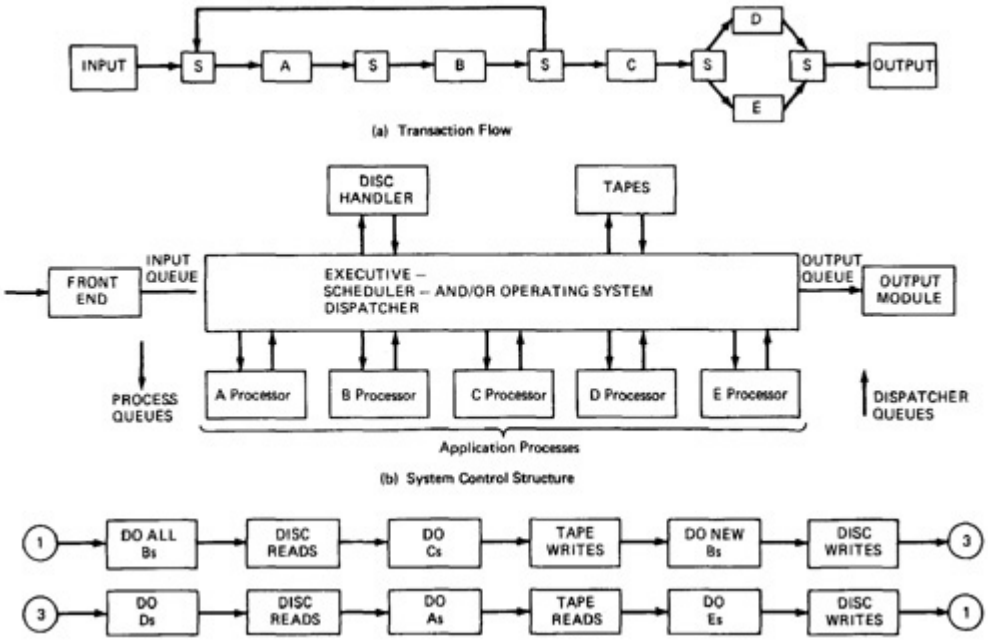

Figure 4.2. A Transaction Flow and Its Implementation.

Figure 4.2c shows a possible implementation of this transaction processing system (simplified). Let's say that while there could be many different transaction flows in the system, they all used only processes A, B, C, D, E, and disc and tape reads and writes, in various combinations. Just because the transaction flow order is A,B,C,D,E is no reason to invoke the processes in that order. For other transactions, not shown, the processing order might be B,C,A,E,D. A fixed processing order based on one transaction flow might not be optimum for another. Furthermore, different transactions have different priorities that may require some to wait for higher-priority transactions to be processed. Similarly, one would not delay processing for all transactions while waiting for a specific transaction to complete a necessary disc read operation.

In general, in multiprocessing systems there is no direct correspondence between the order in which processes are invoked and transaction flows. A given transaction will, of course, receive processing attention from the appropriate processing modules in the strict order required, but there could be many other things going on between the instances in which that transaction was being processed.

I left out the scheduler calls in Figure 4.2c to simplify things. Assume that there's a return of control to the scheduler after each process box. The whole program is organized as a simple loop. First, the scheduler invokes processing module B, which cleans up all transactions waiting for B processing at that

moment. Then the disc reads are initiated and the scheduler turns control over to module C, which clears up all of its tasks. After the tape writes are initiated, module B is invoked again to take care of any additional work that may have accumulated for it. The process continues, and finally the entire loop starts over again. A cyclic structure like this is common in process control and communications systems, among many others. Alternatively, a more complex control structure can be used in which the processing modules are not invoked in fixed order but in an order determined by the length of the queues for those modules, the priority of the tasks, the priority of the active modules, and the state of the system with respect to I/O operations. A queue-driven approach is more common in commercial operating systems. The reasons for choosing one control architecture over another is not germane to testing. It is a performance and resource-optimization question. For more information, see BEIZ78.

3.5. Perspective

We didn't say how many computers there are: it could be one, it could be dozens, uniprocessor or parallel, single-instruction multiple data **(SIMD)** or multi-instruction multiple data **(MIMD)**. We didn't restrict the communication methods between processing components: it could be via data structures, over communication lines, processing queues, or direct transfers in a call. We assumed nothing about the system's executive structure or operating system(s): interrupt driven, cyclic, multiprocessing, polled, free-running. There were no restrictions on how a transaction's identity is maintained: implicit, explicit, in transaction control blocks, or in task tables. How is the transaction's state recorded? Any way you want (but typically in the transaction's control block, assuming that such a thing exists). Transaction-flow testing is the ultimate black-box technique because all we ask is that there be something identifiable as a transaction and that the system will do predictable things to transactions.

Transaction flowgraphs are a kind of **data flowgraph** (KAV187, KODR77, RUGG79). That is, we look at the history of operations applied to data objects. You'll note some stylistic differences in how we constructed data flowgraphs compared to the rules for constructing control flowgraphs. The most important of these is the definition of link or basic block. In control flowgraphs we defined a link or block as a set of instructions such that if any one of them was executed, all (barring bugs) would be executed. For data flowgraphs in general, and transaction flowgraphs in particular, we change the definition to identify all processes of interest. For example, in Figure 4.1 the link between nodes A and C had five distinct processes, whereas in a control flowgraph there would be only one. We do this to get a more useful, revealing, model. Nothing is lost by this practice; that is, all theory and all characteristics of graphs hold. We have broken a single link into several. We'll do the same with data flow-graphs in Chapter 5.

Another difference to which we must be sensitive is that the decision nodes of a transaction flowgraph can be complicated processes in their own rights. Our transaction-flow model is almost always a simplified version of those decisions. Many decisions have exception exits that go to central recovery processes. Similarly, links may actually contain decisions, such as a recovery process invoked when a queue integrity routine determines that the link (queue) has been compromised (for example, it was linked back to itself in a loop). The third difference we tend to ignore in our transaction-flow models is the effect of interrupts. Interrupts can do the equivalent of converting every process box into a many-splendored thing with more exit links than a porcupine has spines. We just can't put all that into our transaction-flow model—if we did, we'd gain "accuracy" at the expense of intelligibility. The model would no longer be fit for test design.

3.6. Complications

3.6.1. General

Although in simple cases transactions have a unique identity from the time they're created to the time they're completed, in many systems a transaction can give birth to others, and transactions can also merge. The simple flowgraph is inadequate to represent transaction flows that split and merge.

3.6.2. Births

Figure 4.3 shows three different possible interpretations of the decision symbol, or nodes with two or more outlinks.

Figure 4.3a shows a decision node, as in a control flowgraph. Interpreted in terms of transaction flowgraphs, this symbol means that the transaction will either take one alternative or the other, but not both. In other words, this is a decision point of a transaction flow. Figure 4.3b shows a different situation. The incoming transaction (the parent) gives birth to a new transaction (the daughter), whence both transactions continue on their separate paths, the parent retaining its identity as a transaction. I call this situation **biosis**. Figure 4.3c is similar to Figure 4.3b, except that the parent transaction is destroyed and two new transactions (daughters) are created. I call this situation **mitosis** because of its similarity to biological mitotic division.

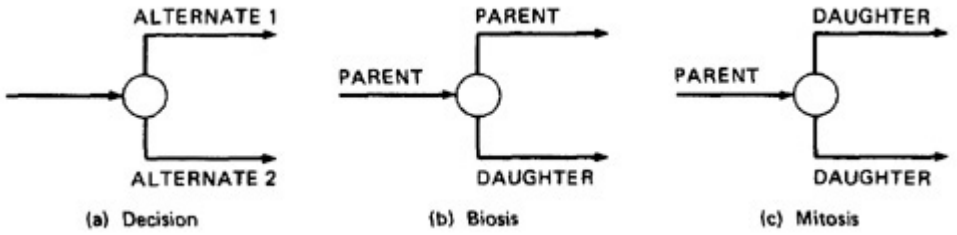

Figure 4.3. Nodes with Multiple Outlinks.

3.6.3. Mergers

Transaction-flow junction points (i.e., nodes with two or more inlinks) are potentially as troublesome as transaction-flow splits. Figure 4.4a shows the ordinary junction, which is similar to the junction in a control flow-graph. It is understood that a transaction can arrive either on one link (path 1) or the other (path 2). In Figure 4.4b (**absorption**) a predator transaction absorbs a prey. The prey is gone but the predator retains its identity. Figure 4.4c shows a slightly different situation in which two parent transactions merge to form a new daughter. In keeping with the biological flavor of this section, I call this act **conjugation.**

3.6.4. Theoretical Status and Pragmatic Solutions

The above examples do not exhaust the possible variations of interpretation of the decision and junction symbols in the context of real transaction flows. If we further consider multiprocessor systems and associated transaction coordination situations, our simple transaction-flow model no longer suffices. The most mature generic model for this kind of thing is a **Petri net** (KAVI87, MURA89, PETE76, PETE81, RAMA85, RUGG79). Petri nets use operations that can include and distinguish between all the above variations and more and also include mechanisms for explicitly representing tokens that traverse stages in the process. There is a mature theory of Petri nets (MURA89, PETE81), but it is beyond the scope of this book. Petri nets have been applied to hardware testing problems, protocol testing, network testing, and other areas, but their application to general software testing is still in its infancy. Nor do we yet have enough experience with the application of Petri nets to software testing to determine whether it is or is not a productive model for general software testing.

Recognizing that the transaction-flow model is imperfect but that a "correct" model is untried, what shall we, as testing practitioners, do? As with all models, we ignore the complexities that can invalidate the model and use what we can apply with ease. After all, models for testing are intended to give us insights into effective test case design—they're intuition joggers—so it doesn't matter that they're imperfect as long as the resulting tests are good. We have no problem with ordinary decisions and junctions. Here's a prescription for the troublesome cases:

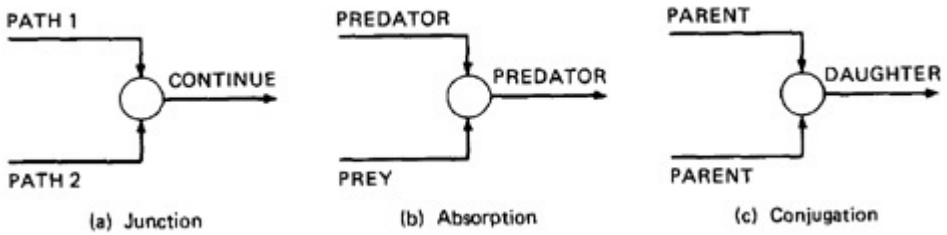

Figure 4.4. Transaction-Flow Junctions and Mergers.

1. *Biosis*—Follow the parent flow from beginning to end. Treat each daughter birth as a new flow, either to the end or to the point where the daughter is absorbed.
2. *Mitosis*—This situation involves three or more transaction flows: from the beginning of the parent's flow to the mitosis point and one additional flow for each daughter, from the mitosis point to each's respective end.
3. *Absorption*—Follow the predator as the primary flow. The prey is modeled from its beginning to the point at which it's eaten.
4. *Conjugation*—Three or more separate flows—the opposite of mitosis. From the birth of each parent proceed to the conjugation point and follow the resulting daughter from the conjugation point to her end.

Births, absorptions, and conjugations are as problematic for the software designer as they are for the software modeler and the test designer; as a consequence, such points have more than their share of bugs. The common problems are: lost daughters, wrongful deaths, and illegitimate births. So although you model transaction flows by simple flowgraphs, recognize that the likeliest place for bugs are where transactions are created, absorbed, or merged; keep track of such points and be sure to design specific tests to validate biosis, mitosis, absorption, and conjugations.

3.7. Transaction-Flow Structure

I've written harsh words about programmers who jump into the middle of loops to save a few lines of coding. The harsh words are warranted because more often than not (but not always), violations of so-called "rules" of structured design, have a deleterious impact on testability. How "well-structured" should transaction flows be? How well-structured should we expect them to be?

Just because transaction flows look like control flows, it does not follow that what constitutes good structure for code constitutes good structure for transaction flows—that's voodoo thinking based on the thaumaturgic principle of similarity.* As it turns out, transaction flows are often illstructured and there's nothing you can do about it. Here are some of the reasons:

---

*A fundamental law of magic. Objects and persons are manipulated through the principle of similarity by acting on an analog or facsimile of the object—for example, by sticking pins into the likeness of a hated person.

---

1. It's a *model* of a process, not just code. Humans may be involved in loops, decisions, and so on. We can't bind human behavior to software structure rules, no matter how fervently we might wish it.
2. Parts of the flows may incorporate the behavior of other systems over which we have no control. Why should we expect their behavior to be structured?
3. No small part of the totality of transaction flows exists to model error conditions, failures, malfunctions, and subsequent recovery actions. These are inherently unstructured—jumping out of loops, rampant GOTOs, and so on.
4. The number of transactions and the complexity of individual transaction flows grow over time as features are added and enhanced. Often, the world imposes the structure of the flows on us. Such structure may be the result of a congressional debate, a bilateral trade agreement, a contract, a salesman's boast after a four-martini "power lunch"—you name it. Should we expect good structure from politicians, salesmen, lawyers, and other drafters of social contracts? Try a transaction flowgraph for your income tax return if you still have doubts.
5. Systems are built out of modules and the transaction flows result from the interaction of those modules. Good system design dictates that we avoid changing modules in order to implement new transactions or to modify existing transactions. The result is that we build new paths between modules, new queues, add modules, and so on, and tie the whole together with ad hoc flags and switches to make it work. We may have to fool existing modules into doing the new job for us—that's usually preferable to changing many different modules. It's not unusual to deliberately block some paths in order to use a common part for several different kinds of transactions.
6. Our models are just that—approximations to reality. Interrupts, priorities, multitasking, multicomputers, synchronization, parallel processing, queue disciplines, polling—all of these make mincemeat out of structuring concepts that apply to units. The consequence of attempting to "structure" the transaction flows could be inefficient processing, poor response times, dangerous processing, security compromises, lost transaction integrity, and so on.

4. TRANSACTION-FLOW TESTING TECHNIQUES

4.1. Get the Transaction Flows

Complicated systems that process a lot of different, complicated transactions should have explicit representations of the transaction flows, or the equivalent, documented. If transaction flows are part of the system's specifications, half the battle is won. Don't expect to get pure transaction flows and don't insist on only that form of representing the system's processing requirements. There are other, equivalent representations, such as HIPO charts and Petri nets, that can serve the same purpose (MURA89, PETE76, PETE80). Also, because transaction flows can be mapped into programs, they can be described in a PDL. If such representations are used and if they are done correctly, it will be easy to create the transaction flows from them. The objective is to have a trace of what happens to transactions—a trace of the progression of actions, which, as we have seen, may not correspond to the design of the system executive or to the relation between the processing modules that work on the transaction. Transaction flows are like control flowgraphs, and consequently we should expect to have them in increasing levels of detail. It is correct and effective to have subflows analogous to subroutines in control flow-graphs, although there may not be any processing module that corresponds to such subflows.

Designers ought to understand what they're doing. And it's obvious that if they don't, they're not likely to do it right. I've made it a practice to ask for transaction flows—say, for the ten most important transactions that a system is to process—preparatory to designing the system's functional test. I hope that that information is in the specification. If it isn't, it's likely that there will be some disagreement as to what the system is supposed to be doing. More important, the system's design documentation should

contain an overview section that details the main transaction flows (all of them, it is hoped). If I can't find that or the equivalent, then I don't need a lot of complicated tests to know that the system will have a lot of complicated bugs. Detailed transaction flows are a mandatory prerequisite to the rational design of a system's functional test.

Like so much in testing, the act of getting the information on which to base tests can be more effective at catching and exterminating bugs than the tests that result from that information. Insisting on getting transaction flows or the equivalent is sometimes a gentle way of convincing inept design groups that they don't know what they're doing. These are harsh words, but let's face it: superb code and unit testing will be useless if the overall design is poor. And how can there be a rational, effective design if no one on the design team can walk you through the more important transactions, step by step and alternative by alternative. I'm sure that mine is a biased sample, but every system I've ever seen that was in serious trouble had no transaction flows documented, nor had the designers provided anything that approximated that kind of functional representation; however, it's certainly possible to have a bad design even with transaction flows.*

---

*I needed transaction flows to do a throughput model of a big system. It was another one of those bad projects on which I was consulted. Timing flows only represent high-probability paths, and the detailed order in which processing occurs within a link does not usually affect the timing analysis. I was having such poor luck getting the designers to create the transaction flows that, in desperation, I sent them simplified versions suitable only to timing analysis and asked for a confirmation or a correction—hoping thereby to stir them into action. You guessed it—my model flows appeared in the next monthly design release as *the* design flows. Thankfully, my name wasn't on them.

To reiterate: the first step in using transaction flows as a basis for system testing is to get the transaction flows. Often, that's the hardest step. Occasionally, it's the only step before the project's canceled.

4.2. Inspections, Reviews, Walkthroughs

Transaction flows are a natural agenda for system reviews or inspections. It's more important over the long haul that the designers know what it is the system is supposed to be doing than how they implement that functionality. I'd start transaction-flow walkthroughs at the preliminary design review and continue them in ever greater detail as the project progresses.

1. In conducting the walkthroughs, you should:
   - **a.** Discuss enough transaction types (i.e., paths through the transaction flows) to account for 98%–99% of the transactions the system is expected to process. Adjust the time spent and the intensity of the review in proportion to the perceived risk of failing to process each transaction properly. Let the nightmare list be your guide. The designers should name the transaction, provide its flowgraph, identify all processes, branches, loops, splits, mergers, and so on.
   - **b.** Discuss paths through flows in functional rather than technical terms. If a nontechnical buyer's representative who understands the application is present, so much the better. The discussion should be almost completely design independent. If the designers keep coming back to the design, its a sign of trouble because they
     may be implementing what they want to implement rather than what the user needs.
   - **c.** Ask the designers to relate every flow to the specification and to show how that transaction, directly or indirectly, follows from the requirements. Don't insist on a slavish one-to-one correspondence because that could lead to a poor implementation.

2. Make transaction-flow testing the cornerstone of system functional testing just as path testing is the cornerstone of unit testing. For this you need enough tests to achieve Cl and C2 coverage of the complete set of transaction flowgraphs.
3. Select additional transaction-flow paths (beyond C1 + C2) for loops, extreme values, and domain boundaries (see Chapter 6).
4. Select additional paths for weird cases and very long, potentially troublesome transactions with high risks and potential consequential damage.
5. Design more test cases to validate all births and deaths and to search for lost daughters, illegitimate births, and wrongful deaths.
6. Publish and distribute the selected test paths through the transaction flows as early as possible so that they will exert the maximum beneficial effect on the project.
7. Have the buyer concur that the selected set of test paths through the transaction flows constitute an adequate system functional test. Negotiate a subset of these paths to be used as the basis for a formal acceptance test.
8. Tell the designers which paths will be used for testing but not (yet) the details of the test cases that force those paths. Give them enough information to design their own test cases but not so much information that they can develop a "Mickey Mouse" fix that covers a specific test case but is otherwise useless.

4.3. Path Selection

Path selection for system testing based on transaction flows should have a distinctly different flavor from that of path selection done for unit tests based on control flowgraphs. Start with a covering set of tests (C1 + C2) using the analogous criteria you used for structural path testing, but don't expect to find too many bugs on such paths.

Select a covering set of paths based on functionally sensible transactions as you would for control flowgraphs. Confirm these with the designers. Having designed those (easy) tests, now do exactly the opposite of what you would have done for unit tests. Try to find the most tortuous, longest, strangest path from the entry to the exit of the transaction flow. Create a catalog of these weird paths. Go over them not just with the high-level designer who laid out the transaction flows, but with the next-level designers who are implementing the modules that will process the transaction. It can be a gratifying experience, even in a good system. The act of discussing the weird paths will expose missing interlocks, duplicated interlocks, interface problems, programs working at cross-purposes, duplicated processing—a lot of stuff that would otherwise have shown up only during the final acceptance tests, or worse, after the system was operating. The entire cost of independent testing can be paid for by a few such paths for a few well-chosen transactions. This procedure is best done early in the game, while the system design is still in progress, before processing modules have been coded. I try to do it just after the internal design specifications for the processing modules are completed and just before those modules are coded. Any earlier than that, you'll get a lot of "I don't know yet" answers to your questions, which is a waste of both your time and the designer's. Any later, it's already cast into code and correction has become expensive.

This process has diminishing returns. Most competent designers won't be caught twice. You have only to show them one nasty case and they'll review all their interfaces and interlocks and you're not likely to find any new bugs from that module—but you can catch most modules and their designers once. Eventually the blatant bugs have been removed, and those that remain are due to implementation errors and wild cards. Bringing up a weird path after a few rounds of this will just make the designer smirk as she shows you just how neatly your weird cases, and several more that you hadn't thought of, are handled.

The covering set of paths belong in the system feature tests. I still try to augment cover with weird paths in addition to normal paths, if I possibly can. It gives everybody more confidence in the system and its test. I also keep weird paths in proportion to what I perceive to be the designer's lack of confidence. I suppose it's sadistic hitting a person who's down, but it's effective. Conversely, you do get fooled by supremely confident idiots and insecure geniuses.

4.4. Sensitization

I have some good news and bad news about sensitization. Most of the normal paths are very easy to sensitize—80%–95% *transaction flow* coverage (C1 + C2) is usually easy to achieve.* The bad news is that the remaining small percentage is often very difficult, if not impossible, to achieve by fair means. While the simple paths are easy to sensitize there are many of them, so that there's a lot of tedium in test design. Usually, just identifying the normal path is enough to sensitizing it. In fact, many test designers who do mainly transaction flow testing (perhaps not by that name) are not even conscious of a sensitization issue. To them, sensitization *is* the act of defining the transaction. If there are sensitization problems on the easy paths, then bet on either a bug in transaction flows or a design bug.

---

*Remember, don't confuse *transaction-flow coverage* with *code coverage*.

How about the off-paths, the exception conditions, the path segments on which we expect to find most of the bugs? The reason these paths are often difficult to sensitize is that they correspond to error conditions, synchronization problems, overload responses, and other anomalous situations. A constant headache along this line is testing a protocol across an external interface. In order to test our abnormal paths we may have to ask the other system to simulate bad transactions or failures—it's usually a live system and getting that kind of cooperation is harder than catching Santa Claus coming down the chimney. I said that it was tough to sensitize such path segments if we were going to be fair about it. It's not as bad if we allow ourselves to play dirty. Here's a short list:

1. *Use Patches*—The dirty system tester's best, but dangerous, friend.* It's a lot easier to fake an error return from another system by a judicious patch than it is to negotiate a joint test session—besides, whom are we kidding? If we don't put the patch into *our* system, they (the ones on the other side of the interface) will have to put the patch into *their* system. In either case, somebody put in an "unrealistic" patch.

---

*And then there was the time I inadvertently cut off a major part of U.S. communications with Europe several times in one night by a badly installed patch during live testing.

---

2. *Mistune*—Test in a system sized with grossly inadequate resources. By "grossly" I mean about 5%–10% of what one might expect to need. This helps to force most of the resource-related exception conditions.

3. *Break the Rules*—Transactions almost always require associated, correctly specified, data structures to support them. Often a system database generator is used to create such objects and to assure that all required objects have been correctly specified. Bypass the database generator and/or use patches to break any and all rules embodied in the database and system configuration that will help you to go down the desired path.

4. *Use Breakpoints*—Put breakpoints at the branch points where the hard-to-sensitize path

segment begins and then patch the transaction control block to force that path.

You can use one or all of the above methods, and many I haven't thought of, to sensitize the strange paths. These techniques are especially suitable for those long tortuous paths that avoid the exit. When you use them, you become vulnerable to the designer's cry of "foul." And there's some justification for that accusation. Here's the point, once you allow such shenanigans, you allow arbitrary patches. Who *can't* crash a system if she's allowed to patch anything anywhere, anytime? It's unreasonable to expect a system to stand up to that kind of abuse. Yet, you must do such things if you're to do a good job of testing. The solution to this dilemma is to be meticulous about such tests; to be absolutely sure that there's no other reasonable way to go down that path; to be absolutely sure that there's no error in the patch; and to be ready for a fight with the designers every time you employ such methods.

4.5. Instrumentation

Instrumentation plays a bigger role in transaction-flow testing than in unit path testing. Counters are not useful because the same module could appear in many different flows and the system could be simultaneously processing different transactions. The information of the path taken for a given transaction must be kept with that transaction. It can be recorded either by a central transaction dispatcher (if there is one) or by the individual processing modules. You need a trace of all the processing steps for the transaction, the queues on which it resided, and the entries and exits to and from the dispatcher. In some systems such traces are provided by the operating system. In other systems, such as communications systems or most secure systems, a running log that contains exactly this information is maintained as part of normal processing. You can afford heavy instrumentation compared to unit-testing instrumentation because the overhead of such instrumentation is typically small compared to the processing. Another, better alternative is to make the instrumentation part of the system design. Augment the design as needed to provide complete transaction-flow tracing for all transactions. The payoff in system debugging and performance testing alone is worth the effort for most systems. And it's a lot easier and better to design the instrumentation in than it is to retrofit it.

4.6. Test Databases

About 30%–40% of the effort of transaction-flow test design is the design and maintenance of the test database(s). It may be a third of the labor, but it carries a disproportionately high part of the headaches. I've seen two major, generic errors in the design of such databases—and they often follow in sequence on the same project. The first error is to be unaware that there's a test database to be designed. The result is that every programmer and tester designs his own, unique database, which is incompatible with all other programmers' and testers' needs. The consequence is that every tester (independent or programmer) needs exclusive use of the entire system. Furthermore, many of the tests are configuration-sensitive, so there's no way to port one set of tests over from another suite. At about the time that testing has ground to a halt because of this tower of babble—when the test bed floor has expanded to several acres of gear running around the clock—it's decided that test data bases must be configuration-controlled and centrally administered under a comprehensive design plan. Often, because the independent testers need more elaborate test setups than do the programmers, the responsibility is given to the test group. That's when the second error occurs. In order to avoid a repetition of the previous chaos, it is decided that there will be one comprehensive database that will satisfy all testing needs. The design and debugging of such a monster becomes a mighty project in its own right. A typical system of a half-million lines of source code will probably need four or five different, incompatible databases to support testing. The design of these databases is no less important than the design of the system data structures. It requires talented, mature, diplomatic, experienced designers—experienced both in the system design and in test design.

4.7. Execution

If you're going to do transaction-flow testing for a system of any size, be committed to test execution automation from the start. If more than a few hundred test cases are required to achieve C1 + C2 transaction-flow coverage, don't bother with transaction-flow testing if you don't have the time and resources to almost completely automate all test execution. You'll be running and rerunning those transactions not once, but hundreds of times over the project's life. Transaction-flow testing with the intention of achieving C1 + C2 usually leads to a big (four- to fivefold) increase in the number of test cases. Without execution automation you can't expect to do it right. Capture/replay tools and other test drivers are essential. See Chapter 13 for more details.

5. IMPLEMENTATION COMMENTS

5.1. Transaction Based Systems

Let's go back to Figure 4.2b. There's a lot we can imply in this design that makes transaction-flow testing easy.

1. *Transaction Control Block*—There's an explicit transaction control block associated with every live transaction. The block contains, among other things, the transaction's type, identity, and processing state. Whether the block itself contains the information or just pointers to the information is immaterial. What matters is that there's a unique data object through which anything we want to know about a transaction is easily found. The control block is created when the transaction is born and is returned to the pool when the transaction leaves the system for archival storage.
2. *Centralized, Common, Processing Queues*—Transactions (actually, transaction control blocks) are not passed directly from one process to another but are transferred from process to process by means of centralized explicit processing queues. The dispatcher links control blocks to processes. Processes link control blocks back to the dispatcher when they are done. Instead of $O(n^2)$ ad hoc queues between n processors, the number of explicit queues is proportional to the number of processors.
3. *Transaction Dispatcher*—There is a centralized transaction dispatcher. Based on the transaction's current state and transaction type, the next process is determined from stored dispatch tables or a similar mechanism. For example, use a finite-state machine implementation (see Chapter 11).
4. *Recovery and Other Logs*—Key points in the transaction's life are recorded for several different purposes—the two most important being transaction recovery support and transaction accounting. At the very least, there are birth and death entries. Other entries may be provided on request as part of development support (e.g., debugging) or for tuning. The most complete instrumentation provides coverage certification (C1 + C2) over the transaction flows. These facilities obviate the need for special instrumentation.
5. *Self-Test Support*—The transaction control tables have privileged modes that can be used for test and diagnostic purposes. There are special transaction types and states for normal transactions whose sole purpose is to facilitate testing. With these features in place, it is never necessary to use patches to force paths.

It's not my intention to tantalize you with a tester's paradise, but it's obvious that transaction-flow testing for a system with the above features is as easy as system testing gets. The designer has recognized that transactions are entities worth reckoning with, has probably done an explicit analysis of all transaction flows, and has implemented a design that is relatively invulnerable to even radical

changes in the required flows—and, incidentally, has made the system tester's job much easier. The designer has also made it relatively unlikely that transaction-flow testing will reveal any bugs at all—the pesticide paradox.

A good system functional test achieves C1 + C2 over the transaction flows, no matter how hard that task may be or how easy the designer has made the job for us. If the design is an ad hoc mess of interprocess queues with decisions made here and there and everywhere, transaction routing determined locally in low-level routines, with most of the transaction state information implicit, then transaction-flow testing will yield rich rewards—personally, it's as much fun as taking candy away from sick babies. It's so easy to break such systems that I've sometimes been ashamed of it. But how about the testable design outlined above? Transaction-flow testing will reveal an occasional goof here and there, but rarely anything over which to panic. And whatever bugs are found are easily fixed. Our testing resources are limited and must be spent on the potentially most revealing tests. If you're testing a system with an explicit transaction-control mechanism, then don't do any more transaction-flow testing than you need to satisfy risk and contractual issues. Conversely, if it's a piece of ad hoc garbage, transaction-flow system testing is where it's at.

5.2. Hidden Languages.

Don't expect to find neat decision boxes and junctions as in a control flowgraph. The actual decision may be made (and usually is) in a processing module, and the central dispatcher (if there is one) is usually indifferent to the transaction flows. Alternatively, the dispatcher may direct the flows based on control codes contained in the transaction's control block or stored elsewhere in the database. Such codes actually constitute an internal language. If an explicit mechanism of this kind does exist, the transaction flows are indeed implemented as "programs" in this internal language. A commercial operating system's job control language or the set of task control codes combined with execution files are examples of such "languages."

The trouble is that these languages are often undeclared, undefined, undocumented, and unrecognized. Furthermore, unlike formal higher-order languages and their associated compilers or interpreters, neither the language's syntax, semantics, nor processor has been debugged. The flow-control language evolves as a series of ad hoc agreements between module designers. Here are some of the things that can go wrong:

1. The language is rarely checked for self-consistency (assuming that its existence has been recognized).
2. The language must be processed, usually one step at a time, by an interpreter that may be centralized in a single routine but is more often distributed in bits and pieces among all the processing modules. That interpreter, centralized or distributed, may have bugs.
3. The "program"—i.e., the steps stored in a transaction-control table—may have bugs or may become corrupted by bugs far removed from the transaction under consideration. This is a nasty kind of bug to find. Just as a routine may pass its tests but foul the database for another routine, a transaction may pass all of its tests but, in so doing, foul the transaction-control tables for subsequent transactions.
4. Finally, any transaction processing module can have bugs.

If transaction-control tables are used to direct the flow, it is effective to treat that mechanism as if an actual language has been implemented. Look for the basic components of any language—processing steps, conditional-branch instructions, unconditional branches, program labels, subroutines (i.e., the tables can direct processing to subtables which can be used in common for several different flows), loop control, and so on. Assuming you can identify all of these elements, document a "syntax" for this

primitive, undeclared language and discuss this syntax with whoever is responsible for implementing the transaction-control structure and software. The pseudolanguage approach was used to provide flexibility in implementing many different, complicated transaction flows. Therefore, it is reasonable to expect that any syntactically valid "statement" in this language should make processing sense. A syntactically valid, arbitrary statement might not provide a useful transaction, but at the very least,the system should not blow up if such a transaction were to be defined. Test the syntax and generate test cases as you would for any syntax-directed test, as discussed in Chapter 9.

6. TESTABILITY TIPS

1. Implement transaction flows along with supporting data structures as a primary specification and design tool.
2. Use explicit transaction-control blocks as a primary control mechanism.
3. Build all instrumentation in as part of the system design.
4. Use a centralized, table-driven, transaction dispatcher to control transaction flows.
5. Implement transaction control as table-driven finite-state machines (see Chapter 11) for which transaction state information is stored in the transaction-control block.
6. Be willing to add fields to transaction-control blocks and states to transaction flows if by doing so you can avoid biosis, mitosis, absorption, and conjugation.
7. Make sure there is one-to-one correspondence between transaction-flow paths and functional requirements, if possible.
8. To the extent that you can control it, apply the same structure rules to transaction flows that you apply to code—i.e., avoid jumping into or out of loops, unessential GOTOs, deeply nested loops, and so on.
9. Emulate the testability criteria for code; that is, keep the number of covering paths, number of feasible paths, and total path number as close as possible to one another.
10. Don't be afraid to design transaction-control languages. But if you do, validate the correctness and sanity of the syntax and semantics. Treat the design of the language and its translator or interpreter as you would the design of a new programming language. Test the language and its processor thoroughly before you test transactions.

7. SUMMARY

1. The methods discussed for path testing of units and programs can be applied with suitable interpretation to functional testing based on transaction flows.
2. The biggest problem and the biggest payoff may be getting the transaction flows in the first place.
3. Full coverage (C1 + C2) is required for all flows, but most bugs will be found on the strange, meaningless, weird paths.
4. Transaction-flow control may be implemented by means of an undeclared and unrecognized internal language. Get it recognized, get it declared, and then test its syntax using the methods of Chapter 9.
5. The practice of attempting to design tests based on transaction-flow representation of requirements and discussing those attempts with the designer can unearth more bugs than any tests you run.

<ch3 toc ch5>