Module 5

System Testing and Interactive Testing

System Testing

Of the three levels of testing, the system level is closest to everyday experience. We test many things: a used car before we buy it, an online network service before we subscribe, and so on. A common pattern in these familiar forms is that we evaluate a product in terms of our expectations - not with respect to a specification or a standard. Consequently, the goal is not to find faults, but to demon- urate correct behavior. Because of this, we tend to approach system testing from a functional stand- point instead of from a structural one. Because it is so intuitively familiar, system testing in practice tends to be less formal than it might be, and this is compounded by the reduced testing interval that usually remains before a delivery deadline.

The craftsperson metaphor continues to serve us. We need a better understanding of the medium. We will view system testing in terms of threads of system-level behavior. We begin with a new construct an atomic system function and further elaboration on the thread concept, highlighting some of the practical problems of thread-based system testing. System testing is closely coupled with requirements specification; therefore, we will discuss how to find threads in common notations. All this leads to an orderly thread-based system testing strategy that exploits the symbiosis between functional and structural testing. We will apply the strategy to our simple automated teller machine (SATM) system.

Threads

Threads are hard to define; in fact, some published definitions are counterproductive, misleading, or wrong. It is possible to simply treat threads as a primitive concept that needs no formal definition. For now, we will use examples to develop a shared vision. Here are several views of a thread:

- A scenario of normal usage
- A system-level test case
- A stimulus-response pair
- Behavior that results from a sequence of system-level inputs
- An interleaved sequence of port input and output events
- A sequence of transitions in a state machine description of the system
- An interleaved sequence of object messages and method executions
- A sequence of machine instructions

- A sequence of source instructions
- A sequence of MM-Paths
- A sequence of atomic system functions

Threads have distinct levels. A unit-level thread is usefully understood as an execution-time path of source instructions or, alternatively, as a sequence of DD-Paths. An integration-level thread is an MM-Path - that is, an alternating sequence of module executions and messages. If we con- tinue this pattern, a system-level thread is a sequence of atomic system functions (to be defined shortly). Because atomic system functions have port events as their inputs and outputs, a sequence of atomic system functions implies an interleaved sequence of port input and output events. The end result is that threads provide a unifying view of our three levels of testing. Unit testing tests individual functions; integration testing examines interactions among units, and system testing examines interactions among atomic system functions. In this chapter, we focus on system-level threads and answer some fundamental questions, such as: How big is a thread? Where do we find them?

Thread Possibilities

Defining the endpoints of a system-level thread is a bit awkward. We motivate a tidy graph theory-based definition by working backward from where we want to go with threads. Here are four candidate threads in our SATM system:

- Entry of a digit
- Entry of a personal identification number (PIN)
- A simple transaction: ATM card entry, PIN entry, select transaction type (deposit, with-
draw), present account details (checking or savings, amount), conduct the operation, and
- report the results
- An ATM session containing two or more simple transactions

Digit entry is a good example of a minimal atomic system function. It begins with a port input event (the digit keystroke) and ends with a port output event (the screen digit echo), so it qualifies as a stimulus-response pair If you go back to our example in Chapter 13. you will see that this atomic system function (ASF) is a subpath of the sample MM-Path that we listed in great detail. This level of granularity is too fine for the purposes of system testing. We saw this to be an appropriate level for integration testing.

The second candidate, PIN entry, is a good example of an upper limit to integration testing and, at the same time, a starting point of system testing. PIN entry is also a good example of an atomic system function and a family of stimulus-response pairs (system-level behavior that is initiated by a port input event, traverses some programmed logic, and terminates in one of several possible responses (port output events). As we saw in Chapter 13, PIN entry entails a sequence of system-level inputs and outputs:

1. A screen requesting PIN digits.
2. An interleaved sequence of digit keystrokes and screen responses.
3. The possibility of cancellation by the customer before the full PIN is entered.
4. A system disposition: A customer has three chances to enter the correct PIN. Once a correct PIN has been entered, the user sees a screen requesting the transaction type, otherwise, a screen advises the customer that the ATM card will not be returned, and no access to ATM functions is provided.

This is clearly in the domain of system-level testing, and several stimulus-response pairs are evident. Other examples of ASFs include card entry, transaction selection, provision of transaction details, transaction reporting, and session termination. Each of these is maximal in an integration testing sense and minimal in a system testing sense. That is, we would not want to integration test something larger than an ASF; at the same time, we would not want to system test anything smaller.

The third candidate, the simple transaction, has a sense of end-to-end completion. A customer could never execute PIN entry alone (a card entry is needed), but the simple transaction is com only executed. This is a good example of a system-level thread, note that it involves the interaction of several ASFS.

The last possibility (the session) is actually a sequence of threads. This is also properly a part of system testing, at this level, we are interested in the interactions among threads. Unfortunately, most system testing efforts never reach the level of thread interaction (more on this in Chapter 15).

We simplify our discussion by defining a new term that helps us get to our desired goal.

Definition

An atomic system function (ASF) is an action that is observable at the system level in terms of port input and output events.

In an event-driven system, ASFs are separated by points of event quiescence: these occur when a system is (nearly) idle, waiting for a port input event to trigger further processing. Event quiescence has an interesting Petri net insight. In a traditional Petri net, deadlock occurs when no transition is enabled. In an event-driven Petri net, event quiescence is similar to deadlock, but an input event can bring new life to the net. The SATM system exhibits event quiescence in several Enlaces: one is the tight loop at the beginning of SATM Main, where the system has displayed the and is waiting for a card to be entered into the card slot. Event quiescence is a system-level property, there is an analog at the integration level - message quiescence. The notion of event quiescence does for ASFs what message quiescence does for MM-Paths:

it provides a natural endpoint. An ASF begins with a port input event, traverses parts of one or more MM-Paths, and terminates with a port output event. When viewed from the system level, no compelling reason exists to decompose an ASF into lower levels of detail (hence the atomicity). In the SATM system, digit entry is a good example of an ASF - so are card entry, cash dispensing, and session closing. PIN entry is probably too big perhaps we should call it a molecular system function.

Atomic system functions represent the seam between integration and system testing. They are the largest item to be tested by integration testing and the smallest item for system testing. We can test an ASF at both levels. Again, the digit entry ASF is a good example During system testing, the port input dataflow testing, and the decomposition (and eventual implementation) of actions is the bush of structural testing.

Basis Concepts for Requirements Specification

Recall the notion of a basis of a vector space a set of independent elements from which all the elements in the space can be generated. Instead of anticipating all the variations in scores of requirements specification methods, notations, and techniques, we will discuss system testing with respect to a basis set of requirements specification constructs data, actions, devices, events. and threads (Jorgensen, 1989). Every system can be expressed in terms of these five fundamental concepts fand every requirements specification technique is some combination of these) We examine these fundamental concepts here to see how they support the tester's process of thread Identification

1. **Data**

When a system is described in terms of its data, the focus is on the information used and created by the system. We describe data in terms of variables, data structures, fields, records, data stores, and files. Entity/relationship models are the most common choice at the highest level, and some form of a regular expression (e.g.. Jackson diagrams or data structure diagrams) is used at a more detailed level. The data-centered view is also the starting point for several flavors of object-oriented analysis. Data refers to information that is initialized, stored, updated, or (possibly) destroyed. In the SATM system, initial data describes the various accounts (PANs) and their PINs, and each account has a data structure with information such as the account balance. As ATM transactions occur, the results are kept as created data and used in the daily posting of terminal data to the central bank. For many systems, the data-centered view dominates. These systems are often developed in terms of CRUD actions (create, retrieve, update, delete). We could describe the transaction portion of the SATM system in this way, but it would not work well for the user interface portion.

Sometimes threads can be identified directly from the data model. Relationships between data entities can be one-to-one, many-to-one, or many-to-many, these distinctions all have implications for threads that process the data. For example, if bank customers can have several accounts, tach account needs a unique PIN. If several people can access the same account, they need ATM card with identical PANs. We can also find initial data (such as PAN, ExpectedPIN pairs) that read but never written. Such read-only data must be part of the system initialization process. If not, there must be threads that create such data. Read-only data is therefore an indicator of source ASFs.

2. Actions

Arion-centered modeling is still a common requirements specification form. This is a historical outgrowth of the action-centered nature of imperative programming languages. Actions have Inputs and outputs, and these can be either data or port events. Here are some methodology- specific synonyms for actions: transform, data transform, control transform, process, activity, task, method, and service. Action: can also be decomposed into lower-level actions, as we saw with the data flow diagrams. The input/output view is exactly the basis of functional testing, and decomposition of actions is the basis of structural testing

3. Devices

Every system has port devices, these are the sources and destinations of system-level inputs and outputs (port events). The slight distinction between ports and port devices is sometimes helpful outputs Technically, a port is the point at which an I/O device is attached to a system, as in serial and parallel ports, network ports, and telephone ports. Physical actions (keystrokes and light serial and from a screen) occur on port devices, and these are translated from physical to logical for logical to physical). In the absence of actual port devices, much of system testing can be accomplished by "moving the port boundary inward" to the logical instances of port events. From now on, we will just use the term port to refer to port devices. The ports in the SATM system include the digit and cancel keys, the function keys, the display screen, the deposit and withdrawal doon, the card and receipt slots, and several less obvious devices, such as the rollers that move cards and

deposit envelopes into the machine, the cash dispenser, the receipt printer, and so on. Thinking about the ports helps the tester define the input space that functional system testing needs, similarly, the output devices provide output-based functional test information. (For example, we would like to have enough threads to generate all 15 SATM screens.)

4. Events

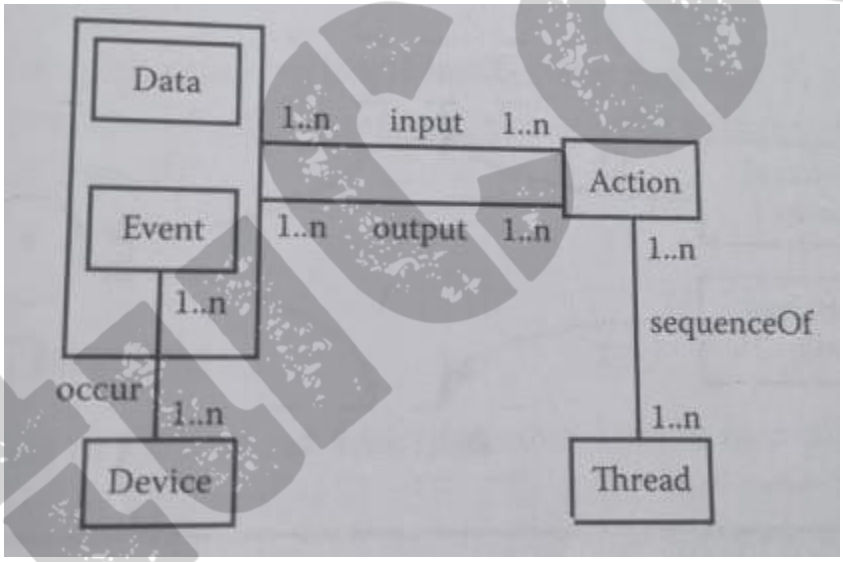
Events are somewhat schizophrenic: they have some characteristics of data and some of actions An event is a system-level input (or output) that occurs on a port device. Similar to data, events can be inputs to or outputs of actions. Events can be discrete (such as SATM keystrokes) or continuous (such as temperature, altitude, or pressure) Discrete events necessarily have a time duration, and this can be a critical factor in real-time systems. We might picture input events as destructive readout data, but it is a stretch to imagine output events as destructive write operations.

Events are like actions in the sense that they are the translation point between real-world physical events and internal logical manifestations of these. Port input events are physical-to-logical translations, and symmetrically, port output events are logical-to-physical translations. System testers should focus on the physical side of events, not the logical side (the focus of integration testers). Situations occur where the context of present data values changes the logical meaning of physical events. In the SATM system, for example, the port input event of depressing button Bl means "Balance" when screen 5 is displayed, "checking" when screen 6 is displayed, and "yes" when screens 10, 11, and 14 are

displayed. We refer to such situations as context-sensitive port events, and we would expect to test such events in each context.

5. Threads

Unfortunately for testers, threads are the least frequently used of the five fundamental constructs. Because we test threads, it usually falls to the tester to find them in the interactions among the data, events, and actions. About the only place that threads appear per se in a requirements specification is when rapid prototyping is used in conjunction with a scenario recorder. It is easy to fol threads in control models, as we will soon see. The problem with this is that control model r just that they are models, not the reality of a system.



14.2.6 Relationships among Basis Concepts

Figure 14.1 is an entity/relationship (E/R) model of our basis concepts. Notice that all relationships are many-to-many- Data and Events are generalized into an entity, the two relationships to the Action entity are for inputs and outputs. The same event can occur on several ports, and typically many events occur on a single port. Finally, an action can occur in several threads, and a thread is composed of several actions. This diagram demonstrates

some of the difficulty of system testing. Testers must use events and threads to ensure that all the many-to-many relationships among the five basis concepts are correct.

14.2.7 Modeling with Basis Concepts

All flavors of requirements specification develop models of a system in terms of the basis concepts. Figure 14.2 shows three fundamental forms of requirements specification models: structural, con textual, and behavioral. Structural models are used for development; these express the functional decomposition, data decomposition, and interfaces among components. Contextual models are often the starting point of structural modeling. They emphasize system devices and, to a lesser extent, actions, and threads very indirectly. The models of behavior (also called control models)

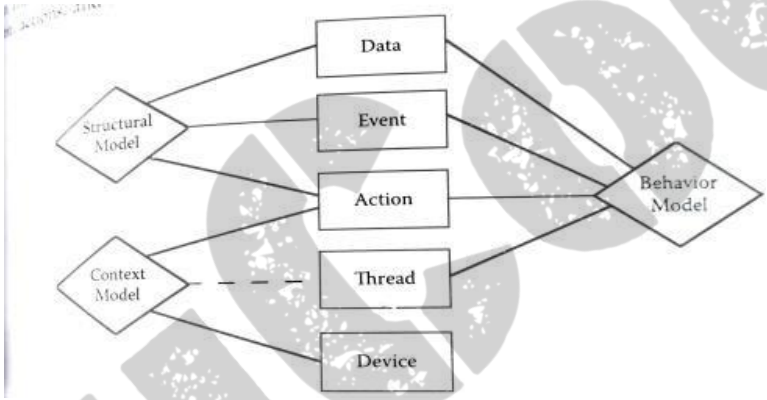


Fig: 14.2 Modeling Relationships among basic constructs

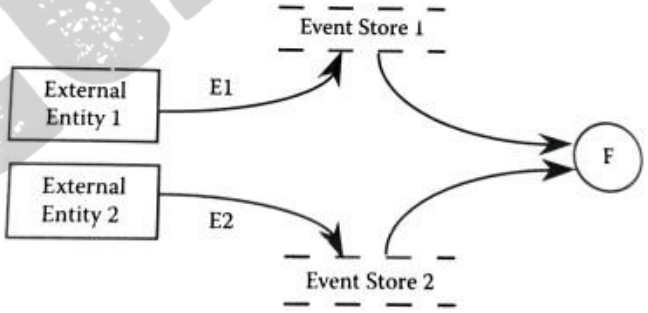


Fig 14.3: Event Partitioning views of function F

are where four of the five basis constructs come together. Selection of an appropriate control model is the essence of requirements specification models that are too weak cannot

express important system behaviors, while models that are too powerful typically obscure interesting behaviors. As a general rule, decision tables are a good choice only for computational systems; finite state machines are good for menu-driven systems; and Petri nets are the model of choice for concurrent systems. Here, we use finite state machines for the SATM system, and in Chapter 15, we will use Petri nets to analyze thread interaction.

We must make an important distinction between a system itself (reality) and models of a system. Consider a system in which some function F cannot occur until two prerequisite events El and E2 have occurred, and that they can occur in either order. We could use the notion of event partitioning to model this situation. The result would be a diagram like that in Figure 14.3.

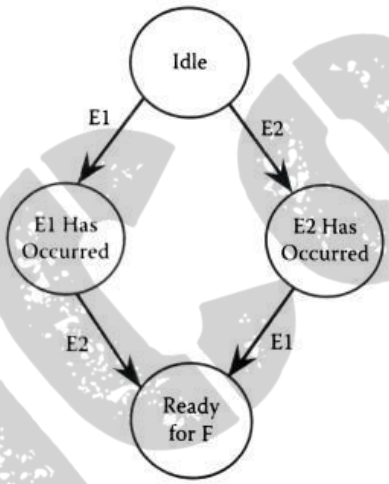


Fig: FSM for function F

In the event partitioning view, events El and E2 occur on their respective external devices. When they occur, they are held in their respective event stores. (An event store acts like a destructive read operation.) When both events have occurred, function F gets its prerequisite information from the event stores. Notice that we cannot tell from the model which event occurs first; we only know that both must occur.

We could also model the system as a finite state machine (FSM) in Figure 14.4, in which states record which event has occurred. The state machine view explicitly shows the two orders of the events.

Both models express the same prerequisites for the function F, and neither is the reality of the system. Of these two models, the state machine is more useful to the tester, because paths are instantly convertible to threads.

14.3 Finding Threads

The finite state machine models of the SATM system are the best place to look for system testing threads. We will start with a hierarchy of state machines; the upper level is shown in Figure. At this this level, states correspond to stages of processing, and transitions are caused by logical (instead of port) events. The card entry state, for example, would be decomposed into lower levels that deal with details like jammed cards, cards that are upside down, stuck card rollers, and checking the card against the list of cards for which service is offered. Once the details of a macro-state.

The PIN entry state is decomposed into more detailed view, which is a slight revision of the previous version.

The Adjacent states are shown because they are sources and destinations of transitions from the PIN entry portion. At this level, we can focus on the pin retry mechanism; all of the output events are true port events, but the input events are still logical events. The states and edges are numbered for reference later when we discuss test coverage.

To start the thread identification process, we first list the port events shown on the state transition; they appear in given table. We Skipped the eject card event because it is not really part of the PIN entry component.

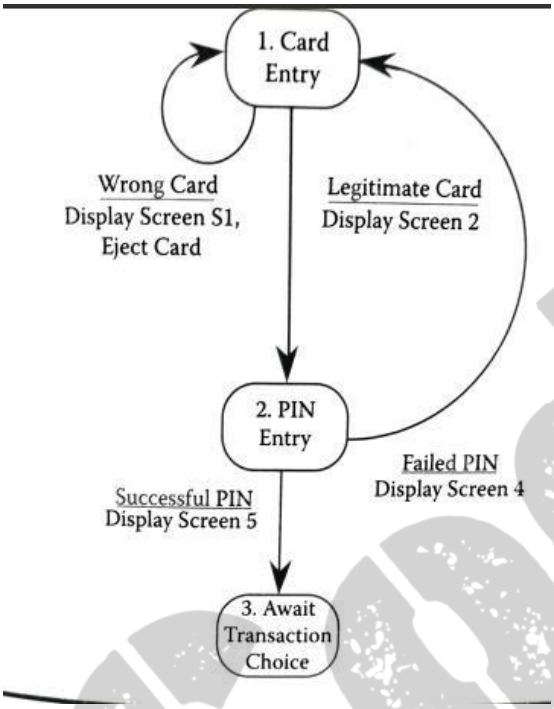


Fig: Top Level SATM state machine

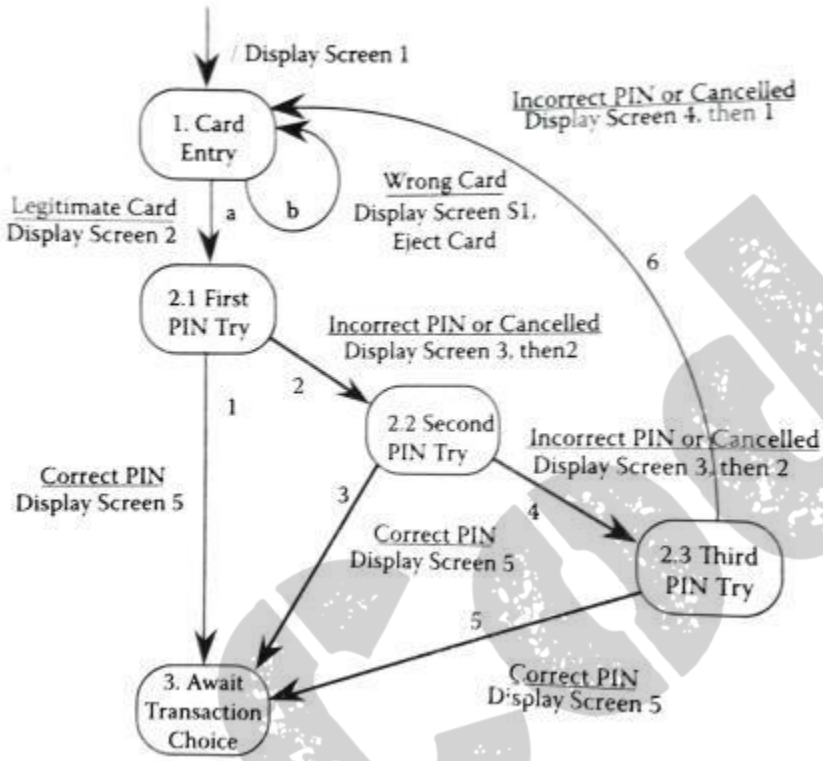


Fig: PIN entry finite state machine

Notice that correct PIN and incorrect PIN are really compound port input events. We cannot actually enter an entire PIN we enter digits, and at any point, we might hit the cancel key. These more detailed possibilities are shown in Figure 14.7. A truly paranoid tester might decom pose the digit port input event into the actual choices (0-pressed, 1-pressed,.... 9-pressed), but this should have been tested at a lower level. The port events in the PIN try finite state machine are in Table 14.2.

The "x" in the state names in the PIN try machine refers to which try (first, second, or third) is passing through the machine

In addition to the true port events in the PIN try finite state machine, there are three logical output events (correct pin, incorrect pin, and canceled); these correspond exactly to the higher- level events in Figure 14.6.

Table 14.1 Events in the PIN Entry Finite State Machine

Port Input Events	*Port Output Events*
Legitimate card	Display screen 1
Wrong card	Display screen 2
Correct PIN	Display screen 3
Incorrect PIN	Display screen 4
Canceled	Display screen 5

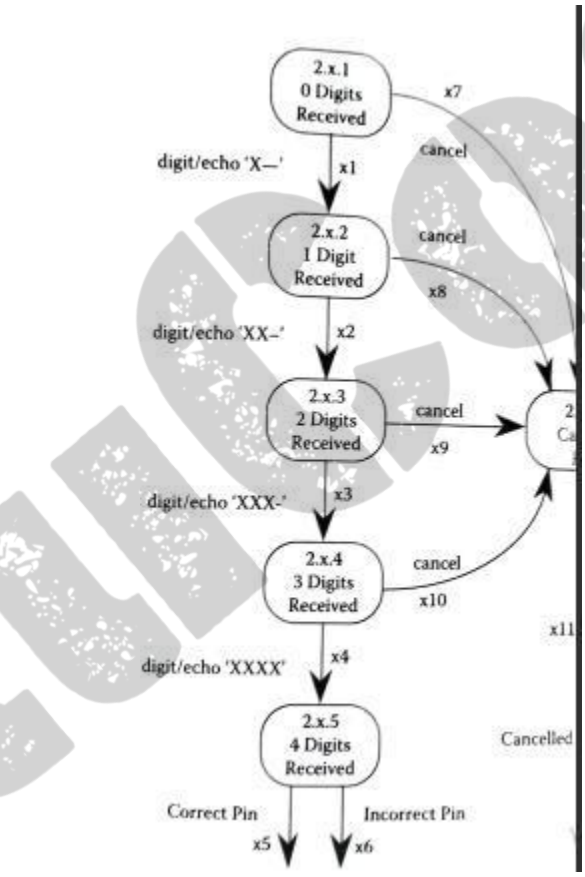


Fig: PIN try finite state machine

The hierarchy of finite state machines multiplies the number of threads. There are 156 distinct paths from the first PIN try state to the await transaction choice or card entry states in Figure distinct Of these, 31 correspond to eventually correct PIN entries (1 on the first try, 5 on the second try and 25 on the third try); the other 125 paths correspond to those

with incorrect digits or try cancel keystrokes. This is a fairly typical ratio. The input portion of systems, especially interactive systems, usually has a large number of threads to deal with input errors and exceptions.

Table 14.2 **Port Events in the PIN Try Finite State Machine**

Port Input Events	*Port Output Events*
Digit	echo "X- - -"
Cancel	echo "XX- -"
	echo "XXX-"
	echo "XXXX"

Table 14.3 **Port Event Sequence for Correct PIN on First Try**

Port Input Event	*Port Output Event*
	Screen 2 displayed with "- - - -"
1 pressed	Screen 2 displayed with "X- - -"
2 pressed	Screen 2 displayed with "XX- -"
3 pressed	Screen 2 displayed with "XXX-"
4 pressed	Screen 2 displayed with "XXXX"
(Correct PIN)	Screen 5 displayed

It is good form to reach a state machine in which transitions are caused by actual port input events, and it has actions on transitions are port output events. It is good factions on transitions are port output events if pre have such a finite state machine, generating system

test cases for these threads is a mechanical process – simply follow a path of transition and note the port inputs and outputs as they occur along the path.

This interleaved sequence is performed by the test executor (person or program). Table and Table 14.4 follow two paths through the hierarchic state machines. Table 14.3 correspond a thread in which a PIN is correctly entered on the first try. Table 14.4 corresponds to a threat which a PIN is incorrectly entered on the first try, canceled after the third digit on the second and correctly entered on the third try. To make the test case explicit, we assume a precondition that the expected PIIN is 1234.

The event in parentheses in the last row of Table 14.3 is the logical event that "bumps up" t parent state machine and causes a transition there to the await transaction choice state. If you look closely at Table 14.3 and Table 14.4, you will see that the bottom third of Table 144 is exactly Table 14.3; thus, a thread can be a subset of another thread.

14.4 Structural Strategies for Thread Testing

Although generating thread test cases is easy, deciding which ones to actually use is more complicated (If you have an automatic test executor, this is not a problem.) We have the same path explosion problem at the system level that we had at the unit level. Just as we did there, we can use t directed graph insights to make an intelligent choice of threads to test.

14.4.1 Bottom-Up Threads

When we organize state machines in a hierarchy, we can work from the bottom up. Six paths are used in the PIN try state machine. If we traverse these six, we test for three things: Correct recognition and echo of entered of entered digits, response to cancel keystroke, and matching expected and entered PINs.

Table 14.4 Port Event Sequence for Correct PIN on Third Try

Port Input Event	Port Output Event
	Screen 2 displayed with "– – – –"
1 pressed	
	Screen 2 displayed with "X– – –"
2 pressed	
	Screen 2 displayed with "XX– –"
3 pressed	
	Screen 2 displayed with "XXX–"
5 pressed	
	Screen 2 displayed with "XXXX"
(Incorrect PIN)	Screen 3 displayed
(Second try)	Screen 2 displayed with "– – – –"
1 pressed	
	Screen 2 displayed with "X– – –"
2 pressed	
	Screen 2 displayed with "XX– –"
3 pressed	
	Screen 2 displayed with "XXX–"
Cancel key pressed	
(End of second try)	Screen 3 displayed
	Screen 2 displayed with "– – – –"
1 pressed	
	Screen 2 displayed with "X– – –"
2 pressed	
	Screen 2 displayed with "XX– –"
3 pressed	
	Screen 2 displayed with "XXX–"
4 pressed	
	Screen 2 displayed with "XXXX"
(Correct PIN)	Screen 5 displayed

Table 14.5 Thread Paths in the PIN Try FSM

Input Event Sequence	*Path of Transitions*
1234	x1, x2, x3, x4, x5
1235	x1, x2, x3, x4, x6
C	x7, x11
1C	x1, x8, x11
12C	x1, x2, x9, x11
123C	x1, x2, x3, x10, x11

Once this portion is tested, we can go up a level to the PIN entry machine, where four patha are used. These four are concerned with the three-try mechanism and the sequence of screm presented to the user. In Table 14.6, the paths in the PIN entry state machine (Figure 14.6) are named as transition sequences.

These threads were identified with the goal of path traversal in mind. Recall from our discus sion of structural testing that these goals can be misleading. The assumption is that path traversal uncovers faults, and traversing a variety of paths reduces redundancy. The last path in Table 146 illustrates how structural goals can be counterproductive. Hitting the cancel key three times does indeed cause the three-try mechanism to fail and returns the system to the card entry state; buti seems like a degenerate thread. A more serious flaw occurs with these threads: we could not realls execute them alone because of the hierarchic state machines. What really happens with the 1235 input sequence in Table 14.5 ? It traverses an interesting path in the PIN try machine and then a "returns" to the PIN entry machine where it is seen as a logical event (incorrect PIN), which causes a transition to state 2.2 (second PIN try). If no additional keystrokes occur, this machine would remain in state 2.2. We show how to overcome such situations next.

14.4.2 Node and Edge Coverage Metrics

Because the finite state machines are directed graphs, we can use the same test coverage metrics that we applied at the unit level. The hierarchic relationship means that the upper-level machine must treat the lower machine as a procedure that is entered and returned. (Actually, we need to do

Table 14.6 Thread Paths in the PIN Entry FSM

Input Event Sequence	*Path of Transitions*
1234	1
12351234	2, 3
1235C1234	2, 4, 5
CCC	2, 4, 6

Table 14.7 **Node and Edge Traversal of a Thread**

Port Input Event	*Port Output Event*	*Nodes*	*Edges*
	Screen 2 displayed with "– – – –"	2.1	a
1 pressed		2.1.1	
	Screen 2 displayed with "X– – –"		x1
2 pressed		2.1.2	
	Screen 2 displayed with "XX– –"		x2
3 pressed		2.1.3	
	Screen 2 displayed with "XXX–"		x3
5 pressed		2.1.4	
	Screen 2 displayed with "XXXX"		x4
(Incorrect PIN)	Screen 3 displayed	2.1.5, 3	x6, 2
(Second try)	Screen 2 displayed with "– – – –"	2.2	
1 pressed		2.2.1	
	Screen 2 displayed with "X– – –"		x1
2 pressed		2.2.2	
	Screen 2 displayed with "XX– –"		x2
3 pressed		2.2.3	
	Screen 2 displayed with "XXX–"		x3
Cancel pressed		2.2.4	x10
(End of second try)	Screen 3 displayed	2.2.6	x11
	Screen 2 displayed with "– – – –"	2.3	4
1 pressed		2.3.1	
	Screen 2 displayed with "X– – –"		x1
2 pressed		2.3.2	
	Screen 2 displayed with "XX– –"		x2
3 pressed		2.3.3	
	Screen 2 displayed with "XXX–"		x3
4 pressed		2.3.4	
	Screen 2 displayed with "XXXX"		x4
(Correct PIN)	Screen 5 displayed	2.3.5, 3	x5, 5

Table 14.8 Thread/State Incidence

Input Events	2.1	2.x.1	2.x.2	2.x.3	2.x.4	2.x.5	2.2.6	2.2	2.3	3	1
1234	x	x	x	x	x	x				x	
12351234	x	x	x	x	x	x		x		x	
C1234	x	x	x	x	x	x	x	x		x	
1C12C1234	x	x	x	x			x	x	x	x	
123C1C1C	x	x	x	x	x		x	x	x		x

with a correct PIN. If you examine Table 14.8, you will see that two threads (initiated by C1234 and 123C1C1C) traverse all the states in both machines.

Edge (state transition) coverage is a more acceptable standard. If the state machines are well formed (transitions in terms of port events), edge coverage also guarantees port event coverage. The threads in Table 14.9 were picked in a structural way to guarantee that the less traveled edges (those caused by cancel keystrokes) are traversed.

Table 14.9 Thread/Transition Incidence

Input Events	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	1	2	3	4	5	6
1234	x	x	x	x	x												
12351234	x	x	x	x	x	x						x					
C1234	x	x	x	x	x		x						x	x			
1C12C1234	x	x	x	x	x			x	x		x		x	x			
123C1C1C	x	x	x					x		x	x		x		x	x	

14.5 Functional Strategies for Thread Testing

The finite state machine-based approaches to thread identification are clearly useful, but what no behavioral model exists for a system to be tested? The testing craftsperson has two

choices develop a behavioral model or resort to the system-level analogs of functional testing. Recall that when functional test cases are identified, we use information from the input and output spaces a well as the function itself. We describe functional threads here in terms of coverage metrics the are derived from three of the basis concepts (events, ports, and data).

14.5.1 Event-Based Thread Testing

Consider the space of port input events. Five port input thread coverage metrics are of interest. Attaining these levels of system test coverage requires a set of threads such that:

PII: Each port input event occurs

System Testing 245

P12: Common sequences of port input events occur

P13: Each port input event occurs in every relevant data context P14: For a given context, all inappropriate input events occure P15: For a given context, all possible input events occur

The Pll metric is a bare minimum and is inadequate for most systems. P12 coverage is the most common, and it corresponds, the intuitive view of system testing because it dealase is the most comtis difficult to quantify, however. What is a common sequence of input edeals with normal uncommon one?

The last three metrics are defined in terms of a context. The best view of a context is that it is a point of event quiescence. In the SATM system, screen displays occur at the point is that it quiescence. The P13 metric deals with context-sensitive port input events. These are physical input events that have logical meanings determined by the context within which they occur. impue SATM system, for example, a keystroke on the B1 function button

occurs in five separate coin the (screens displayed) and has three different meanings. The key to this metric is that it is driven by an event in all of its contexts. The P14 and P15 metrics are converses: they start with a conreven by seek a variety of events. The P14 metric is often used on an informal basis by testers who try to break a system. At a given context, they want to supply unanticipated input events just to see what happens. In the SATM system, for example, what happens if a function button is depressed during the PIN entry stage? The appropriate events are the digit and cancel keystrokes. The inappropriate input events are the keystrokes on the B1, B2, and B3 buttons.

This is partially a specification problem: we are discussing the difference berween prescribed behavior (things that should happen) and proscribed behavior (things that should not happen). Most requirements specifications have a hard time only describing prescribed behavior, it is usu ally testers who find proscribed behavior. The designer who maintains my local ATM system told me that once someone inserted a fish sandwich in the deposit envelope slot. (Apparently they thought it was a waste receptacle.) At any rate, no one at the bank ever anticipated insertion of a fish sandwich as a port input event. The P14 and PIS metrics are usually very effective, but they raise one

curious difficulty. How does the tester know what the expected response should be to a proscribed input? Are they simply ignored? Should there be an output warning message? Usually, this is left to the tester's intuition. If time permits, this is a powerful point of feedback to requirements specification. It is also a highly desirable focus for either rapid prototyping or executable specifications.

We can also define two coverage metrics based on port output events:

POI: Each port output event occurs PO2: Each port output event occurs for each cause

PO1 coverage is an acceptable minimum. It is particularly effective when a system has a rich variety of output messages sages for error conditions. (The SATM system does not.) PO2 coverage is a good goal, but it is hard to quantify; we will revisit this in Chapter 15 when we examine thread interaction. For now, note that PO2 coverage refers to threads that interact with respect to a port output event. Usually, a given output event only has a

small number of causes. In the SATM, screen 10 might be displayed for three reasons: the terminal might be out of cash, it may be impossible to make a connection with the central bank to get the account balance, or the withdrawal door might be jammed. In practice, some of the most difficult faults found in field trouble reports are those in which an output occurs for an unsuspected cause. Here is one example My bleat ATM system (not the SATM) has seen the info to withdraw daily withdrawal local ATMs This screen should occur when had made a more than $300 in day. Upon seeing this screen, I used to assume that feat heade major withdrawal (thre day. Upon sof requested a lesser amount, I found od of providing a lot o produces this sem interaction), sulf of cash in the dispenser is low. Instead of providing a lot of cash to the first wen the central bank prefers to provide less cash to more users.

14.5.2 Port-Based Thread Testing

Port-based testing is a useful complement to event-based testing. With port-based resting, we ask, for each port, what events can occur at that port. We then seek threads that exercise input ports and output ports with respect to the event lists for each port. (This presumes such event lists have been specified: some requirements specification techniques mandate such lists.) Port based testing is particularly useful for systems in which the port devices come from external suppliers. The main reason for port-based testing can be seen in the E/R model of the bam constructs (Figure 14.1). The many-to-many relationship between devices and events should be exercised in both directions. Event-based testing covers the one-to-many relationship from events to ports, and conversely, port-based testing covers the one-to-many relationship from ports to events. The SATM system fails us at this point no SATM event occurs at more than one port.

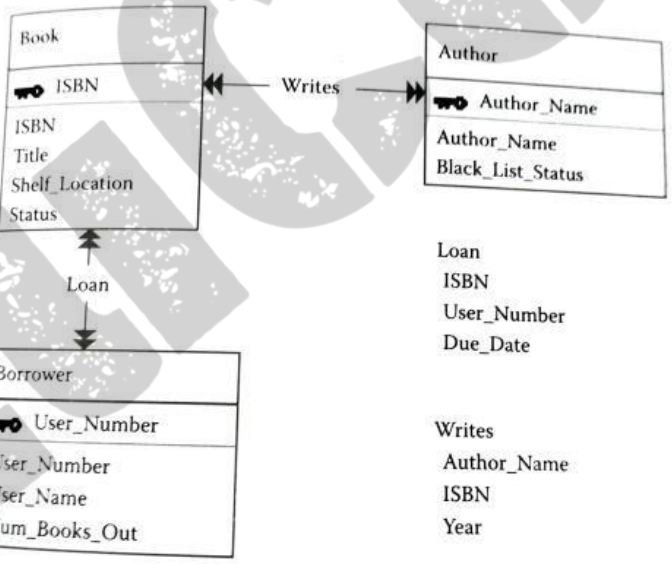
14.5.3 Data-Based Thread Testing

Port- and event-based testing work well for systems that are primarily event driven. Such systema are sometimes called reactive systems because they react to stimuli (port input events), and often the reaction is in the form of port output events. Reactive systems have two important characteristics: they are long running (as opposed to the short burst of computation we see in a payroll program) and they maintain a relationship with their environment. Typically, event-driven, respective systems do not have a very interesting

data model (as we see with the SATM system), so data model-based threads are not particularly useful. So, what about conventional systems that are data driven? These systems, described as "static" in Topper (1993), are transformational (instead of reactive); they support transactions on a database. When these systems are specified, the ER model is dominant and is therefore a fertile source of system testing threads. To attach our discussion to something familiar, we use the E/R model of a simple library system. See Figure 14.8 from Topper (1993).

Here are some typical transactions in the library system:

1. Add a book to the library.
2. Delete a book from the library.
3. Add a borrower to the library. 3
4. Delete a borrower from the library.
5. Loan a book to a borrower.
6. Process the return of a book from a borrower.



These transactions are all mainline threads; in fact, they represent families of threads. For sample, suppose the book loan transaction is attempted for a borrower whose current number checked-out books is at the lending limit (a nice boundary value example) We might also try to return a book that was never owned by the library. Here is one more:

suppose we delete borrower that has some unreturned books. All are interesting threads to test, and all are at the system level.

We can identify each of these examples, and many more, by close attention to the information the entity/relationship model. As we did with event-based testing, we describe sets of threads in terms of data-based coverage metrics. These refer to relationships for an important reason. Information in relationships is generally populated by system-level threads, whereas that in the entities usually handled at the unit level. (When E/R modeling is the starting point of object-oriented analysis, this is enforced by encapsulation.)

DM1: Exercise the cardinality of every relationship
DM2: Exercise the participation of every relationship
DM3: Exercise the functional dependencies among relationships

Cardinality refers to the four possibilities of relationship that we discussed in Chapter 3: one- none, one-to-many, many-to-one, and many-to-many In the library example, both the loan and the writes relationships are many-to-many, meaning that one author can write many books, and one book can have many authors; and that one book can be loaned to many borrowers (in sequence), and one borrower can borrow many books. Each of these possibilities results in a useful system testing thread.

Participation refers to whether every instance of an entity participates in a relationship. In the writes relationship, both the book and the author entities have mandatory participation (we met have a book with no authors, or an author of no books) In some modeling techniques. participation is expressed in terms of numerical limits, the author entity, for example, might be.

5.6 SATM Test Threads

If we apply the discussion of this chapter to the SATM system, we get a set of threads that constitutes a thorough system level test. We develop such a set of threads here in terms of an overall state model in which states correspond to key atomic system functions. The macro-level states are: Card Entry, PIN Entry, Transaction Request, (and processing), and Session Management. The stated order is the testing order, because these stages are in prerequisite order. (We cannot enter a PIN until successful card entry, we cannot request a transaction until successful PIN entry, and so on.) We also need some pre-condition data that define some actual accounts with PANs, Expected PINs, and account balances. These are given in Table 10. Two less obvious pre-conditions are that the ATM terminal is initially displaying screen 1 and the total cash available to the withdrawal dispenser is $500 (in $10 notes).

Table 10 SATM Test Data **PAN**

	Expected PIN	Checking Balance	Savings Balance
100	1234	$1000.00	$800.00
200	4567	$100.00	$90.00
300	6789	$25.00	$20.00

We will express threads in tables in which pairs of rows correspond to port inputs and expected port outputs at each of the four major stages. We start with three basic threads, one for each transaction type (balance inquiry, deposit, and withdrawal).

Thread 1 (balance)	Card Entry (PAN)	PIN Entry	Transaction Request	Session Management
Port Inputs	100	1234	B1, B1	B2
Port Outputs	screen 2	screen 5	screen 6, screen 14 $1000.00	screen 15, eject card, screen 1

In thread 1, a valid card with PAN = 100 is entered, which causes screen 2 to be displayed. The PIN digits '1234' are entered, and since they match the expected PIN for the PAN, screen 5 inviting a transaction selection is displayed. When button B1 is touched the first time (requesting a balance inquiry), screen 6 asking which account is displayed. When B1 is pressed the second time (checking), screen 14 is displayed and the checking account balance ($1000.00) is printed on the receipt. When B2 is pushed, screen 15 is displayed, the receipt is printed, the ATM card is ejected, and then screen 1 is displayed.

Thread 2 is a deposit to checking: Same PAN and PIN, but B2 is touched when screen 5 is displayed, and B1 is touched when screen 6 is displayed. The amount 25.00 is entered when screen 7 is displayed and then screen 13 is displayed. The deposit door opens and the deposit envelope is placed in the deposit slot. Screen 14 is displayed, and when B2 is pushed, screen 15 is displayed, the receipt showing the new checking account balance of $1025.00 is printed, the ATM card is ejected, and then screen 1 is displayed.

Thread 2 (deposit)	Card Entry (PAN)	PIN Entry	Transaction Request	Session Management
Port Inputs	100	1234	B2, B1, 25.00 insert env.	B2
Port Outputs	screen 2	screen 5	screen 6, screen 7, screen 13, dep. door opens, screen 14, $1025.00	screen 15, eject card, screen 1

Thread 3 is a withdrawal from savings: Again the same PAN and PIN, but B3 is touched when screen 5 is displayed, and B2 is touched when screen 6 is displayed. The amount 30.00 is entered when screen 7 is displayed and then screen 11 is displayed. The withdrawal door opens and three $10 notes are dispensed. Screen 14 is displayed, and when B2 is pushed, screen 15 is displayed, the

receipt showing the new savings account balance of $770.00 is printed, the ATM card is ejected, and then screen 1 is displayed.

Thread (withdrawal) 3	Card Entry (PAN)	PIN Entry	Transaction Request	Session Management
Port Inputs	100	1234	B3, B2, 30.00	B2
Port Outputs	screen 2	screen 5	screen 6, screen 7, screen 11, withdrawal door opens, 3 $10 notes, screen 14, $770.00	screen 15, eject card, screen 1

A few of these detailed descriptions are needed to show the pattern; the remaining threads are described in terms of input and output events that are the objective of the test thread.

Thread 4 is the shortest thread in the SATM system, it consists of an invalid card, which is immediately rejected.

Thread 4	Card Entry (PAN)	PIN Entry	Transaction Request	Session Management
Port Inputs	400			
Port Outputs	eject card screen 1			

A few of these detailed descriptions are needed to show the pattern; the remaining threads are described in terms of input and output events that are the objective of the test thread.

Thread 4 is the shortest thread in the SATM system, it consists of an invalid card, which is immediately rejected.

Thread 4	Card Entry (PAN)	PIN Entry	Transaction Request	Session Management
Port Inputs	400			
Port Outputs	eject card screen 1			

Following the macro-states along thread 1, we next perform variations on PIN Entry. We get four new threads from Table 9, which yield edge coverage in the PIN Entry finite state machines.

Thread 5 (balance)	Card Entry (PAN)	PIN Entry	Transaction Request	Session Management
Port Inputs	100	12351234	as in thread 1	
Port Outputs	screen 2	screens 3,2,5		

Thread 6 (balance)	Card Entry (PAN)	PIN Entry	Transaction Request	Session Management
Port Inputs	100	C1234	as in thread 1	
Port Outputs	screen 2	screens 3,2,5		

Thread 7 (balance)	Card Entry (PAN)	PIN Entry	Transaction Request	Session Management
Port Inputs	100	1C12C1234	as in thread 1	
Port Outputs	screen 2	screens 3,2, 3,2,5		
Thread 8 (balance)	**Card Entry (PAN)**	**PIN Entry**	**Transaction Request**	**Session Management**
Port Inputs	100	123C1C1C		
Port Outputs	screen 2	screens 3,2, 3,2,4,1		

Moving to the Transaction Request stage, there are variations with respect to the type of transaction (balance, deposit, or withdraw), the account (checking or savings) and several that deal with the amount requested. Threads 1, 2, and 3 cover the type and account variations, so we focus on the amount-driven threads. Thread 9 rejects the attempt to withdraw an amount not in $10 increments, Thread 10 rejects the attempt to withdraw more than the account balance, and Thread 11 rejects the attempt to withdraw more cash than the dispenser contains.

Thread (withdrawal)	9	Card Entry (PAN)	PIN Entry	Transaction Request	Session Management
Port Inputs		100	1234	B3, B2, 15.00 Cancel	B2
Port Outputs		screen 2	screen 5	screens 6,7, 9, 7	screen 15, eject card, screen 1

Thread (withdrawal)	10	Card Entry (PAN)	PIN Entry	Transaction Request	Session Management
Port Inputs		300	6789	B3, B2, 50.00 Cancel	B2
Port Outputs		screen 2	screen 5	screens 6,7,8	screen 15, eject card, screen 1

Thread (withdrawal)	11	Card Entry (PAN)	PIN Entry	Transaction Request	Session Management
Port Inputs		100	1234	B3, B2, 510.00 Cancel	B2

Port Outputs	screen 2	screen 5	screens 6,7, 10	screen 15, eject card, screen 1

Having exercised the transaction processing portion, we proceed to the session management stage, where we test the multiple transaction option.

Thread 12 (balance)	Card Entry (PAN)	PIN Entry	Transaction Request	Session Management
Port Inputs	100	1234	B1, B1	B1, Cancel
Port Outputs	screen 2	screen 5	screen 6, screen 14 $1000.00	screen 15, screen 5, screen 15, eject card, screen 1

At this point, the threads provide coverage of all output screens except for screen 12, which informs the user that deposits cannot be processed. Causing this condition is problematic (maybe we should place a fish sandwich in the deposit envelope slot). This is an example of a thread selected by a pre-condition that is a hardware failure. We just give it a thread name here, it's thread 13. Next, we develop threads 14 through 22 to exercise context sensitive input events. They are shown in Table 11; notice that some of the first 13 threads exercise context sensitivity.

Table 11 Threads for Context Sensitive Input Events **Thread**

	Keystroke	Screen	Logical Meaning
6	cancel	2	PIN Entry error
14	cancel	5	transaction selection error
15	cancel	6	account selection error
16	cancel	7	amount selection error
17	cancel	8	amount selection error
18	cancel	13	deposit envelope not ready
1	B1	5	balance
1	B1	6	checking
19	B1	10	yes (a non-withdrawal transaction)

20	B1	12	yes (a non-deposit transaction)
12	B1	14	yes (another transaction)
2	B2	5	deposit
3	B2	6	savings
21	B2	10	no (no additional transaction)
22	B2	12	no (no additional transaction)
1	B2	14	no (no additional transaction)

These 22 threads comprise a reasonable test of the portion of the SATM system that we have specified. Of course there are untested aspects; one good example involves the balance of an account. Consider two threads, one that deposits $40 to an account, and a second that withdraws $80, and suppose that the balance obtained form the central bank at the Card Entry stage is $50. There are two possibilities: one is to use the central bank balance, record all transactions, and then resolve these when the daily posting occurs. The other is to maintain a running local balance, which is what would be shown on a balance inquiry transaction. If the central bank balance is used, the withdrawal transaction is rejected, but if the local balance is used, it is processed.

Another prominent untested portion of the SATM system is the Amount Entry process that occurs in screens 7 and 8. The possibility of a cancel keystroke at any point during amount entry produces a multiplicity greater than that of PIN Entry. There is a more subtle (and therefore more interesting) test for Amount Entry. What actually happens when we enter an amount? To be specific, suppose we wish to enter $40.00. We expect an echo after each digit keystroke, but in which position does the echo occur? Two obvious solutions: always require six digits to be entered (so we would enter `004000`) or use the high order digits first and shift left as successive digits are entered, as shown in Figure 14.10. Most ATM systems use the shift approach, and this raises the subtle point: how does the ATM system know when all amount digits have been entered? The ATM system clearly cannot predict that the deposit amount is $40.00 instead of $400.00 or $4000.000 because there is no "enter" key to signify when the last digit has been entered. The reason for this digression is that this is a good example of the kind of detail discovered by testers that is often missing from a requirements specification. (Such details would likely be found with either Rapid Prototyping or using an executable specification.)

5.7 System Testing Guidelines

If we disallow compound sessions (more than one transaction) and if we disregard the multiplicity due to Amount Entry possibilities, there are 435 distinct threads per valid account in the SATM system. Factor in the effects of compound sessions and the Amount Entry possibilities and there are tens of thousands of possible threads for the SATM system. We end this chapter with three strategies to deal with the thread explosion problem.

5.7.1 Pseudo-Structural System Testing

When we studied unit testing, we saw that the combination of functional and structural testing yields a desirable cross-check. We can only claim pseudo-structural testing [Jorgensen 94], because the node and edge coverage metrics are defined in terms of a control model of a system, and are not derived directly from the system implementation. (Recall we started out with a concern over the distinction between reality and models of reality.) In general, behavioral models are only approximations of a system's reality, which is why we could decompose our models down to several levels of detail. If we made a true structural model, its size and complexity would make it too cumbersome to use. The big weakness of pseudo-structural metrics is that the underlying model may be a poor choice. The three most common behavioral models (decision tables, finite state machines, and Petri nets) are appropriate, respectively, to transformational, interactive, and concurrent systems. Decision tables and finite state machines are good choices for ASF testing. If an ASF is described using a decision table, conditions typically include port input events, and actions are port output events. We can then devise test cases that cover every condition, every action, or most completely, every rule. As we saw for finite state machine models, test cases can cover every state, every transition, or every path.

Thread testing based on decision tables is cumbersome. We might describe threads as sequences of rules from different decision tables, but this becomes very messy to track in terms of coverage. We need finite state machines as a minimum, and if there is any form of interaction, Petri nets are a better choice. There we can devise thread tests that cover every place, every transition, and every sequence of transitions.

Chapter 2: Interaction Testing

Faults and failures due to interaction are the bane of testers. Their subtleties make them difficult to recognize and even more difficult to reveal by testing. These are deep faults, ones that remain in a system even after extensive thread testing. Unfortunately, faults of interaction most frequently occur as failures in delivered systems that have been in use for some time. Typically, they have a very low probability of execution, and they occur only after a large number of threads has been executed. Most of this chapter is devoted to describing forms of interaction, not to testing them. As such, it is really more concerned with requirements specification than with testing. The connection is important: knowing how to specify interactions is the first step in detecting and testing for them. This chapter is also a somewhat philosophical and mildly mathematical discussion of faults and failures of interaction, we cannot hope to test something if we do not understand it. We begin with an important addition to our five basic constructs and use this to develop a taxonomy of types of interaction. Next, we develop a simple extension to conventional Petri nets that reflects the basic constructs, and then we illustrate the whole discussion with the simple automated teller machine (SATM) and Saturn windshield wiper systems, and sometimes with examples from telephone systems. We conclude by applying the taxonomy to an important application type: client/server systems.

15.1 Context of Interaction

Part of the difficulty of specifying and testing interactions is that they are so common. Think of all the things that interact in everyday life: people, automobile drivers, regulations, chemical compounds, and abstractions, to name just a few. We are concerned with interactions in software- controlled systems (particularly the unexpected ones), so we start by restricting our discussion to interactions among our basic system constructs: actions, data, events, ports, and threads.

One way to establish a context for interaction is to view it as a relationship among the five constructs. If we did this, we would find that the relation Interacts With is a reflexive relation- ship on each entity (data interact with data, actions with other actions, and so on). It also is a binary relationship between data and events, data and threads, and events and threads. The data modeling approach is not a dead end, however. Whenever a data model contains such pervasive relationships, that is a clue that an important entity is missing. If we add some tangible reality to our fairly abstract constructs, we get a more useful framework for our study of interaction. The missing element is location, and location has two components: time and position. Data modeling provides another choice: we can treat

location as a sixth basic entity or as an attribute of the other five. We choose the attribute approach here

What does it mean for location (time and position) to be an attribute of any of the five basic constructs? This is really a shortcoming of nearly all requirements, specification notations, and techniques. (This is probably also the reason that interactions are seldom recognized and tested.) Information about location is usually created when a system is implemented. Sometimes location is mandated as a requirement when this happens, the requirement is actually a forced implementation choice. We first clarify the meaning of the attributes of location: time and position.

We can take two views of time: as an instant or as an interval. The instantaneous view lets us describe when something happens it is a point on the time axis. The duration view is an interval on the time axis. When we think about durations, we usually are interested in the length of the time interval, not the endpoints (the start and finish times). Both views are useful. Because threads execute, they have duration, and they also have points in time when they execute. Similar observations apply to events. Often, events have very short durations, and this is problematic if the duration is so short that the event is not recognized by the system.

The position aspect is easier. We could take a very tangible, physical view of position and describe it in terms of some coordinate system. Position can be a three-dimensional Cartesian coordinate system with respect to some origin, or it could be a longitude-latitude-elevation geo- graphic point. For most systems, it is more helpful to slightly abstract position into processor residence. Taken together, time and position tell the tester when and where something happens, and this is essential to understanding interactions.

Before we develop our taxonomy, we need some ground rules about threads and processors. For now, a processor is something that executes threads or a device where events occur

1. Because threads execute, they have a strictly positive time duration. We usually speak of the execution time of a thread, but we might also be interested in when thread execution begins.

Actions are degenerate cases of threads; therefore, actions also have durations. 2. In a single processor, two threads cannot execute simultaneously. This resembles a fundamental

precept of physics: no two bodies may occupy the same space at the same time. Sometimes threads appear to be simultaneous, as in time-sharing on a single processor:

in fact, time-shared threads are interleaved. Even though threads cannot execute simultaneously on a single processor, events can be simultaneous. (This is really problematic for testers.)

3. Events have a strictly positive time duration. When we consider events to be actions that execute on port devices, this reduces to the first ground rule.

4. Two (or more) input events can occur simultaneously, but an event cannot occur simultaneously in two (or more) processors. This is immediately clear if we consider port devices to be separate processors.

5. In a single processor, two output events cannot begin simultaneously. This is a direct con- sequence of output events being caused by thread executions. We need both the instantaneous and duration views of time to fully explain this ground rule. Suppose two output events are such that the duration of one is much greater than the duration of the other. The durations may overlap (because they occur on separate devices), but the start times cannot be identical, as shown in Figure 15.1. An example of this occurs in the SATM system, when a thread causes screen 15 to be displayed and then ejects the ATM card. The screen is still displayed when the card eject event occurs. (This may be a fine distinction; we could also say that port devices are separate processors, and that port output events are really a form of interprocessor communication.)

6. A thread cannot span more than one processor. This convention helps in the definition of threads. By confining a thread to a single processor, we create a natural endpoint for threads, this also results in more simple threads instead of fewer complex threads. In a multi- processing setting, this choice also results in another form of quiescence-transprocessor quiescence.

Taken together, these six ground rules force what we might call sane behavior onto the interac tions in the taxonomy we define in Section 15.2.

15.2 A Taxonomy of Interactions

The two aspects of location, time and position, form the starting point of a useful taxonomy of interaction. Certain interactions are completely independent of time; for example, two

data items that interact exhibit their interaction regardless of time. Certain time-dependent interactions also occur, such as when something is a prerequisite for something else. We will refer to time- independent interactions as static and time-dependent interactions as dynamic. We can refine the static/dynamic dichotomy with the distinction between single and multiple processors. These two considerations yield a two-dimensional plane (as shown in Figure 15.2) with four basic types of interactions:

- Static interactions in a single processor
- Static interactions in multiple processors
- Dynamic interactions in a single processor
- Dynamic interactions in multiple processors

5.9 Interaction, Composition, and Determinism

The question of non-determinism looms as a backdrop to deep questions in science and philosophy. Einstein didn't believe in non-determinism; he once commented that he doubted that God would play dice with the universe. Non-determinism generally refers to consequences of random events, asking in effect, if there are truly random events (inputs), can we ever predict their consequences? The logical extreme of this debate ends in the philosophical/theological question of free will versus pre-destination. Fortunately, for testers, the software version of non-determinism is less severe. You might want to consider this section to be a technical editorial. It is based on my experience and analysis using the OSD framework. I find it yields reasonable answers to the problem of non-determinism; you may too.

Let's start with a working definition if determinism; here are two possibilities:

1. A system is deterministic if, given its inputs, we can always predict its outputs.
2. A system is deterministic if it always produces the same outputs for a given set of inputs.

Since the second view (repeatable outputs) is less stringent than the first (predictable outputs), we'll use it as our working definition. Then a non-deterministic system is one in which there is at least one set of inputs that results in two distinct sets of outputs. It's easy to devise a non-deterministic finite state machine; Figure 16.11 is one example.

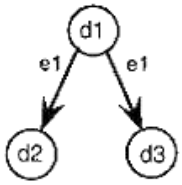


Figure 5.11 A Non-deterministic Finite State Machine

When the machine in Figure 5.11 is in state d1, if event e1 occurs, there is a transition either to state d2 or to d3.

If it is so easy to create a non-deterministic finite state machine, why all the fuss about determinism in the first place? (It turns out that we can always find a deterministic equivalent to any non-deterministic finite state machine anyway.) Finite state machines are models of reality; they only approximate the behavior of a real system. This is why it is so important to choose an appropriate model we would like to use the best approximation. Roughly speaking, decision tables are the mode l of choice for static interactions, finite state machines suffice for dynamic interactions in a single processor, and some form of Petri net is needed for dynamic interactions in multiple processors. Before going on, we should indicate instances of non-determinism in the other two models. A multiple hit decision table is one in which the inputs (variables in the condition stub) are such that more than one rule is selected. In Petri nets, non-determinism occurs when more than one transition is enabled. The choice of which rule executes or which transition fires is made by an external agent. (Notice that the choice is actually an input!)

Our question of non-determinism reduces to threads in an OSD net, and this is where interactions, composition, and determinism come together. To ground our discussion in something "real", consider the SATM threads we used earlier:

T1: withdraw $40.00

T2: withdraw $60.00

T3: deposit $30.00

Threads T1, T2, and T3 interact via a data place for the account balance, and they may be executed in different processors. The initial balance is $50.00.

Begin with thread TI; if no other thread executes, it will execute correctly, leaving a balance of $10.00. Suppose we began with thread T2; we should really call it "attempt to withdraw $60.00", because, if no other thread executes, it will result in the insufficient funds screen. We should really separate T2 into two threads, T2.1 which is a successful withdrawal that ends with the display of screen 11 (take cash), and T2.2 which is a failed withdrawal that ends with the display of screen 8 (insufficient funds). Now let's add some interaction with thread T3. Threads T2 and T3 are 2-connected via the balance data place. If T3 executes before T2 reads the balance data, then T2.1 occurs, otherwise T2.2 occurs. The difference between the two views of determinism is visible here: When the OSD net of T2 begins to execute, we cannot predict the outcome (T2.1 or T2.2), so by the first definition, this is non-deterministic. By the second definition, however, we can recreate the interaction (including times) between T2 and T2. If we do, and we capture the behavior as a marking of the composite OSD net, we will satisfy the repeatable definition of determinism.

5.11 Client-Server Testing

Client-server systems are difficult to test because they exhibit the most difficult form of interactions, the dynamic ones across multiple processors. Here we can enjoy the benefits of our strong theoretical development. Client-server systems always entail at least two processors, one where the server software exists and executes, and one (usually several) where the client software executes. The main components are usually a database management system, application programs that use the database, and presentation programs that produce user-defined output. The position of these components results in the fat server vs. fat client distinction [Lewis 94] (see Figure 5.12).

Figure 5.12 Fat Clients and Servers

Client-server systems also include a network to connect the clients with the server, the network software, and a graphical user interface (GUI) for the clients. To make matters worse, we can differentiate homogeneous and heterogeneous CS systems in terms of client processors that are identical or diverse. The multiple terminal version of the SATM system would be a fat client system, since the central bank does very little of the transaction processing.