frame. It is important to note that the test frame contains information about the environment objects. Hence, it is useful in setting up the environment prior to a test.

Test frames are generated by generating all possible combinations of choices while satisfying the constraints. Choices that are marked error or single are not combined with others and produce only one test case. It is easy to observe that without any constraints (selector expressions), a total of 160 test frames will be generated from the specification in Example 2.17.

Step 7: Evaluate Generator Output

In this step the tester examines the test frames for any redundancy or missing cases. This might lead to a modification in the test specification (Step 5) and a return to Step 6.

Step 8: Generate Test Scripts

Test cases generated from test frames are combined into test scripts. A test script is a grouping of test cases. Generally, test cases that do not require any changes in settings of the environment objects are grouped together. This enables a test driver to efficiently execute the tests.

That completes our description of the category-partition method. As you may have noticed, the method is essentially a systematization of the equivalence partitioning and boundary-value techniques discussed earlier.

Writing a test specification allows a test team to take a close look at the program specifications. For large systems, this task can be divided among members of the test team. While a significant part of the method requires manual work, availability of a tool that processes TSL specifications helps in test generation and documentation. It also reduces the chances of errors in test cases.

2.6 Cause–Effect Graphing

We described two techniques for test selection based on equivalence partitioning and boundary-value analysis. One is based on unidimensional partitioning and the other on multidimensional partitioning of the input domain. While equivalence classes created using multidimensional partitioning allow the selection of a large number of input combinations, the number of such combinations could be astronomical. Further, many of these combinations, as in Example 2.8, are infeasible and make the test-selection process tedious.

Cause–effect graphing, also known as dependency modeling, focuses on modeling dependency relationships among program input conditions, known as *causes*, and output conditions, known as *effects*. The relationship is expressed visually in terms of a cause–effect graph. The graph is a visual representation of a logical relationship among inputs and outputs that can be expressed as a Boolean expression. The graph allows selection of various combinations of input values as tests. The combinatorial explosion in the number of tests is avoided by using certain heuristics during test generation.

A cause is any condition in the requirements that may effect the program output. An effect is the response of the program to some combination of input conditions. It may be, for example, an error message displayed on the screen, a new window displayed, or a database updated. An effect need not be an *output* visible to the user of the program. Instead, it could also be an internal *test point* in the program that can be probed during testing to check if some intermediate result is as expected. For example, the intermediate test point could be at the entrance to a method to indicate that indeed the method has been invoked.

A requirement such as "Dispense food only when the DF switch is ON" contains a cause "DF switch is ON" and an effect "Dispense food". This requirement implies a relationship between the "DF switch is ON" and the effect "Dispense food". Of course, other requirements might require additional causes for the occurrence of the "Dispense food" effect. The following generic procedure is used for the generation of tests using cause–effect graphing:

1. Identify causes and effects by reading the requirements. Each cause and effect is assigned a unique identifier. Note that an effect can also be a cause for some other effect.
2. Express the relationship between causes and effects using a cause–effect graph.
3. Transform the cause–effect graph into a limited entry decision table, hereafter referred to simply as decision table.
4. Generate tests from the decision table.

The basic notation used in cause–effect graphing and a few illustrative examples are given below.

2.6.1 Notation Used in Cause–Effect Graphing

The basic elements of a cause–effect graph are shown in Figure 2.10. A typical cause–effect graph is formed by combining the basic elements so as to capture the relations between causes and effects derived from

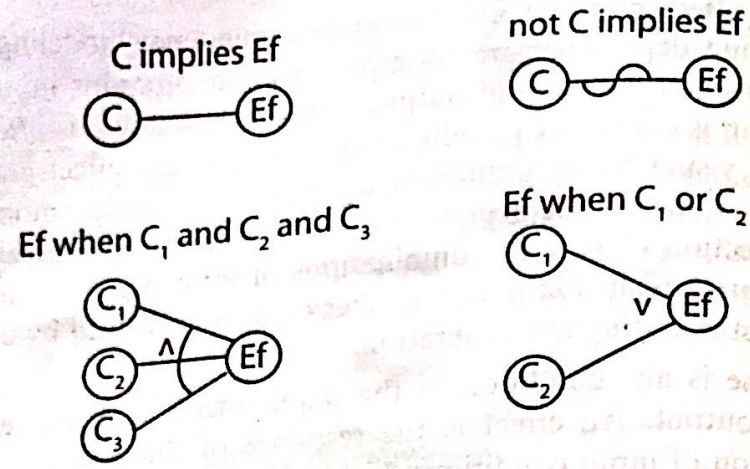

Fig. 2.10 Basic elements of a cause–effect graph: *implication, not* (~), *and* (∧), *or* (∨). C, C_1, C_2, C_3 denote causes and *Ef* denotes effect. An arc is used, for example in the *and* relationship, to group three or more causes.

the requirements. The semantics of the four basic elements shown in Figure 2.10 is expressed below in terms of the `if-then` construct; C, C_1, C_2, and C_3 denote causes, and *Ef* denotes an effect.

C implies *Ef*:	`if(`C`) then` *Ef*;
not C implies *Ef*:	`if (`$\neg C$`) then` *Ef*;
Ef when C_1 and C_2 and C_3:	`if(`C_1 `&&` C_2 `&&` C_3`) then` *Ef*;
Ef when C_1 or C_2:	`if(`$C_1 \| \| C_2$`) then` *Ef*;

There often arise constraints among causes. For example, consider an inventory-control system that tracks the inventory of various items that are stocked. For each item, an inventory attribute is set to *Normal, Low, and Empty*. The inventory control software takes actions as the value of this attribute changes. When identifying causes, each of the three inventory levels will lead to a different cause listed below.

C_1 : "Inventory is normal"
C_2 : "Inventory is low"
C_3 :"Inventory is zero"

However, at any instant, exactly one of C_1, C_2, and C_3 can be true. This relationship among the three causes is expressed in a cause–effect graph using the *Exclusive (E)* constraint shown in Figure 2.11. In addition, shown in this figure are the *Inclusive (I)*, *Requires (R)*, and *One and only one (O)* constraints. The I constraint between two causes C_1 and C_2 implies that at least one of the two must be present. The R constraint between C_1 and C_2 implies the C_1 requires C_2. The O-constraint models the condition that one, and only one, of C_1 and C_2 must hold.

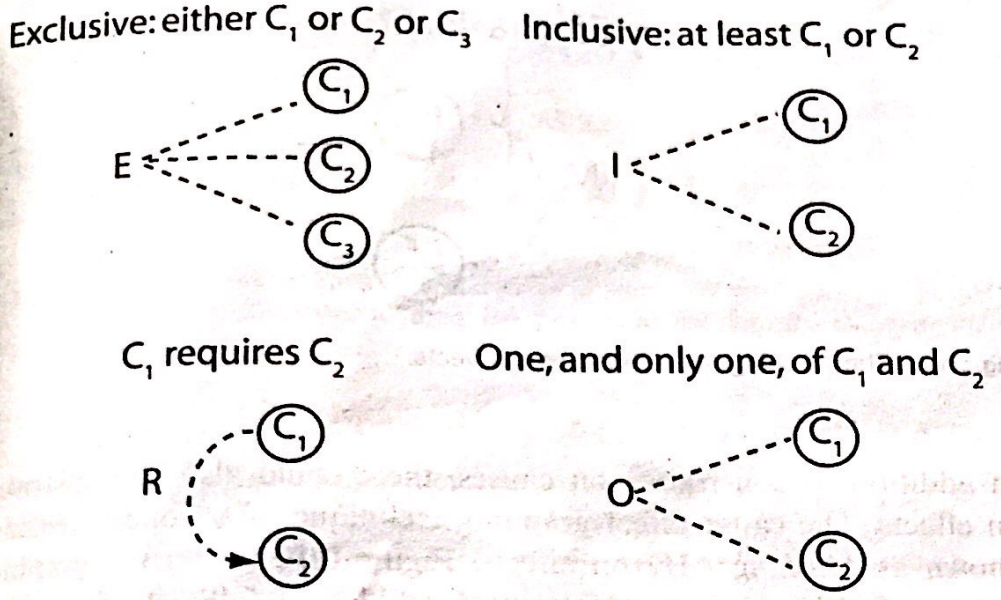

Fig. 2.11 Constraints among causes (*E*, *I*, *O*, and *R*) and among effects (*M*).

The table below lists all possible values of causes constrained by E, I, R, and O. A 0 or a 1 under a cause implies that the corresponding condition is, respectively, false and true. The arity of all constraints, except R, is greater than 1, that is all except the R constraint can be applied to two or more causes; the R constraint is applied to two causes.

Constraint	Arity	Possible values		
		C_1	C_2	C_3
$E(C_1, C_2, C_3)$	$n > 1$	0	0	0
		1	0	0
		0	1	0
		0	0	1
$I(C_1, C_2)$	$n > 1$	1	0	–
		0	1	–
		1	1	–
$R(C_1, C_2)$	$n > 1$	1	1	–
		1	1	–
		0	0	–
		0	1	–
$O(C_1, C_2, C_3)$	$n > 1$	1	0	0
		0	1	0
		0	0	1

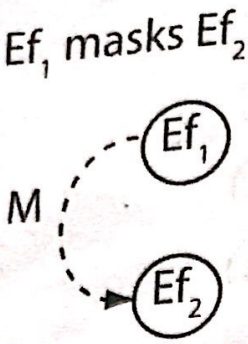

Fig. 2.12 The masking constraint among effects.

In addition to constraints on causes, there could also be constraints on effects. The cause–effect graphing technique offers one constraint, known as *Masking (M)* on effects. Figure 2.12 shows the graphical notation for the masking constraint. Consider the following two effects in the inventory example mentioned earlier.

Ef_1 : Generate "Shipping invoice".

Ef_2 : Generate an "Order not shipped" regret letter.

Effect Ef_1 occurs when an order can be met from the inventory. Effect Ef_2 occurs when the order placed cannot be met from the inventory or when the ordered item has been discontinued after the order was placed. However, Ef_2 is masked by Ef_1 for the same order, that is both effects cannot occur for the same order.

A condition that is false (true) is said to be in the "0-state" (1-state). Similarly, an effect can be *present* (1-state) or *absent* (0-state).

2.6.2 Creating Cause–Effect Graphs

The process of creating a cause–effect graph consists of two major steps. First, the causes and effects are identified by a careful examination of the requirements. This process also exposes the relationships among various causes and effects as well as constraints among the causes and effects. Each cause and effect is assigned a unique identifier for ease of reference in the cause–effect graph.

In the second step, the cause–effect graph is constructed to express the relationships extracted from the requirements. When the number of causes and effects is large, say over 100 causes and 45 effects, it is appropriate to use an incremental approach. An illustrative example follows:

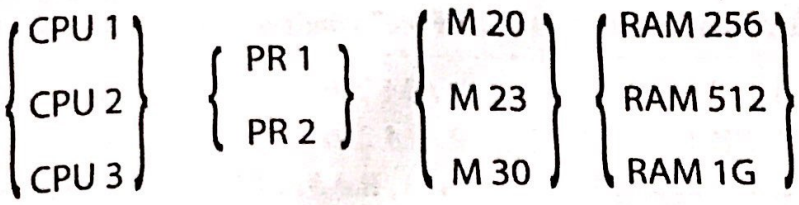

Fig. 2.13 Possible configurations of a computer system sold by a web-based company: CPU: CPU configuration; PR, printer; M, monitor and RAM, memory upgrade.

Example 2.19: Let us consider the task of test generation for a GUI-based computer purchase system. A Web-based company is selling computers (CPU), printers (PR), monitors (M), and additional memory (RAM). An order configuration consists of one to four items as shown in Figure 2.13. The GUI consists of four windows for displaying selections from CPU, Printer, Monitor, and RAM and one window where any free giveaway items are displayed.

For each order, the buyer may select from three CPU models, two printer models, and three monitors. There are separate windows one each for CPU, printer, and monitor that show the possible selections. For simplicity we assume that RAM is available only as an upgrade and that only one unit of each item can be purchased in one order.

Monitors M 20 and M 23 can be purchased with any CPU or as a standalone item. M 30 can only be purchased with CPU 3. PR 1 is available free with the purchase of CPU 2 or CPU 3. Monitors and printers, except for M 30, can also be purchased separately without purchasing any CPU. Purchase of CPU 1 gets RAM 256 upgrade and purchase of CPU 2 or CPU 3 gets a RAM 512 upgrade. The RAM 1G upgrade and a free PR 2 is available when CPU 3 is purchased with monitor M 30.

When a buyer selects a CPU, the contents of the printer and monitor windows are updated. Similarly, if a printer or a monitor is selected, contents of various windows are updated. Any free printer and RAM available with the CPU selection is displayed in a different window marked "Free". The total price, including taxes, for the items purchased is calculated and displayed in the "Price" window. Selection of a monitor could also change the items displayed in the "Free" window. Sample configurations and contents of the "Free" window are given below.

Items purchased	"Free" window	Price
CPU 1	RAM 256	$499
CPU 1. PR 1	RAM 256	$628
CPU 2. PR 2. M 23	PR 1, RAM 512	$2257
CPU 3. M 30	PR 2, RAM 1G	$3548

The first step in cause–effect graphing is to read the requirements carefully and make a list of causes and effects. For this example, we will consider only a subset of the effects and leave the remaining as an exercise. This strategy also illustrates how one could generate tests using an incremental strategy.

A careful reading of the requirements is used to extract the following causes. We have assigned a unique identifier, C 1 through C 8 to each cause. Each cause listed below is a condition that can be true or false. For example, C 8 is true if monitor M 30 is purchased.

C 1: Purchase CPU 1.
C 2: Purchase CPU 2.
C 3: Purchase CPU 3.
C 4: Purchase PR 1.
C 5: Purchase PR 2.
C 6: Purchase M 20.
C 7: Purchase M 23.
C 8: Purchase M 30.

Note that while it is possible to order any of the items listed above, the GUI will update the selection available depending on what CPU or any other item is selected. For example, if CPU 3 is selected for purchase, then monitors M 20 and M 23 will not be available in the monitor-selection window. Similarly, if monitor M 30 is selected for purchase, then CPU 1 and CPU 2 will not be available in the CPU window.

Next, we identify the effects. In this example, the application software calculates and displays the list of items available free with the purchase and the total price. Hence, the effect is in terms of the contents of the "Free" and "Price" windows. There are several other effects related to the GUI update actions. These effects are left for the exercise (see Exercise 2.21).

Calculation of the total purchase price depends on the items purchased and the unit price of each item. The unit price is obtained by the application from a price database. The price calculation and display is a cause that creates the effect of displaying the total price.

For simplicity, we ignore the price-related cause and effect. The set of effects in terms of the contents of the "Free" display window are listed below.

Ef_1 : RAM 256.
Ef_2 : RAM 512 and PR 1.
Ef_3 : RAM 1G and PR 2.
Ef_4 : No giveaway with this item.

Now that we have identified causes, effects, and their relationships, we begin to construct the cause–effect graph. Figure 2.14 shows the complete graph that expresses the relationships between C_1 through C_8 and effects Ef1 through Ef4.

From the cause–effect graph in Figure 2.14, we notice that C_1, C_2, and C_3 are constrained using the E (exclusive) relationship. This

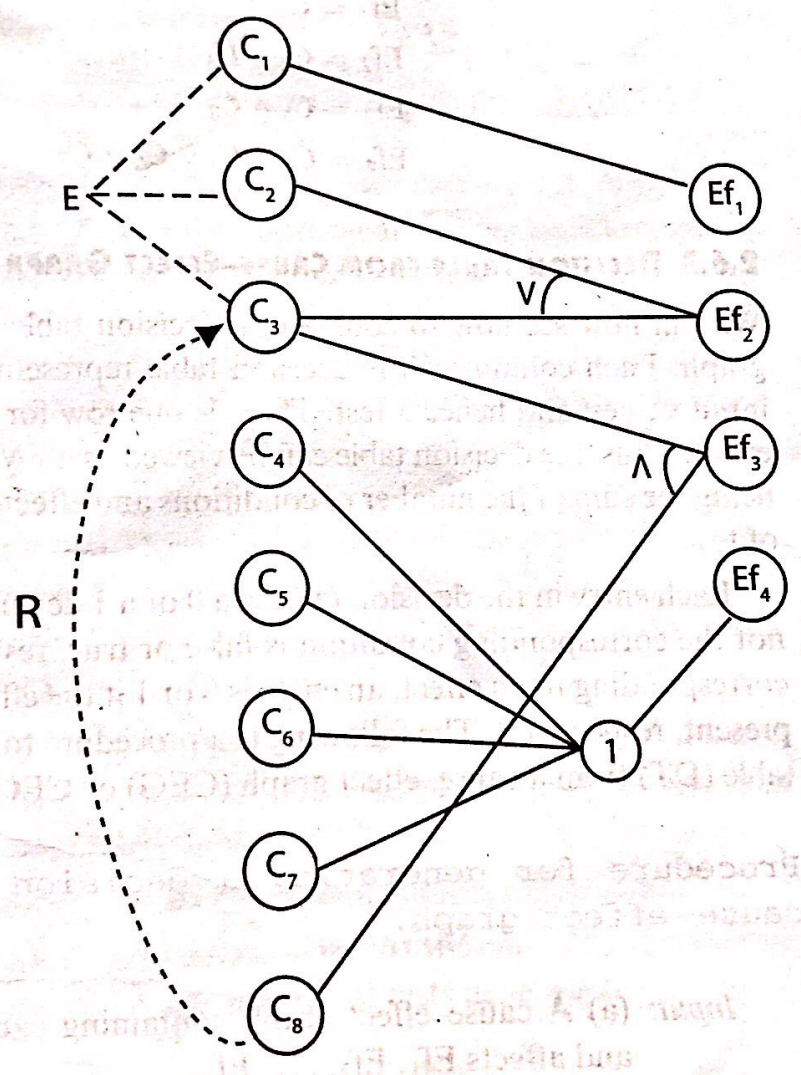

Fig. 2.14 Cause–effect graph for the web-based computer sale application. C_1, C_2, and C_3 denote the purchase of, respectively, CPU 1, CPU 2, and CPU 3. C_4 and C_5 denote the purchase of printers PR 1 and PR 2, respectively. C_6, C_7, and C_8 denote the purchase of monitors M 20, M 23, and M 30, respectively.

expresses the requirement that only one CPU can be purchased in one order. Similarly, C_3 and C_8 are related via the R (requires) constraint to express the requirement that monitor M 30 can only be purchased with CPU 3. Relationships among causes and effects are expressed using the basic elements shown earlier in Figure 2.10.

Note the use of an intermediate node labeled 1 in Figure 2.14. Though not necessary in this example, such intermediate nodes are often useful when an effect depends on conditions combined using more than one operator, for example $(C_1 \wedge C_2) \vee C_3$. Note also that purchase of printers and monitors without any CPU leads to no free item (Ef_4).

The relationships between effects and causes shown in Figure 2.14 can be expressed in terms of Boolean expressions as follows:

$$Ef_1 = C_1$$
$$Ef_2 = C_2 \vee C_3$$
$$Ef_3 = C_3 \wedge C_8$$
$$Ef_4 = C_4 \vee C_5 \vee C_6 \vee C_7$$

2.6.3 Decision Table from Cause–Effect Graph

We will now see how to construct a decision table from a cause–effect graph. Each column of the decision table represents a combination of input values, and hence a test. There is one row for each condition and effect. Thus, the decision table can be viewed as an $N \times M$ matrix with N being the sum of the number of conditions and effects and M the number of tests.

Each entry in the decision table is a 0 or a 1 depending on whether or not the corresponding condition is false or true, respectively. For a row corresponding to an effect, an entry is 0 or 1 if the effect is not present or present, respectively. The following is a procedure to generate a decision table (DT) from a cause–effect graph (CEG) or CEGDT.

Procedure for generating a decision table from a cause–effect graph.

Input: (a) A cause–effect graph containing causes $C_1, C_2, \ldots, C_p$ and affects $Ef_1, Ef_2, \ldots, Ef_q$.

Output: A decision table DT containing $N = p + q$ rows and M columns, where M depends on the relationship between the causes and effects as captured in the cause–effect graph.

Procedure: CEGDT

/* i is the index of the next effect to be considered.

next_dt_col is the next empty column in the decision table.

V_k: a vector of size $p + q$ containing 1s and 0s. V_j, $1 \leq j \leq p$, indicates the state of condition C_j and $V_l, p < l \leq p+q$, indicates the presence or absence of effect Ef_{l-p}.

*/

Step 1 *Initialize DT to an empty decision table.*

next_ dt_col = 1.

Step 2 Execute the following steps for $i = 1$ to q.

2.1 *Select the next effect to be processed.*

Let e = Ef_i.

2.2 *Find combinations of conditions that cause e to be present.*

Assume that e is present. Starting at e, trace the cause–effect graph backward and determine the combinations of conditions $C_1, C_2, \ldots, C_p$ that lead to e being present. Avoid combinatorial explosion by using the heuristics given in the text following this procedure. Make sure that the combinations satisfy any constraints among the causes.

Let $V_1, V_2, \ldots, V_{m_i}$ be the combinations of causes that lead to e being present. There must be at least one combination that makes e to be present, that is in 1-state, and hence $m_i \geq 1$. Set $V_k(l)$, $p < l \leq p + q$ to 0 or 1 depending on whether effect Ef_{l-p} is present for the combination of all conditions in V_k.

2.3 *Update the decision table.*

Add $V_1, V_2, \ldots, V_{m_i}$ to the decision table as successive columns starting at *next_dt_col*.

2.4 *Update the next available column in the decision table.*

next_dt_col = *next_dt_col* + m_i. At the end of this procedure, *next_dt_col* − 1 is the number of tests generated.

End of Procedure CEGDT

Procedure CEGDT can be automated for the generation of tests. However, as indicated in Step 2.2, one needs to use some heuristics in order to avoid combinatorial explosion in the number of tests generated.

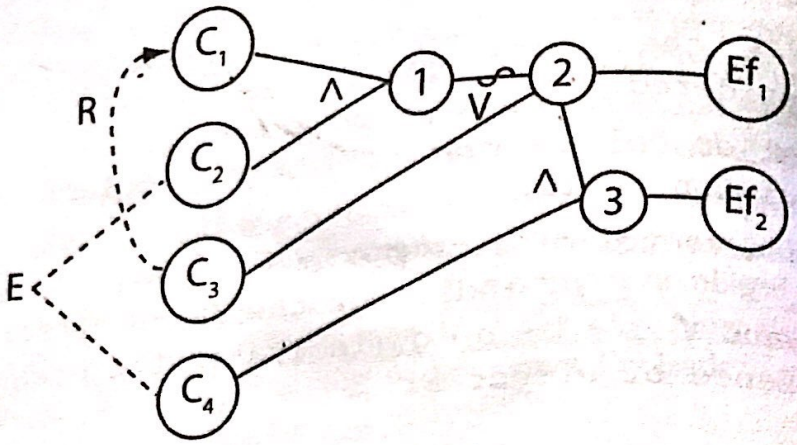

Fig. 2.15 A cause-effect graph to illustrate procedure CEGDT.

Before we introduce the heuristics, we illustrate through a simple example the application of Procedure CEGDT *without* applying the heuristics in Step 2.

Example 2.20: Consider the cause–effect graph in Figure 2.15. It shows four causes labeled C_1, C_2, C_3, and C_4 and two effects labeled Ef_1 and Ef_2. There are three intermediate nodes labeled 1, 2, and 3. Let us now follow procedure CEGDT step-by-step to generate a decision table.

In Step 1 we set *next_dt_col* = 1 to initialize the decision table to empty. Next, $i = 1$ and, in accordance with Step 2.1, $e = Ef_1$. Continuing further and in accordance with Step 2, we trace backward from e to determine combinations that will cause e to be present. e must be present when node 2 is in 1-state. Moving backward from node 2 in the cause–effect graph, we note that any of the following three combinations of states of nodes 1 and C_3 will lead to e being present: (0, 1), (1, 1), and (0, 0).

Node 1 is also an internal node and hence we move further back to obtain the values of C_1 and C_2 that effect node 1. Combination of C_1 and C_2 that brings node 1 to 1-state is (1, 1) and combinations that bring to 0-state are (1, 0), (0, 1), and (0, 0). Combining this information with that derived earlier for nodes 1 and C_3, we obtain the following seven combinations of C_1, C_2, and C_3 that cause e to be present.

Next, from Figure 2.15, we note that C_3 requires C_1, which implies that C_1 must be in 1-state for C_3 to be in 1-state. This constraint makes infeasible the second and third combinations above. In the end, we obtain the following five combinations of the four causes that lead to e being present.

1	0	1
1	1	1
1	0	0
0	1	0
0	0	0

Setting C_4 to 0 and appending the values of Ef_1 and Ef_2, we obtain the following five vectors. Note that $m_1 = 5$ in Step 2. This completes the application of Step 2.2, without the application of any heuristics, in the CEGDT procedure.

	C_1	C_2	C_3	C_4	e_1	e_2
V_1	1	0	1	0	1	0
V_2	1	1	1	0	1	0
V_3	1	0	0	0	1	0
V_4	0	1	0	0	1	0
V_5	0	0	0	0	1	0

The five vectors are transposed and added to the decision table starting at column *next_dt_ col*, which is 1. The decision table at the end of Step 2.3 is as follows:

	1	2	3	4	5
C_1	1	1	1	0	0
C_2	0	1	0	1	0
C_3	1	1	0	0	0
C_4	0	0	0	0	0
Ef_1	1	1	1	1	1
Ef_2	0	0	0	0	0

We update *next_dt_col* to 6, increment i to 2, and get back to Step 2.1. We now have $e = Ef_2$. Tracing backward we find that for e to be present, node 3 must be in the 1-state. This is possible with only one combination of node 2 and C_4, which is (1, 1).

Earlier we derived the combinations of C_1, C_2, and C_3 that lead node 2 into the 1-state. Combining these with the value of C_4, we arrive at the following combination of causes that lead to the presence of Ef_2.

1	0	1	1
1	1	1	1
1	0	0	1
0	1	0	1
0	0	0	1

From Figure 2.15, we note that C_2 and C_4 cannot be present simultaneously. Hence, we discard the second and the fourth combinations from the list above and obtain the following three feasible combinations.

1	0	1	1
1	0	0	1
0	0	0	1

Appending the corresponding values of Ef_1 and Ef_2 to each of the above combinations, we obtain the following three vectors.

V_1	1	0	1	1	1	1
V_2	1	0	0	1	1	1
V_3	0	0	0	1	1	1

Transposing the vectors listed above and appending them as three columns to the existing decision table, we obtain the following.

	1	2	3	4	5	6	7	8
C_1	1	1	1	0	0	1	1	0
C_2	0	1	0	1	0	0	0	0
C_3	1	1	0	0	0	1	0	0
C_4	0	0	0	0	0	1	1	1
Ef_1	1	1	1	1	1	1	1	1
Ef_2	0	0	0	0	0	1	1	1

Next we update *nxt_dt_col* to 9. Of course, doing so is useless as the loop set up in Step 2 is now terminated. The decision table listed

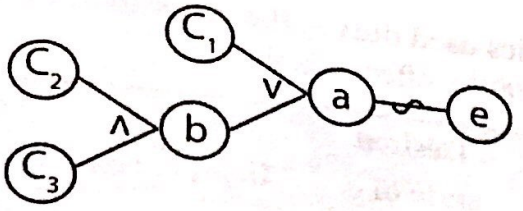

Fig. 2.16 Cause–effect graph for Example 2.21.

above is the output obtained by applying procedure CEGDT to the cause–effect graph in Figure 2.15.

2.6.4 Heuristics to Avoid Combinatorial Explosion

While tracing back through a cause–effect graph, we generate combinations of causes that set an intermediate node, or an effect, to 0-state or a 1-state. Doing so in a brute-force manner could lead to a large number of combinations. In the worst case, if n causes are related to an effect e, then the maximum number of combinations that bring e to a 1-state is 2^n.

As tests are derived from the combinations of causes, large values of n could lead to an exorbitantly large number of tests. We avoid such a combinatorial explosion by using simple heuristics related to the "AND" ($\wedge$) and "OR" ($\vee$) nodes.

Certainly, the heuristics described below are based on the assumption that certain types of errors are less likely to occur than others. Thus, while applying the heuristics to generate test inputs is likely to lead to a significant reduction in the number of tests generated, it may also discard tests that would have revealed a program error. Hence, one must apply the heuristics with care and only when the number of tests generated without their application is too large to be useful in practice.

A set of four heuristics labeled H_1 through H_4 is given in Table 2.4. The leftmost column shows the node type in the cause–effect graph, the center column is the desired state of the dependent node, and the rightmost column is the heuristic for generating combinations of inputs to the nodes that effect the dependent node e.

For simplicity we have shown only two nodes n_1 and n_2 and the corresponding effect e; in general there could be one or more nodes related to e. Also, each of n_1 and n_2 might represent a cause or may be an internal node with inputs, shown as dashed lines, from causes or other internal nodes. The next example illustrates the application of the four