Chapter 14

Model-based Testing

Models are often used to express requirements, and embed both structure and fault information that can help generate test case specifications. Control flow and data flow testing are based on models extracted from program code. Models can also be extracted from specifications and design, allowing us to make use of additional information about intended behavior. Model based testing consists in using or deriving models of expected behavior to produce test case specifications that can reveal discrepancies between actual program behavior and the model.

Required Background

- Chapter 10:
  The rationale of systematic approaches to functional testing is a key motivation for the techniques presented in this chapter.

- Chapters 12 and 13:
  The material on control and data flow graphs is required to understand Section 14.4, but it is not necessary to comprehend the rest of the chapter.

14.1 Overview

Combinatorial approaches to specification-based testing (Chapter 11) primarily select combinations of orthogonal choices. They can accommodate constraints among choices, but their strength is in systematically distributing combinations of (purportedly) independent choices. The human effort in applying those techniques is primarily in characterizing the elements to be combined and constraints on their combination, often starting from informal or semi-structured specifications.

Specifications with more structure can be exploited to help test designers identify input elements, constraints, and significant combinations. The structure may be explicit and available in a specification, e.g., in the form of a finite state machine or grammar. It may be derivable from a semi-formal model, such as class and object diagrams,
with some guidance by the designer. Even if the specification is expressed in natural language, it may be worthwhile for the test designer to manually derive one or more models from it, to make the structure explicit and suitable for automatic derivation of test case specifications.

Models can be expressed in many ways. Formal models, e.g., finite state machines or grammars, provide enough information to allow one to automatically generate test cases. Semi-formal models, e.g., class and object diagrams, may require some human judgment to generate test cases. This chapter discusses some of the most common models used to express requirements specifications. Models used for object oriented design are discussed in Chapter 15.

Models can provide two kinds of help. They describe the structure of the input space and thus allow test designers to take advantage of work done in software requirements analysis and design. Moreover, discrepancies from the model can be used as an implicit fault model to help identify boundary and error cases.

The utility of models for generating test cases is an important factor in determining the cost-effectiveness of producing formal or semi-formal specifications. The return on investment for model building should be evaluated not only in terms of reduced specification errors and avoided misinterpretation, but also improved effectiveness and reduced effort and cost in test design.

### 14.2 Deriving Test Cases from Finite State Machines

Finite state machines are often used to specify sequences of interactions between a system and its environment. State machine specifications in one form or another are common for control and reactive systems, such as embedded systems, communication protocols, menu driven applications, and threads of control in a system with multiple threads or processes.

Specifications may be expressed directly as some form of finite-state machine. For example, embedded control systems are frequently specified with Statecharts, communication protocols are commonly described with SDL diagrams, and menu driven applications are sometimes modeled with simple diagrams representing states and transitions.

Sometimes the finite state essence of systems is left implicit in informal specifications. For instance, the informal specification of feature *Maintenance* of the Chipmuk web site given in Figure 14.1 describes a set of interactions between the maintenance system and its environment that can be modeled as transitions through a finite set of process states. The finite state nature of the interaction is made explicit by the finite state machine shown in Figure 14.2. Note that some transitions appear to be labeled by conditions rather than events, but they can be interpreted as shorthand for an event in which the condition becomes true or is discovered (e.g., “lack component” is shorthand for “discover that a required component is not in stock”).

Many control or interactive systems have a potentially infinite set of states. Fortunately, the non-finite-state parts of the specification are often simple enough that finite state machines remain a useful model for testing as well as specification. For example, communication protocols are frequently specified using finite state machines, often
Maintenance: The Maintenance function records the history of items undergoing maintenance.

If the product is covered by warranty or maintenance contract, maintenance can be requested either by calling the maintenance toll free number, or through the web site, or by bringing the item to a designated maintenance station.

If the maintenance is requested by phone or web site and the customer is a US or EU resident, the item is picked up at the customer site, otherwise, the customer shall ship the item with an express courier.

If the maintenance contract number provided by the customer is not valid, the item follows the procedure for items not covered by warranty.

If the product is not covered by warranty or maintenance contract, maintenance can be requested only by bringing the item to a maintenance station. The maintenance station informs the customer of the estimated costs for repair. Maintenance starts only when the customer accepts the estimate. If the customer does not accept the estimate, the product is returned to the customer.

Small problems can be repaired directly at the maintenance station. If the maintenance station cannot solve the problem, the product is sent to the maintenance regional headquarters (if in US or EU) or to the maintenance main headquarters (otherwise).

If the maintenance regional headquarters cannot solve the problem, the product is sent to the maintenance main headquarters.

Maintenance is suspended if some components are not available.

Once repaired, the product is returned to the customer.

Figure 14.1: The functional specification of feature Maintenance of the Chipmuk web site.
Figure 14.2: A finite state machine corresponding to functionality *Maintenance* specified in Figure 14.1
Deriving Test Cases from Finite State Machines

T-Cover

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC-1</td>
<td>0 → 2 → 4 → 1 → 0</td>
</tr>
<tr>
<td>TC-2</td>
<td>0 → 5 → 2 → 4 → 5 → 6 → 0</td>
</tr>
<tr>
<td>TC-3</td>
<td>0 → 3 → 5 → 9 → 6 → 0</td>
</tr>
<tr>
<td>TC-4</td>
<td>0 → 3 → 5 → 7 → 5 → 8 → 7 → 8 → 9 → 7 → 9 → 6 → 0</td>
</tr>
</tbody>
</table>

Table 14.1: A test suite satisfying the transition coverage criterion with respect to the finite state machine of Figure 14.2

Table 14.1: A test suite satisfying the transition coverage criterion with respect to the finite state machine of Figure 14.2

with some extensions that make them not truly finite-state. Even a state machine that simply receives a message on one port and then sends the same message on another port is not really finite-state unless the set of possible messages is finite, but is often rendered as a finite state machine, ignoring the contents of the exchanged messages.

State-machine specifications can be used both to guide test selection and in construction of an oracle that judges whether each observed behavior is correct. There are many approaches for generating test cases from finite state machines, but most are variations on a basic strategy of checking each state transition. One way to understand this basic strategy is to consider that each transition is essentially a specification of a precondition and postcondition, e.g., a transition from state \( S \) to state \( T \) on stimulus \( i \) means "if the system is in state \( S \) and receives stimulus \( i \), then after reacting it will be in state \( T \)." For instance, the transition labeled accept estimate from state Wait for acceptance to state Repair (maintenance station) of Figure 14.2 indicates that if an item is on hold waiting for the customer to accept an estimate of repair costs, and the customer accepts the estimate, then the item is designated as eligible for repair.

A faulty system could violate any of these precondition, postcondition pairs, so each should be tested. For example, the state Repair (maintenance station) can be arrived at through three different transitions, and each should be checked.

Details of the approach taken depend on several factors, including whether system states are directly observable or must be inferred from stimulus/response sequences, whether the state machine specification is complete as given or includes additional, implicit transitions, and whether the size of the (possibly augmented) state machine is modest or very large.

The transition coverage criterion requires each transition in a finite state model to be traversed at least once. Test case specifications for transition coverage are often given as state sequences or transition sequences. For example, the test suite T-Cover in Table 14.1 is a set of four paths, each beginning at the initial state, which together cover all transitions of the finite state machine of Figure 14.2. T-Cover thus satisfies the transition coverage criterion.

The transition coverage criterion depends on the assumption that the finite-state machine model is a sufficient representation of all the "important" state, e.g., that transitions out of a state do not depend on how one reached that state. Although it can be considered a logical flaw, in practice one often finds state machines that exhibit "history sensitivity," i.e., the transitions from a state depend on the path by which one reached that state. For example, in Figure 14.2, the transition taken from state Wait for com-

Draft version produced February 22, 2006
ponent when the component becomes available depends on how the state was entered. This is a flaw in the model — there really should be three distinct \textit{Wait for component} states, each with a well-defined action when the component becomes available. However, sometimes it is more expedient to work with a flawed state-machine model than to repair it, and in that case test suites may be based on more than the simple transition coverage criterion.

\begin{quote}
Michal: The criteria here are not quite right — we actually allow abbbc as a representative of abbc. Reword appropriately.
\end{quote}

Coverage criteria designed to cope with history sensitivity include \textit{single state path coverage}, \textit{single transition path coverage}, and \textit{boundary interior loop coverage}. Each of these criteria requires execution of paths that include certain sub-paths in the FSM. The \textit{single state path coverage} criterion requires each path that traverses states at most once to be exercised. The \textit{single transition path coverage} criterion requires each path that traverses transitions at most once to be exercised. The \textit{boundary interior loop coverage} criterion requires each distinct loop of the state machine to be exercised the minimum, an intermediate, and the maximum or a large number of times\footnote{The boundary interior path coverage was originally proposed for structural coverage of program control flow, and is described in Chapter 12}. These criteria may be practical for very small and simple finite-state machine specifications, but since the number of even simple paths (without repeating states) can grow exponentially with the number of states, they are often impractical.

Specifications given as finite-state machines are typically incomplete, i.e., they do not include a transition for every possible (state, stimulus) pair. Often the missing transitions are implicitly error cases. Depending on the system, the appropriate interpretation may be that these are \textit{don’t care} transitions (since no transition is specified, the system may do anything or nothing), \textit{self} transitions (since no transition is specified, the system should remain in the same state), or (most commonly) \textit{error} transitions that enter a distinguished state and possibly trigger some error handling procedure. In at least the latter two cases, thorough testing includes the implicit as well as the explicit state transitions. No special techniques are required: The implicit transitions are simply added to the representation before test cases are selected.

The presence of implicit transitions with a \textit{don’t care} interpretation is typically an implicit or explicit statement that those transitions are impossible, e.g., because of physical constraints. For example, in the specification of the maintenance procedure of Figure 14.2, the effect of event \textit{lack of component} is specified only for the states that represent repairs in progress, because only in those states might we discover a needed is missing.

Sometimes it is possible to test \textit{don’t care} transitions even if they are believed to be impossible in the fielded system, because the system does not prevent the triggering event from occurring in a test configuration. If it is not possible to produce test cases for the \textit{don’t care} transitions, then it may be appropriate to pass them to other validation or verification activities, for example, by including explicit assumptions in a requirements or specification document that will undergo inspection.
Terminology: Predicates and Conditions

A predicate is a function with a boolean (True or False) value. When the input argument of the predicate is clear, particularly when it describes some property of the input of a program, we often leave it implicit. For example, the actual representation of account types in an information system might be as three-letter codes, but in a specification we may not be concerned with that representation — we know only that there is some predicate educational-account which is either True or False.

A basic condition is a single predicate that cannot be decomposed further.

A complex condition is made up of basic conditions, combined with boolean connectives.

The boolean connectives include “and” (\(\land\)), “or” (\(\lor\)), “not” (\(\neg\)), and several less common derived connectives such as “implies” and “exclusive or.”

14.3 Testing Decision Structures

Specifications are often expressed as decision structures, i.e., sets of conditions on input values and corresponding actions or results. A model of the decision structure can be used to choose test cases that may reveal discrepancies between the decisions actually made in the code and the intended decision structure.

The example specification of Figure 14.3 describes outputs that depend on type of account (either educational, or business, or individual), amount of current and yearly purchases, and availability of special prices. These can be considered as boolean conditions, e.g., the condition educational account is either true or false (even if the type of account is actually represented in some other manner). Outputs can be described as boolean expressions over the inputs, e.g., the output no discount can be associated with the boolean expression

\[
\begin{align*}
\text{(individual account)} & \land \neg \text{current purchase} > \text{tier 1 individual threshold} \\
& \land \neg \text{special offer price} < \text{individual scheduled price} \\
\lor & \text{(business account)} \\
& \land \neg \text{current purchase} > \text{tier 1 business threshold} \\
& \land \neg \text{current purchase} > \text{tier 1 business yearly threshold} \\
& \neg \text{special offer price} < \text{business scheduled price}
\end{align*}
\]

When functional specifications can be given as boolean expressions, we can apply any of the condition testing approaches described in Chapter 12, Section 12.4. A good test suite should at least exercise all elementary conditions occurring in the expression, and for simple conditions we might derive test case specifications for all possible combinations of truth values of the elementary conditions. For complex formulas, when testing all \(2^n\) combinations of \(n\) elementary conditions is apt to be too expensive, the modified decision/condition coverage criterion (page 12.4) derives a small set of test conditions such that each elementary condition independently affects the outcome.
**Pricing:** The *pricing* function determines the adjusted price of a configuration for a particular customer. The scheduled price of a configuration is the sum of the scheduled price of the model and the scheduled price of each component in the configuration. The adjusted price is either the scheduled price, if no discounts are applicable, or the scheduled price less any applicable discounts.

There are three price schedules and three corresponding discount schedules, *Business*, *Educational*, and *Individual*. The Business price and discount schedules apply only if the order is to be charged to a business account in good standing. The Educational price and discount schedules apply to educational institutions. The Individual price and discount schedules apply to all other customers. Account classes and rules for establishing business and educational accounts are described further in [...]..

A discount schedule includes up to three discount levels, in addition to the possibility of “no discount.” Each discount level is characterized by two threshold values, a value for the current purchase (configuration schedule price) and a cumulative value for purchases over the preceding 12 months (sum of adjusted price).

**Educational prices** The adjusted price for a purchase charged to an educational account in good standing is the scheduled price from the educational price schedule. No further discounts apply.

**Business account discounts** Business discounts depend on the size of the current purchase as well as business in the preceding 12 months. A tier 1 discount is applicable if the scheduled price of the current order exceeds the tier 1 current order threshold, or if total paid invoices to the account over the preceding 12 months exceeds the tier 1 year cumulative value threshold. A tier 2 discount is applicable if the current order exceeds the tier 2 current order threshold, or if total paid invoices to the account over the preceding 12 months exceeds the tier 2 cumulative value threshold. A tier 2 discount is also applicable if both the current order and 12 month cumulative payments exceed the tier 1 thresholds.

**Individual discounts** Purchase by individuals and by others without an established account in good standing are based on current value alone (not on cumulative purchases). A tier 1 individual discount is applicable if the scheduled price of the configuration in the current order exceeds the the tier 1 current order threshold. A tier 2 individual discount is applicable if the scheduled price of the configuration exceeds the tier 2 current order threshold.

**Special-price non-discountable offers** Sometimes a complete configuration is offered at a special, non-discountable price. When a special, non-discountable price is available for a configuration, the adjusted price is the non-discountable price or the regular price after any applicable discounts, whichever is less.

Figure 14.3: The functional specification of feature *pricing* of the Chipmunk web site.
We can produce different models of the decision structure of a specification depending on the original specification and on the technique we want to use for deriving test cases. If the original specification is expressed informally as in Figure 14.3, we can transform it into either a boolean expression, a graph, or a tabular model before applying a test case generation technique.

Techniques for deriving test case specifications from decision structures were originally developed for graph models, and in particular cause-effect graphs, which have been used since the early seventies. Cause-effect graphs are tedious to derive and do not scale well to complex specifications. Tables, on the other hand, are easy to work with and scale well.

The rows of a decision table represent basic conditions, and columns represent combinations of basic conditions. The last row of the table indicates the expected outputs for each combination. Cells of the table are labeled either True, False, or don’t care (usually written –), to indicate the truth value of the basic condition. Thus, each column is equivalent to a logical expression joining the required values (negated, in the case of False entries) and omitting the basic conditions with don’t care values.\(^2\)

Decision tables can be augmented with a set of constraints that limit the possible combinations of basic conditions. A constraint language can be based on boolean logic. Often it is useful to add some shorthand notations for common combinations such as at-most-one(C1, …, Cn) and exactly-one(C1, …, Cn), which are tedious to express with the standard boolean connectives.

Figure 14.4 shows the decision table for the functional specification of feature pricing of the Chipmunk web site presented in Figure 14.3.

The informal specification of Figure 14.3 identifies three customer profiles: educational, business, and individual. Table 14.4 has only rows Educational account (EduAc) and Business account (BusAc). The choice individual corresponds to the combination False, False for choices EduAc and BusAc, and is thus redundant. The informal specification of Figure 14.3 indicates different discount policies depending on the relation between the current purchase and two progressive thresholds for the current purchase and the yearly cumulative purchase. These cases correspond to rows 3 through 6 of Table 14.4. Conditions on thresholds that do not correspond to individual rows in the table can be defined by suitable combinations of values for these rows. Finally, the informal specification of Figure 14.3 distinguishes the cases in which special offer prices do not exceed either the scheduled or the tier 1 or tier 2 prices. Rows 7 through 9 of the table, suitably combined, capture all possible cases of special prices without redundancy.

Constraints formalize the compatibility relations among different basic conditions listed in the table. For example, a cumulative purchase exceeding threshold tier 2 also exceeds threshold tier 1.

The basic condition adequacy criterion requires generation of a test case specification for each column in the table. Don’t care entries of the table can be filled out arbitrarily, so long as constraints are not violated.

\(^2\)The set of columns sharing a label is therefore equivalent to a logical expression in sum-of-products form.
### Constraints

- at-most-one(EduAc, BusAc) at-most-one(YP < YT1, YP > YT2)
- YP > YT2 ⇒ YP > YT1 at-most-one(CP < CT1, CP > CT2)
- CP > CT2 ⇒ CP > CT1 at-most-one(SP < T1, SP > T2)
- SP > T2 ⇒ SP > T1

### Abbreviations

<table>
<thead>
<tr>
<th>EduAc</th>
<th>Educational account</th>
<th>BusAc</th>
<th>Business account</th>
<th>ND</th>
<th>No discount</th>
<th>T1</th>
<th>Tier 1</th>
<th>T2</th>
<th>Tier 2</th>
<th>SP</th>
<th>Special Price</th>
<th>T1</th>
<th>Special Price better than tier 1</th>
<th>T2</th>
<th>Special Price better than tier 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CP &gt; CT1</td>
<td>Current purchase greater than threshold 1</td>
<td>YP &gt; YT1</td>
<td>Year cumulative purchase greater than threshold 1</td>
<td>CP &gt; CT2</td>
<td>Current purchase greater than threshold 2</td>
<td>YP &gt; YT2</td>
<td>Year cumulative purchase greater than threshold 2</td>
<td>SP &gt; Sc</td>
<td>Special Price better than scheduled price</td>
<td>SP &gt; T1</td>
<td>Special Price better than tier 1</td>
<td>SP &gt; T2</td>
<td>Special Price better than tier 2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 14.4: The decision table for the functional specification of feature **pricing** of the Chipmunk web site of Figure 14.3.

Draft version produced February 22, 2006
The compound condition adequacy criterion requires a test case specification for each combination of truth values of basic conditions. The compound condition adequacy criterion generates a number of cases exponential in the number of basic conditions ($2^n$ combinations for $n$ conditions), and can thus be applied only to small sets of basic conditions.

For the modified condition/decision adequacy criterion (MC/DC), each column in the table represents a test case specification. In addition, for each of the original columns, MC/DC generates new columns by modifying each of the cells containing True or False. If modifying a truth value in one column results in a test case specification consistent with an existing column (agreeing in all places where neither is don’t care), the two test cases are represented by one merged column, provided they can be merged without violating constraints.

The MC/DC criterion formalizes the intuitive idea that a thorough test suite would not only test positive combinations of values, i.e., combinations that lead to specified outputs, but also negative combinations of values, i.e., combinations that differ from the specified ones and thus should produce different outputs, in some cases among the specified ones, in some other cases leading to error conditions.

Applying MC/DC to column 1 of Table 14.4 generates two additional columns: one for Educational Account = False and Special Price better than scheduled price = False, and the other for Educational Account = True and Special Price better than scheduled price = True. Both columns are already in the table (columns 3 and 2, respectively) and thus need not be added.

Similarly, from column 2, we generate two additional columns corresponding to Educational Account = False and Special Price better than scheduled price = True, and Educational Account = True and Special Price better than scheduled price = False, also already in the table.

Generation of a new column for each possible variation of the boolean values in the columns, varying exactly one value for each new column, produces 78 new columns, 21 of which can be merged with columns already in the table. Figure 14.5 shows a table obtained by suitably joining the generated columns with the existing ones. Many don’t care cells from the original table are assigned either True or False values, to allow merging of different columns or to obey constraints. The few don’t-care entries left can be set randomly to obtain a complete test case.

There are many ways of merging columns that generate different tables. The table in Figure 14.5 may not be the optimal one, i.e., the one with the fewest columns. The objective in test design is not to find an optimal test suite, but rather to produce a cost effective test suite with an acceptable trade-off between the cost of generating and executing test cases and the effectiveness of the tests.

The table in Figure 14.5 fixes the entries as required by the constraints, while the initial table in Figure 14.4 does not. Keeping constraints separate from the table corresponding to the initial specification increases the number of don’t care entries in the original table, which in turn increases the opportunity for merging columns when generating new cases with the MC/DC criterion. For example, if business account (BusAc) = False, the constraint at-most-one(EduAc, BusAc) can be satisfied by assigning either True or False to entry educational account. Fixing either choice prematurely may later make merging with a newly generated column impossible.
### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EduAc</td>
<td>Educational account</td>
</tr>
<tr>
<td>BusAc</td>
<td>Business account</td>
</tr>
<tr>
<td>CP &gt; CT1</td>
<td>Current purchase greater than threshold 1</td>
</tr>
<tr>
<td>YP &gt; YT1</td>
<td>Year cumulative purchase greater than threshold 1</td>
</tr>
<tr>
<td>CP &gt; CT2</td>
<td>Current purchase greater than threshold 2</td>
</tr>
<tr>
<td>YP &gt; YT2</td>
<td>Year cumulative purchase greater than threshold 2</td>
</tr>
<tr>
<td>SP &gt; Sc</td>
<td>Special Price better than scheduled price</td>
</tr>
<tr>
<td>SP &gt; T1</td>
<td>Special Price better than tier 1</td>
</tr>
<tr>
<td>SP &gt; T2</td>
<td>Special Price better than tier 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edu</td>
<td>Educational price</td>
</tr>
<tr>
<td>ND</td>
<td>No discount</td>
</tr>
<tr>
<td>T1</td>
<td>Tier 1</td>
</tr>
<tr>
<td>T2</td>
<td>Tier 2</td>
</tr>
</tbody>
</table>

Figure 14.5: The set of test cases generated for feature *pricing* of the Chipmunk web site applying the modified adequacy criterion.

Draft version produced February 22, 2006
14.4 Deriving Test Cases from Control and Data Flow Graphs

Functional specifications are seldom given as control or data flow graphs, but sometimes they describe a set of mutually dependent steps to be executed in a given (partial) order, and can thus be modeled with flow graphs.

The specification in Figure 14.6 describes the Chipmunk functionality that prepares orders for shipping. The specification indicates a set of steps to check the validity of fields in the order form. Type and validity of some of the values depend on other fields in the form. For example, shipping methods are different for domestic and international customers, and payment methods depend on customer type.

The informal specification in Figure 14.6 can be modeled with a control flow graph, where the nodes represent computations and branches represent control flow consistent with the dependencies among computations, as illustrated in Figure 14.7. Given a control or a data flow graph model, we can generate test case specifications using the criteria originally devised for structural testing and described in Chapters 12 and 13.

Control flow testing criteria require test cases that exercise all elements of a particular kind in a graph model. The node adequacy criterion requires each node to be exercised at least once, and corresponds to statement testing. It is easy to verify that test suite $T_{node}$ in Figure 14.8, consisting of test case specifications TC-1 and TC-2, causes all nodes of the control flow graph of Figure 14.7 to be traversed, and thus $T_{node}$ satisfies the node adequacy criterion.

The branch adequacy criterion requires each branch to be exercised at least once, i.e., each edge of the graph must be traversed by at least one test case. Test suite $T_{branch}$ (Figure ??) covers all branches of the control flow graph of Figure 14.7 and thus satisfies the branch adequacy criterion.

In principle, other test adequacy criteria described in Chapters 12 and 13 can be applied to more complex control structures derived from specifications, such as loops. A good functional specification should rarely result in a complex control structure, but data flow testing may be useful at a much coarser structure, e.g., to test interaction of transactions through a database.

14.5 Deriving Test Cases from Grammars

Functional specifications for complex documents or domain-specific notations, as well as for conventional compilers and interpreters, are often structured as an annotated grammar or set of regular expressions. Test suites can be systematically derived from these grammatical structures.

The informal specification of the Chipmunk web site advanced search, shown in Figure 14.10, defines the syntax of a search pattern. Not surprisingly, this specification can easily be expressed as a grammar. Figure 14.11 expresses the specification as a grammar in Bachus Naur Form (BNF).

A second example is given in Figure 14.12, which specifies a product configuration of the Chipmunk website. In this case, the syntactic structure of product configuration...
**Process shipping order:** The *Process shipping order* function checks the validity of orders and prepares the receipt.

A valid order contains the following data:

- **cost of goods** If the cost of goods is less than the minimum processable order (*MinOrder*) then the order is invalid.
- **shipping address** The address includes name, address, city, postal code, and country.
- **preferred shipping method** If the address is domestic, the shipping method must be either *land freight*, or *expedited land freight*, or *overnight air*. If the address is international, the shipping method must be either *air freight* or *expedited air freight*; a shipping cost is computed based on address and shipping method.
- **type of customer** A customer can be *individual*, *business* or *educational*
- **preferred method of payment** Individual customers can use only credit cards, while business and educational customers can choose between *credit card* and *invoice*.

- **card information** if the method of payment is credit card, fields credit card number, name on card, expiration date, and billing address, if different from shipping address, must be provided. If credit card information is not valid the user can either provide new data or abort the order.

The outputs of *Process shipping order* are

- **validity** Validity is a boolean output which indicates whether the order can be processed.
- **total charge** The total charge is the sum of the value of goods and the computed shipping costs (only if validity = true).
- **payment status** if all data are processed correctly and the credit card information is valid or the payment method is by invoice, payment status is set to valid, the order is entered and a receipt is prepared; otherwise validity = false.

Figure 14.6: Functional specification of feature *process shipping order* of the Chipmunk web site.
Figure 14.7: A control flow graph model corresponding to functionality Process shipping order of Figure 14.6
### T-node

<table>
<thead>
<tr>
<th>Case</th>
<th>Too small</th>
<th>Ship where</th>
<th>Ship method</th>
<th>Cust type</th>
<th>Pay method</th>
<th>Same addr</th>
<th>CC valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC-1</td>
<td>No</td>
<td>Int</td>
<td>Air</td>
<td>Bus</td>
<td>CC</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>TC-2</td>
<td>No</td>
<td>Dom</td>
<td>Air</td>
<td>Ind</td>
<td>CC</td>
<td>–</td>
<td>No (abort)</td>
</tr>
</tbody>
</table>

#### Abbreviations:
- Too small: CostOfGoods < MinOrder?
- Ship where: Shipping address, Int = international, Dom = domestic
- Ship how: Air = air freight, Land = land freight
- Cust type: Bus = business, Edu = educational, Ind = individual
- Pay method: CC = credit card, Inv = invoice
- Same addr: Billing address = shipping address?
- CC Valid: Credit card information passes validity check?

Figure 14.8: Test suite T-node, comprising test case specifications TC-1 and TC-2, exercises each of the nodes in a control flow graph model of the specification in 14.6

### T-branch

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

#### Abbreviations:
- Too small: CostOfGoods < MinOrder?
- Ship where: Shipping address, Int = international, Dom = domestic
- Ship how: Air = air freight, Land = land freight
- Cust type: Bus = business, Edu = educational, Ind = individual
- Pay method: CC = credit card, Inv = invoice
- Same addr: Billing address = shipping address?
- CC Valid: Credit card information passes validity check?

Figure 14.9: Test suite T-branch exercises each of the decision outcomes in a control flow flow graph model of the specification in 14.6

Draft version produced February 22, 2006
Advanced search: The Advanced search function allows for searching elements in the website database.

The key for searching can be:

a simple string, i.e., a simple sequence of characters,

a compound string, i.e.,

- a string terminated with character *, used as wild character, or
- a string composed of substrings included in braces and separated with commas, used to indicate alternatives.

a combination of strings, i.e., a set of strings combined with the boolean operators NOT, AND, OR, and grouped within parenthesis to change the priority of operators.

Examples:

laptop The routine searches for string “laptop”

{DVD*,CD*} The routine searches for strings that start with substring “DVD” or “CD” followed by any number of characters

NOT (C2021*) AND C20* The routine searches for strings that start with substring “C20” followed by any number of characters, except substring “21”

Figure 14.10: Functional specification of feature advanced search of the Chipmunk web site.

⟨search⟩ ::= ⟨search⟩ ⟨binop⟩ ⟨term⟩ | not ⟨search⟩ | ⟨term⟩
⟨binop⟩ ::= and | or
⟨term⟩ ::= ⟨regexp⟩ | { ⟨search⟩ }
⟨regexp⟩ ::= Char ⟨regexp⟩ | Char | { ⟨choices⟩ } | *
⟨choices⟩ ::= ⟨regexp⟩ | ⟨regexp⟩ + ⟨choices⟩

Figure 14.11: BNF description of functionality Advanced search.
is described by an XML schema, which defines an element Model of type ProductConfigurationType. XML schemata are essentially a variant of BNF, so it is not difficult to render the schema in the same BNF notation, as shown in Figure 14.12.

Grammars are well suited to represent inputs of varying and unbounded size, with recursive structures and boundary conditions. These characteristics are not easily addressed with the fixed lists of parameters required by conventional combinatoric techniques described in Chapter 11, nor by other model-based techniques presented in this chapter.

Generating test cases from grammar specifications is straightforward and can easily be automated. Each test case is a string generated from the grammar. To produce a string, we start from a non-terminal symbol and progressively apply productions to substitute substrings for non-terminals occurring in the current string, until we obtain a string composed only of terminal symbols.

In general, we must choose among several applicable production rules at each step. A simple criterion require each production to be exercised at least once in producing a set of test cases.

The number and complexity of the generated test cases depend on the order of application of the productions. If we first apply productions with non-terminals on the right hand side, we generate a smaller set of large test cases. First applying productions with only terminals on the right hand side generates larger sets of smaller test cases. An algorithm that favors non-terminals applied to the BNF for advanced search of Figure 14.10, exercises all the productions to generate the single test case

\[ \text{not Char \{*, Char\} and (Char or Char)} \]

The derivation tree for this test case is given in Figure 14.14. It shows that each productions of the BNF is exercised at least once.

The simple production coverage criterion is subsumed by a richer criterion that applies boundary conditions on the number of times each recursive production is applied successively. To generate test cases for boundary conditions we need to choose a minimum and maximum number of applications of each recursive production and then generate a test case for the minimum, maximum, one greater than minimum and one smaller than maximum. The approach is essentially similar to boundary interior path testing of program loops (see Section 12.5 of Chapter 12, page 269), where the “loop” in this case is in repeated applications of a production.

To apply the boundary condition criterion, we need to annotate recursive productions with limits. Names and limits are shown in Figure 14.15, which extends the grammar of Figure 14.13. Alternatives within compound productions are broken out into individual productions. Production names are added for reference, and limits are added to recursive productions. In the example of Figure 14.15, the limit of productions compSeq1 and optCompSeq1 is set to 16, i.e., we assume that each model can have at most 16 required and 16 optional components.

The boundary condition grammar based criterion would extend the minimal set by adding test cases that cover the following choices:

- zero required components (compSeq1 applied 0 times)
- one required component (compSeq1 applied 1 time)
Figure 14.12: An XML Schema description of a Product configuration on the Chipmunk website. Items are enclosed in matching tags (⟨tag⟩ text ⟨/tag⟩) or incorporated in a self-terminating tag (⟨tag attribute="value" /⟩). The schema describes type ProductConfigurationType as a tuple composed of a required field modelNumber of type string; a set (possibly empty) of Components, each of which is composed of two string-valued fields ComponentType and ComponentValue; and a possibly empty set of OptionalComponents, each of which is composed of a single string-valued ComponentType.
(Model) ::= ⟨modelNumber⟩ ⟨compSequence⟩ ⟨optCompSequence⟩

(compSequence) ::= ⟨Component⟩ ⟨compSequence⟩ | empty

(optCompSequence) ::= ⟨OptionalComponent⟩ ⟨optCompSequence⟩ | empty

(Component) ::= ⟨ComponentType⟩ ⟨ComponentValue⟩

(OptionalComponent) ::= ⟨ComponentType⟩

(modelNumber) ::= string

(ComponentType) ::= string

(ComponentValue) ::= string

Figure 14.13: BNF description of Product Configuration.

Figure 14.14: The derivation tree of a test case for functionality Advanced Search derived from the BNF specification of Figure 14.11.
Figure 14.15: The BNF description of Product Configuration extended with production names and limits.

- fifteen required components (compSeq1 applied \(n - 1\) times)
- sixteen required components (compSeq1 applied \(n\) times)
- zero optional components (optCompSeq1 applied 0 times)
- one optional component (optCompSeq1 applied 1 time)
- fifteen optional components (optCompSeq1 applied \(n - 1\) times)
- sixteen optional components (optCompSeq1 applied \(n\) times)

Probabilistic grammar based criteria assign probabilities to productions, indicating which production to select at each step to generate test cases. Unlike names and limits, probabilities are attached to grammar productions as a separate set of annotations. We can generate several sets of test cases from the same grammar with different sets of probabilities, called “seeds.” Figure 14.16 shows a sample seed for the grammar that specifies the product configuration functionality of the Chipmunk web site presented in Figure 14.15.

Probabilities are interpreted as weights that determine how frequently each production is used to generate a test case. The equal weight for compSeq1 and optCompSeq1 in Figure 14.15 indicates that test cases are generated by balancing use of these two productions, i.e., they contain approximately the same number of required and optional components. Weight 0 disables the productions, which are then applied only when application of competing productions reaches the limit indicated in the grammar.
Open Research Issues

As long as there have been models of software, there has been model-based testing. A recent and ongoing ferment of activity in model-based testing is partly the result of wider use of models throughout software development. Ongoing research will certainly include test design based on software architecture, domain-specific models, and models of emerging classes of systems such as service-oriented architectures and adaptive systems, as well as additional classes of systems and models that we cannot yet anticipate.

As well as following the general trend toward greater use of models in development, though, research in model-based testing reflects greater understanding of the special role that models of software can play in test design and in combining conventional testing with analysis. A model is often the best way — perhaps the only way — to divide one property to be verified into two, one part that is best verified with static analysis and another part that is best verified with testing. Conformance testing of all kinds exploits models in this way, focusing analysis techniques where they are most necessary (e.g., non-deterministic scheduling decisions in concurrent software) and using testing to cost-effectively verify consistency between model and program.

Models are also used to specify and describe system structure at levels of organization beyond those that are directly accommodated in conventional programming languages, e.g., components and subsystems. Analysis, and to a lesser extent testing, has been an explicit concern in development of architecture description languages. Still there remains a divide between models developed primarily for people to communicate and record design decisions (e.g., UML) and models developed primarily for verification (e.g., various FSM notations). Today we see a good deal of research re-purposing design models for test design, which involves adding or disambiguating semantics of notations intended for human communication. A challenge for future design notations is to provide a better foundation for analysis and testing without sacrificing the characteristics that make them useful for communicating and recording design decisions.

An important issue in modeling, and by extension in model-based testing, is how to use multiple model “views” to together form a comprehensive model of a program.
More work is needed on test design that uses more than one modeling view, or on the potential interplay between test specifications derived from different model views of the same program.

As with many other areas of software testing and analysis, more empirical research is also needed to characterize cost and effectiveness of model-based testing approaches. Perhaps even more than in other areas of testing research, this is not only a matter of carrying out experiments and case studies, but is at least as much a matter of understanding how to pose questions that can be effectively answered by experiments and whose answers generalize in useful ways.

**Further Reading**

Myers’ classic text [Mye79] describes a number of techniques for testing decision structures. Richardson, O’Malley, and Tittle [ROT89] and Stocks and Carrington [SC96] are among attempts to generate test cases based on the structure of (formal) specifications. Beizer’s *Black Box Testing* [Bei95] is a popular presentation of techniques for testing based on control and data flow structure of (informal) specifications.

Test design based on finite state machines has long been important in the domain of communication protocol development and conformance testing; Fujiwara, von Bochmann, Amalou, and Ghedamsi [FvBK+91] is a good introduction. Gargantini and Heitmeyer [GH99] describe a related approach applicable to software systems in which the finite-state machine is not explicit but can be derived from a requirements specification.

Test generation from context-free grammars is described by Celentano et al [CCD+80] and apparently goes back at least to Hanford’s test generator for an IBM PL/I compiler [Han70]. The probabilistic approach to grammar-based testing is described by Sirer and Bershad [SB99], who use annotated grammars to systematically generate tests for Java virtual machine implementations.

Heimdahl et al [HDW04] provide a cautionary note regarding how naive model-based testing can go wrong, while a case study by Pretschner et al [PPW+05] suggests that model-based testing is particularly effective in revealing errors in informal specifications.

**Related Topics**

Readers interested in finite state machine based testing may proceed to Chapter 15, in which finite state models are applied to testing object-oriented programs.
Exercises

Ex14.1. Consider the following columns that correspond to educational and individual accounts of feature pricing of Figure ??:

<table>
<thead>
<tr>
<th></th>
<th>Education</th>
<th>Individual</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Edu.</em></td>
<td><em>T</em></td>
<td><em>F</em></td>
</tr>
<tr>
<td><em>CP &gt; CT1</em></td>
<td><em>-</em></td>
<td><em>F</em></td>
</tr>
<tr>
<td><em>CP &gt; CT2</em></td>
<td><em>-</em></td>
<td><em>F</em></td>
</tr>
<tr>
<td><em>SP &gt; Sc</em></td>
<td><em>F</em></td>
<td><em>F</em></td>
</tr>
<tr>
<td><em>SP &gt; T1</em></td>
<td><em>-</em></td>
<td><em>F</em></td>
</tr>
<tr>
<td><em>SP &gt; T2</em></td>
<td><em>-</em></td>
<td><em>F</em></td>
</tr>
<tr>
<td><em>Out</em></td>
<td><em>Edu</em></td>
<td><em>SP</em></td>
</tr>
</tbody>
</table>

write a set of boolean expressions for the outputs and apply the modified condition/decision adequacy criterion (MC/DC) presented in Chapter 12 to derive a set of test cases for the derived boolean expressions. Compare the result with the test case specifications given in Figure ??.

Make an exercise: If a “don’t care” transition can be triggered, should we treat it as a “self” or as an “error” transition?

Ex14.2. Derive sets of test cases for functionality Maintenance applying Transition Coverage, Single State Path Coverage, Single Transition Path Coverage, and Boundary Interior Loop Coverage to the FSM specification of Figure ??

Ex14.3. Derive test cases for functionality Maintenance applying Transition Coverage to the FSM specification of Figure ??, assuming that implicit transitions are (1) error conditions or (2) self transitions.

Ex14.4. We have stated that the transitions in a state-machine specification can be considered as precondition, postcondition pairs. Often the finite-state machine is an abstraction of a more complex system which is not truly finite-state. Additional “state” information is associated with each of the states, including fields and variables that may be changed by an action attached to a state transition, and a predicate that should always be true in that state. The same system can often be described by a machine with a few states and complicated predicates, or a machine with more states and simpler predicates. Given this observation, how would you combine test selection methods for finite-state machine specifications with decision structure testing methods? Can you devise a method that selects the same test cases regardless of the specification style (more or fewer states)? Is it wise to do so?