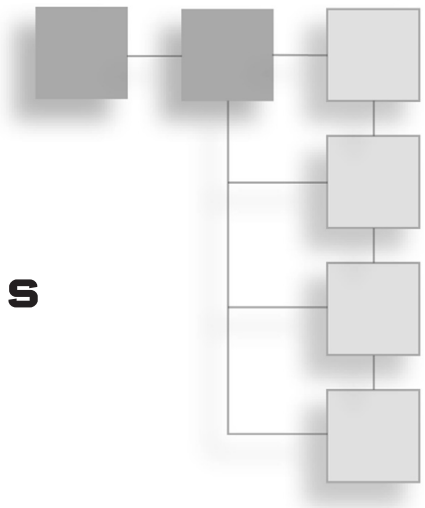
CHAPTER 12



Testing Simulations and Event Models

This chapter centers on a few techniques and tools you can employ to test simulations and event models. Testing begins with establishing approaches to depicting the cognitive characteristics of a simulation. Toward this end, you can use an approach to diagram the interactions that constitute a simulation. To accomplish this, you picture the simulation as consisting of nodes, transitions, event contexts, and pathways of interaction. To assess how completely you have employed the functionality you have developed as you have implemented your simulation, you use the notion of "cognitive saturation." To illustrate how to apply a cognitively oriented approach to testing, this chapter offers you a basic simulation testing application, *Inspect*. Using a game developed in an earlier chapter (now named *Gold Finder*), you generate test data. You then employ the test application to assess the effectiveness of the simulation. The topics covered include the following, among others:

- Conceptual foundations of testing simulation and event models
- Approaches to evaluating systems, contexts, and interactions
- Context influence
- Path transitions
- Using *Insight* to process data from *Gold Finder*
- System Cognitive Saturation Index

The Effectiveness of Simulation

As Chapter 1, "What is Simulation?" emphasized, an event model determines the extent to which your simulation can regenerate experiences. When you develop an event model, your options vary according to the description of the project you are involved with. Simulation can be characterized as having both subjective and objective aspects. A

subjective approach to simulation emphasizes experiences that are not easily repeated or summarized. An objective approach to simulation emphasizes experiences that can be repeated and summarized.

In addition to classifying them according to subjective and objective descriptions, it is also possible to regard simulations as static and dynamic. This approach to classifying simulations originates in part with designer Suguru Ishizaki, who proposed that computer application design efforts can proceed from two basic starting points. One starting point is based on traditional views embodied in print and film media. Think of a framed picture that you manipulate within a frame. You do not change the frame, only the features you see framed. Such a view of design is static.

A dynamic approach to design involves thinking of a framed picture that can change both its frame and what it depicts. This constitutes a dynamic approach to design. This approach to design views an application as a context of elements that continuously changes.

If you work in an industrial setting in which your job entails creating simulations for games, you are likely to start with a static set of specifications for the application you want to create and work toward satisfying these specifications. While a great deal of exploration might characterize such a development effort, in the end, you face what can be described as a standard design for the game. You implement the game according to the standard design. What applies to game development also applies to other computer applications, such as those used for training.

Another type of project places much less emphasis on the use of a standard template. In this case, a dynamic design applies, for the template changes with use. Suguru Ishizaki refers to such design efforts as "improvisational." With respect to simulation, improvisational design encompasses several of the ideas discussed in this book. Your development efforts involve much more of the creation of contexts in which the constraints you introduce afford those who use your application, not a static path of interaction that leads to a determined end, but rather a fairly unlimited set of alternative paths to a multiplicity of ends.

Cognitive Saturation

In this chapter, you work with a software tool that measures a quality called *cognitive saturation.* Cognitive saturation addresses both static and dynamic design activities. It describes the interactive potentials of an application, and it is based on an assessment of the extent to which given contexts of interaction possess potentials to lead application users to discover other contexts of interaction. To calculate the cognitive saturation of a system of interaction, you combine measurements of the potentials each node of interaction possesses and the significance of the transitions and pathways that connect the nodes.

To grasp cognitive saturation on a tactile level, consider a situation in which you have a set of marbles. You place the marbles on the floor and begin playing with them. The play constitutes a game. By the time you finish playing the game, you might have used all the marbles or only a few. If your play involves all the marbles and leads you to make many innovations, then the game possesses a high degree of cognitive saturation. The available marbles and the actions you take toward them represent the cognitive structure of the game. The more marbles you use and the more ways you find to interact with them, the more you realize the cognitive structure the game offers. Realization of the cognitive structure can be viewed as saturation.

Cognitive saturation measures satisfaction. If you use only a few marbles and make few innovations as you play, then the degree of cognitive saturation remains low. You see a number of marbles lying dormant and untouched after the play ends, and you do not feel a great degree of satisfaction with the actions you have taken as you have played the game.

What applies to a game involving marbles applies to software. If you develop an application that offers a hundred functional contexts and you find that, on the average, those who use your application visit only ten, then the design of your application lacks conceptual balance. It is likely that the lack of balance will become especially evident when you consider the experiences the users report. If the users make use of only ten of one hundred options and perhaps understand the application only in terms of these ten options, they are not likely to express a high level of satisfaction. Much of your design and development effort will have been wasted.

If cognitive saturation measures the effectiveness to which users actually use functionality, it also measures the extent to which a system induces its users to investigate potentials. When users investigate potentials, they find different pathways through the functionality the system offers. The pathways of interaction present users with interesting challenges that result in rich, new experiences. Such experiences alter their understanding of the significance of the events the system maps.

Systems Significance

A system interaction provides significant experiences when it in some way allows its users to change their understanding of a given set of events. As has been discussed in previous chapters, understanding possesses subjective and objective characteristics. Understanding also possesses characteristics that relate to logic. You can understand a group of events in one way, using one set of rules and one path of reasoning to arrive at your understanding. You can refer to this type of logic as *mono-modal*. On the other hand, you can understand a group of events in a multitude of ways, using many sets of rules and following paths of reasoning that vary with the feelings and perspectives you encounter as you experience the events that constitute a given context of interaction.

You can associate *subjective* understanding with multi-modal reasoning and *objective* understanding with mono-modal reasoning, but it is unlikely that any given analysis falls into one or the other category. Testing systems of interaction using both perspectives is the best approach, for you then have a way to adjust your tests so that you achieve useful results. You might use tests that are objective and mono-modal to evaluate a control that allows users to control a standard feature of a game; you might use tests that are subjective and multi-modal to evaluate options concerning the appearance and position of the control.

Testing and interaction induce you to discover new ways to understand the system as you work with it. You can use the concept of *iconic logic* to guide your activities in this respect. When you evaluate a system using a logic that you find implicit in the system, then you follow an iconic approach to logic, and your reasoning is multi-modal. On the other hand, if you undertake this activity in an experimentally controlled manner, then your activity takes on an objective description.

From an objective perspective, a simulation can originate with an explicitly defined model. The model embodies understanding, but this type of understanding consists of rules and events you have carefully described. When you participate in the simulation, you discover facts about the model that allow you to improve on the model. In this respect, a simulation brings the model to life in a dynamic way, so that you can critically evaluate the model and improve it. See Figure 12.1.

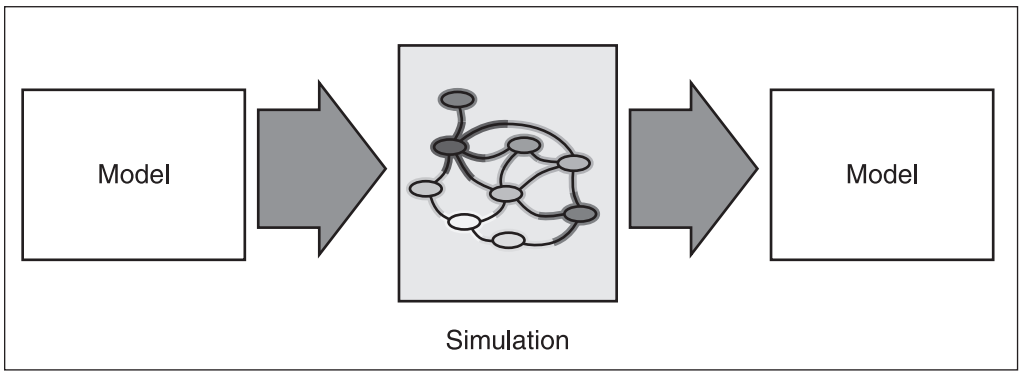


Figure 12.1 Significance arises from explicit understanding.

From a subjective perspective, what applies to a model applies to subjective understanding. The type of understanding consists of rules and events, but the rules and events are purposely left in a complex, largely undefined condition. When you participate in such a simulation, you transform and extend your understanding as an undifferentiated vehicle for mediating experience. See Figure 12.2.

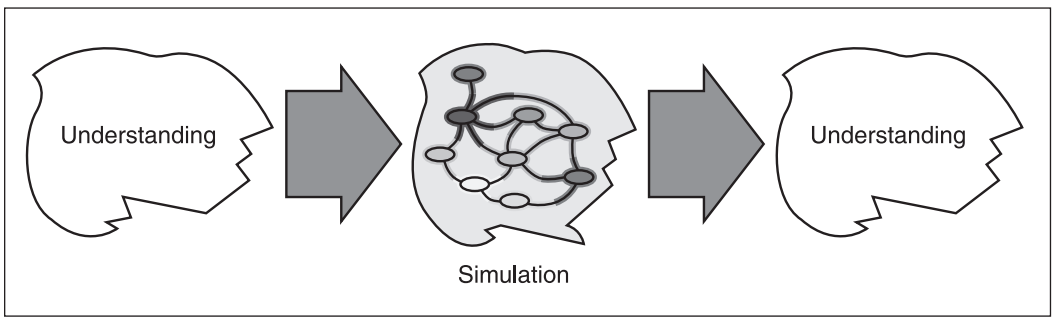


Figure 12.2 Significance arises through implicit understanding.

The Significance of Interaction

Significance depends on interaction. If you cannot interact with a system, your lack of interaction restricts you to the role of an observer or spectator. Whether you view the use of your product as a participant or a spectator heavily impacts the ways you conceptualize, design, and test your product. Clearly, participatory simulations must be evaluated in terms of the interactions they foster. Still, for many analysts, interaction constitutes a relative term. The term is relative because its meaning depends on the analytical tools the analyst applies. When you assess a system, among other things, you begin with certain assumptions about how the elements that constitute the system (nodes and transitions, for example) should be defined, measured, and tested. Ultimately, how you test for significance depends on the approach you develop to measure how interaction can be significant.

Approaches to Diagrammatic Systems Evaluation

When you assess interactive systems using multi-modal logic, you can quantitatively analyze the significance of event contexts (nodes) and scenario mappings (pathways) without heavily encroaching on the integrity of someone who is interacting with the system. At the same time, you are in a position to impose fairly high objective standards on your testing.

When you interact with a system, what you find significant depends on what you bring to the experience of interacting with the system. When you design a system, however, you face the problem of instilling in the system the play potentials that lead to rich experiences for its users. Play potentials depend to a great extent on the cognitive complexity of the application you develop. As mentioned previously, cognitive complexity relates to whether the user of the system discovers its domain of potentials. You can characterize a domain of potentials as the number of possible interactions the system makes available to its players. Alone, however, a count of the possible interactions does not reveal the significance of the experience the interactions create.

Interactions imply mappings of events, and event mappings relate to system transitions, system event contexts, and the scenarios that emerge as you follow pathways of interactions. The sections that follow provide extensive discussion of a model derived from systems theory that you can use to assess the levels of cognitive saturation a system of interaction supports. A software application, *Inspect*, embodies functionality that allows you to put this model to use to test *Gold Finder*, which you explored in Chapter 9.

Nodes

Nodes are elements within a system. As the discussion in Chapter 2 emphasized, any discernable entity or control can serve as a node of interaction. What creates a node is whether the entity interacts with other entities or allows you to interact, as the user, with it. To fairly assess the potentials the element offers for interaction involves avoiding assumptions that it does or does not possess only a specific set of interactive potentials. The case is likely to be that while the element possesses the potentials designers have designed it to possess, it possesses many others as well.

To test interactive potentials, you assign the discernable elements of a system a tentative node status. Then, to avoid testing only for assumed potentials, you can begin your examination of the nodes of a system by first isolating them from each other. You isolate them so that you can inductively establish that they do, in fact, possess properties that allow you to associate them. Figure 12.3 illustrates a collection of elements viewed in this way. The dotted line surrounding the collection signifies the tentative standing of the system of interactions that join the elements together (properly transforming them into nodes).

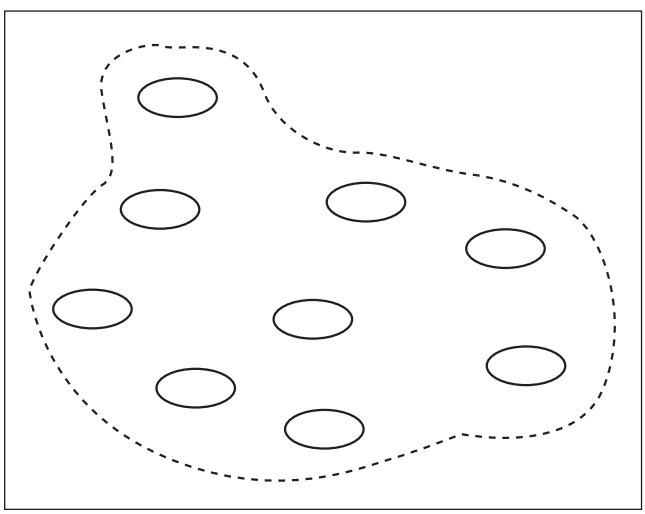


Figure 12.3 A system begins as a collection of tentative nodes.

In Figure 12.3, the nodes appear as a collection bounded by a dotted line. The dotted line is the *system boundary*. System boundaries usually represent arbitrary designations that systems analysts propose for establishing a *domain* of actions. Setting the scope of the domain usually involves designating the level of significance you wish to assign to relationships. Ecologists contend that all living processes on Earth relate to each other, so the scope of an ecological system could be extremely broad.

System boundaries represent arbitrary designations of a domain. They emerge from an *existential* definition of the *scope* of the system. In other words, you acknowledge from the start that your definition of the domain is conditional and arbitrary. To establish the validity of an existential definition, your main obligation to your audience involves stating your assumptions. In this case, the assumptions involve establishing a context in which a limited number of nodes using a limited number of interactions create a context in which levels of cognitive saturation can be quantitatively assessed.

Naming Nodes

When you designate an element as a node, it is important to create a naming system that does not immediately imply relationships. One approach is to use the number of nodes and apply a combinatorial algorithm to create an array of random names. In Figure 12.4, for example, if you consider that the interaction network consists of nine nodes, then you can take the square root of nine to determine the minimum number of letters required to generate unique names beginning with the same letter. If you generate a complete matrix that encompasses all possible names, you have 27 to work with.

Having developed a set of names, you can then apply the names to the prospective nodes. Again, for purposes of objective analysis, you should randomize the application of names as much as possible. Numbers and letters used with periods or other punctuation immediately imply an order. Generally, if you select names from a table of the type Figure 12.4 illustrates, then you are likely to be able to name nodes without at the same time assigning an implied order to them. Figure 12.5 illustrates the appearance of the collection of prospective nodes after names have been applied.

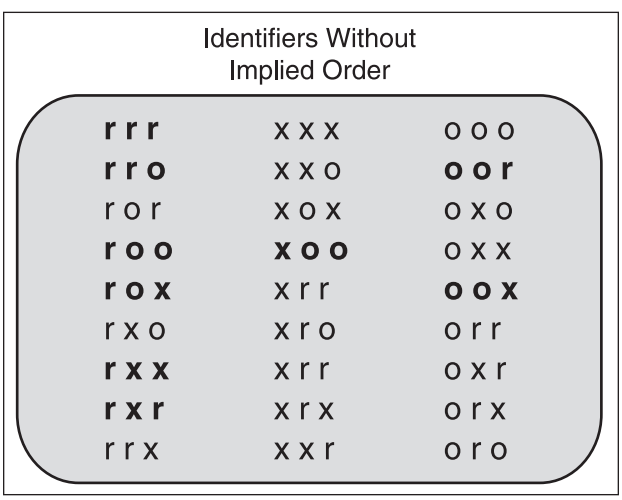


Figure 12.4 Name nodes so that you imply no order.

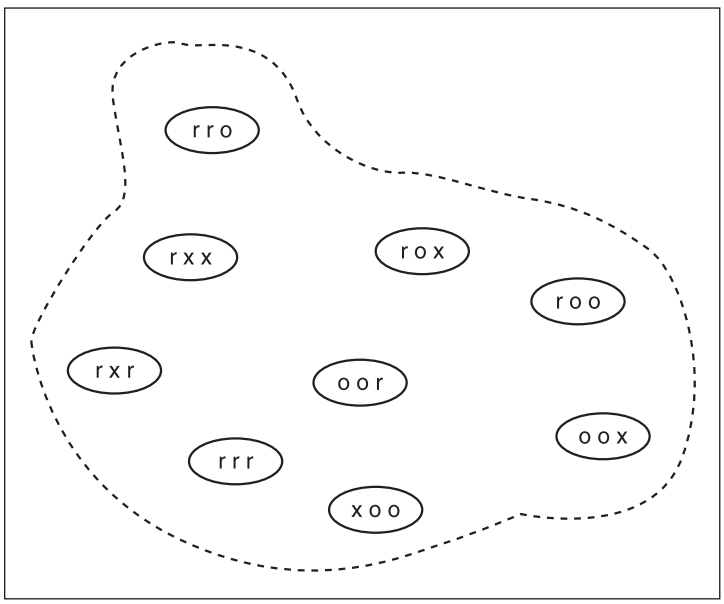


Figure 12.5 Apply random names to identify prospective nodes.

Transitions

Transitions represent relationships between nodes. To speak generally about transitions, you can call them connectors. Connectors and transitions are analogous to elements and nodes. A connector lacks directionality. Directionality designates that the connector occasions the movement of information from one node to another.

The construction of a system results from the application of transitions to the nodes. In the system Figure 12.6 illustrates, transitions relate nodes to each other. Each transition represents a flow of information, and to determine whether a transition exists between two nodes, you ask whether one node receives information from or transmits information to another node. Nodes transmit and receive information according to the boundary, or scope, of interactions that existentially characterize the system.

You can use a number of symbols to represent connectors and transitions. Curved lines without arrows designate *connectors*. Curved lines with arrows designate transitions. See Figure 12.7.

The Focus of Awareness

A transition indicates that information flows between two nodes, but this is only part of the picture. When information flows between two nodes, the node toward which the arrow points indicates the direction of information flow and the movement of the focus of awareness that accompanies the flow of information. The focus of awareness refers to the focal point of *cognitive awareness*—awareness as established by the structure of interaction the system sustains.

Consider what happens if you use a pencil to examine the transitions in the diagram. Using the pencil, you trace the paths of arrows from node to node. (See Figure 12.8.) When you trace the paths of the arrows in this way, your activity creates a *focal point of interaction*. The tip of the pencil at any given point represents the focal point of cognitive awareness that you sustain as you trace the path of the transitions.

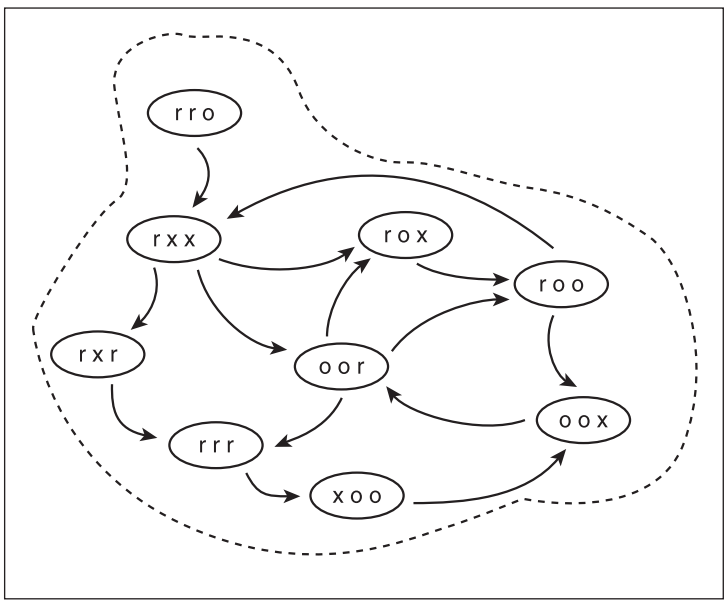


Figure 12.6 A system consists of transitions and nodes.

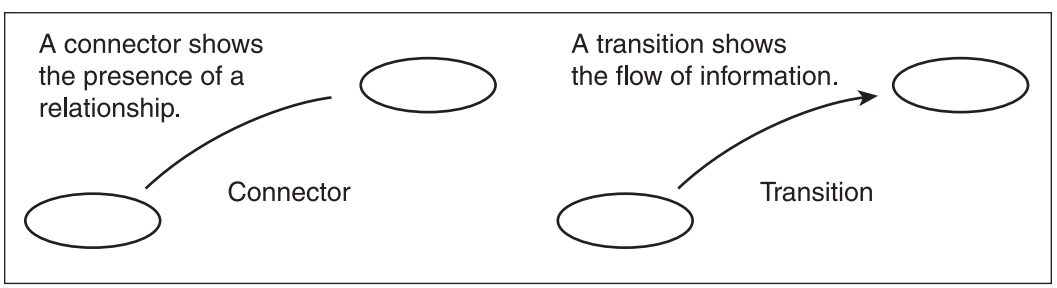


Figure 12.7 A transition differs from a connector because a transition indicates directionality.

The movement of the pencil provides a way to understand the directionality of the arrows. You carry information from one node to the next, and the significance of the information you carry is relative to the position you occupy in the system. As you move through the system from node to node, the information you gather allows you to make decisions about the directions you take. With each decision, your awareness grows.

Figure 12.9 illustrates that the transition from node oor to node roo shows that node oor contributes information to node roo. If you are tracing this path, when you reach roo, you are aware that you have passed through oor and you are aware of the potentials oor holds with respect to roo.

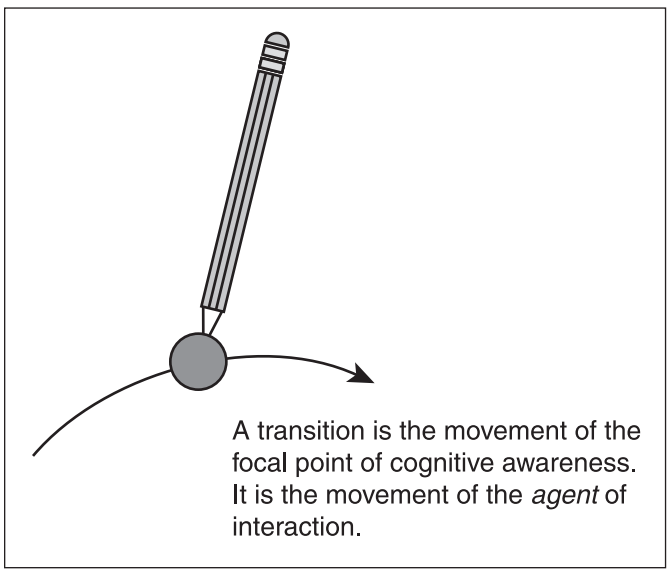


Figure 12.8 The context of cognitive awareness moves as you trace the path of the transitions.

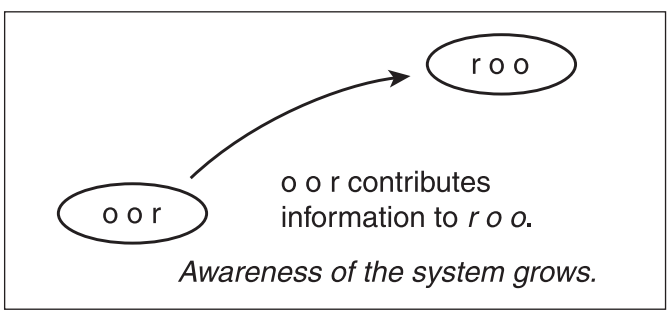


Figure 12.9 Transitions represent increased awareness.

If you consider that the transition from one node to the next represents a growth in your awareness of the system, then you also have a way to understand that the direction of the arrow represents, not the solicitation of information, but the growth or transformation of understanding. The nodes are not in themselves agents of intelligence. They represent moments in which decisions can be made about how information is to continue to flow.

A transition implies that an agent acquires and transforms information as he or she moves from one node to another. From the standpoint of information theory, this acquisition and transformation of information is accompanied by an *acknowledgment* of some type. You can view acknowledgement as a focus of awareness accented by a moment of new understanding.

Abstracting and Diagramming Acknowledgment

As becomes evident in the discussion a little later in the chapter, you can diagram an acknowledgement if you picture it as an event context. You can picture an event context as a combination of several factors. Consider, for example, what happens if you stand on the peak of a hill. You can turn on your heel and see a varied set of views. A simple turn on your heal gives a different view, but each view differs according to how you think about it. If in one direction you see a park, you might begin planning an outing for a picnic. If you see a river in a second direction, you might think about kayaking. If you see houses in a third direction, you might start planning a move or a construction project.

What you see is not simply a result of having turned on your heel. You might see a park as an opportunity for a picnic, but you might also remember that you were once mugged in a park. Again, you might remember a time when you played a softball game in a park. In other respects, you might see a valley you think a wonderful spot for a house, but you might also be thinking about the opportunities afforded by a raise or a new job. To assess the significance of your interaction with the settings you see as you turn on your heel atop the hill, you must consider the significance of memories, what you see, your sense of your abilities in the future, and among many other things, the general set of chance views your turns on your heels offer you.

How you account for your thoughts as you turn atop the hill is analogous to what happens when you analyze a transition. Each factor that you include in your analysis represents an abstraction of acknowledgement. As Figure 12.10 illustrates, you use the curved arrow to represent a moment that almost inevitably consists of a complex exchange of information and complex form of acknowledgement.

Contexts of Interaction

You can begin to see nodes as giving shape to a system when you see the nodes as contexts of interaction. Contexts of interaction emerge from the ways that nodes relate to each other to provide information, foster the growth of awareness, and set occasions for decisions. (See Figure 12.11.) They do so as a system, and the shape the system assumes depends on the ways that the contexts of interaction relate to each other. The great benefit you derive from this view of the system consists of understanding how the parts and the whole relate. As the elements of the system (the parts) join to create the system (the whole), the system subordinates the elements to the scope of activity that the system existentially represents.

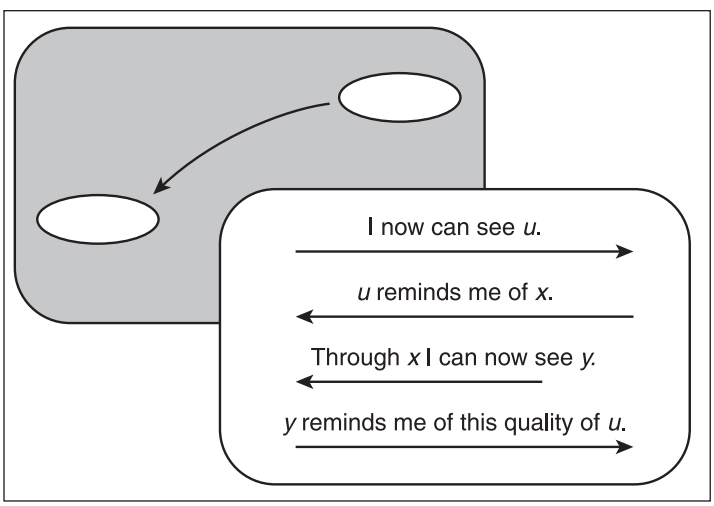


Figure 12.10 Transitions imply acknowledgements.

At first, your view of the whole system might lead you to concentrate specifically on the connections between specific nodes, so that you resist allowing the elements to merge into the whole. There is a benefit to be derived from this view. The tension you feel as you view the whole and in relation to the parts allows you to discern how your awareness of the system changes depending on the point of focus.

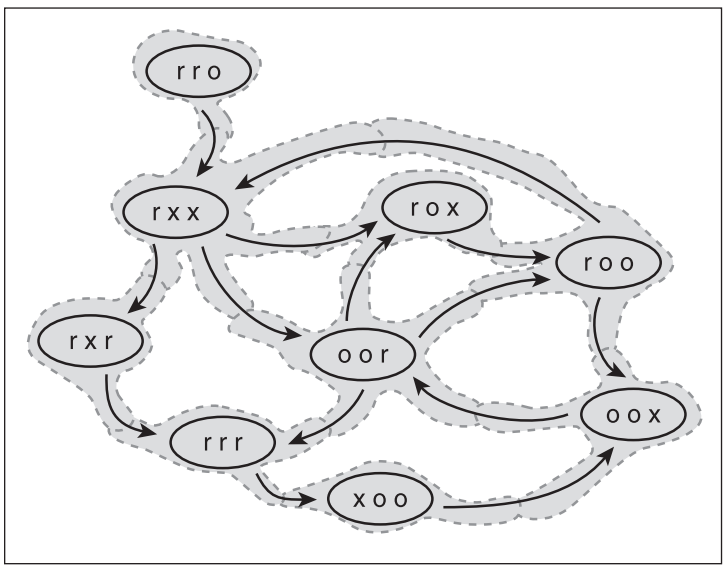


Figure 12.11 The view of the whole begins to subordinate the elements.

In Figure 12.12, the double circles represent moments of transition. A moment of transition constitutes what might be viewed as a testing or confirmation of understanding. Imagine moving the pencil tip from rox to roo. The awareness you possess as you make this movement can vary according to the path you follow. Your most recent transition might have been from rxx to rox. Or it might have been from oor to rox. What you find significant as you reach roo depends on the path leading to it. With each new movement, the center of focus for the system changes, and with this change comes a new view of the system, a new awareness of the reality it creates for someone who participates in it.

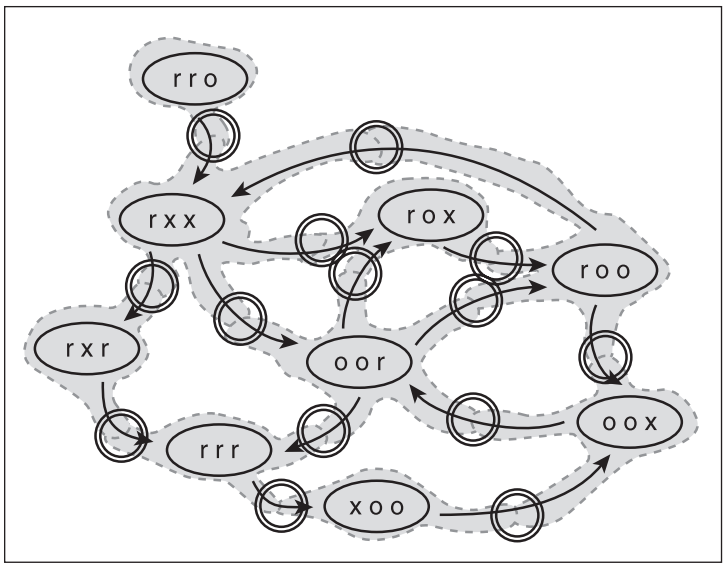


Figure 12.12 Junctions of interaction might draw your attention as you first examine a system.

As your awareness of the whole increases, so does your awareness of the parts. The awareness consists of acknowledgements given to contexts of interaction. Contexts of interaction represent nodal clusters of transitions. These clusters are dynamic and their significance takes shape according to the paths that create them. Figure 12.13 illustrates the emergence of nodes as dynamic contexts of interaction.

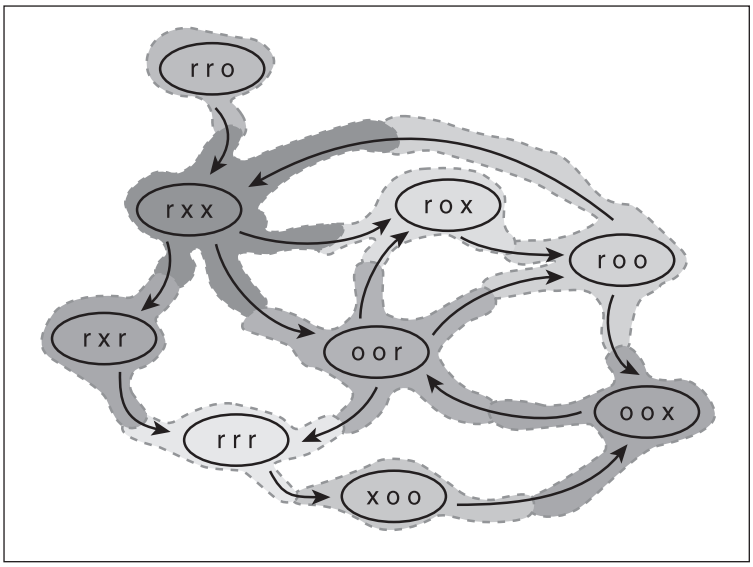


Figure 12.13 The significance of nodes emerges from system interactions.

Determining Significance

You can determine the significance of each context of interaction if you first isolate it from other contexts of interaction. In Figure 12.14, for example, node rxx now appears in the foreground. When positioned in the foreground, node rxx has a distinct form, one that abstractly represents its potentials for dynamically interacting with other nodes.

One way to picture the situation that now emerges involves considering for a moment the concept of a hologram. A hologram is a photograph, but it differs from other photographs because each part of a hologram possesses the information you require to re-create the image of the whole. Normally, photographs consist of a set of dots colored to provide a single image. A hologram is different. Imagine, for instance, that you have a holographic picture of an apple. If you cut the hologram into pieces and then illuminate any one of the pieces using a laser, you will see that each piece creates a whole image of the apple. The part re-creates the whole because each part of a hologram possesses the information you require to re-create the whole.

Figure 12.14 is not a hologram. Still, node rxx possesses the shape it possesses because it interacts with the other nodes in the system in a specific way. In one sense, the shape of the whole depends on the shape of the part. On the other hand, the shapes of the parts depend on the shape of the whole. Transitions converge in node rxx in a specific way. Transitions converge in all the other nodes in specific ways. From these convergences—these contexts of interaction—arise the distinctive character of the system as a whole.

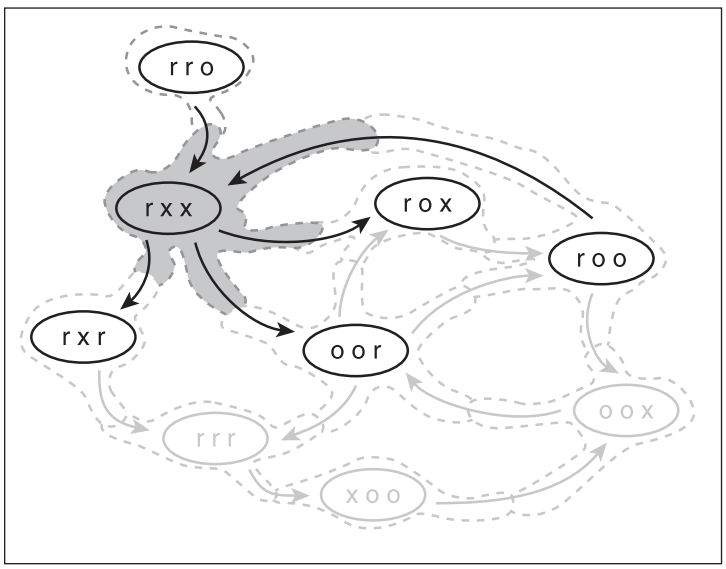


Figure 12.14 Isolating a context of interaction allows you to explore its significance.

note

A gateway node is a node that has only one connector and that you do not need to depict as receiving feedback. Node rro is a gateway node. Such nodes provide ways to define open systems without at the same time making evaluations of the system impossible.

Context Influence

A node can be described as a context of interaction. You can begin to evaluate the significance of a given context of interaction by considering both how it influences other contexts of interaction and how other contexts of interaction influence it. This type of relatedness constitutes the general connectivity of the node (NC). Influence flows in two directions. Movement from the node consists of an output (NO). Connectivity can also be viewed as reception of information (NR). The number of connectors that tie a context of interaction to other contexts of interaction provides a quantitative assessment of this connectivity. The value you arrive at when you assess connectivity in this way gives you the context influence (CI) of the node you are examining.

CI = NR x NO x NC,

where NR, NO, or NC = 1 if NR, NO, or NC = 0.

This formula indicates that the product of the number of node receptors (NR), the number of node outputs (NO), and the number of node connectors (NC) provides the influence of the context of interaction (CI). Figure 12.15 illustrates an event context and the factors involved in determining its influence. Table 12.1 provides a summary view of the values involved and how they can be used to determine the context influence of a given event context.

In Figure 12.15 and Table 12.1, the formula used to determine the influence of the context of interaction (CI) contains an indexed value. For example, notice that CI(rxx)[1] has an index of 1. The 1 indicates the *order* of the analysis. Order is something akin to the internal scope or boundary of analysis. The order is the degree of analysis that you conduct to establish a given value. In this case, as Figure 12.15 shows, you evaluate the node only to discover its significance relative to the nodes with which it is most immediately connected. This is the first order of nodes.

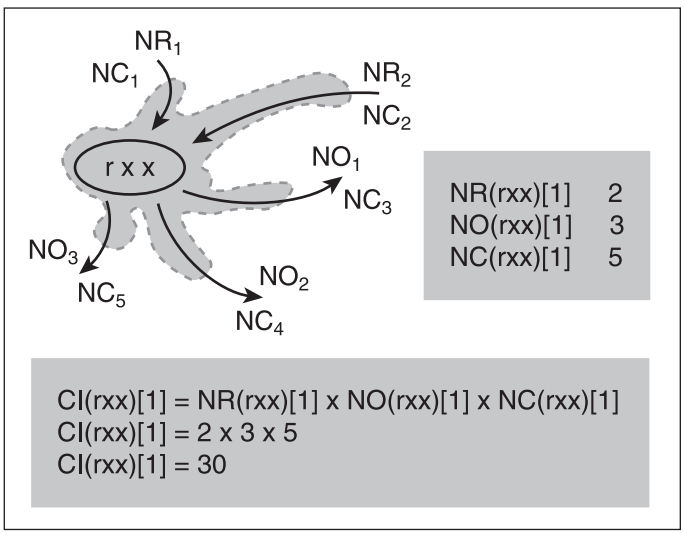


Figure 12.15 The influence of a context of interactions can be discovered by considering its connectors.

If you wanted to take the analysis to a second order, you could assess the significance of each of the transitions of each of the nodes with which rxx is connected. This order of evaluation lies beyond the scope of this discussion.

Table 12.1 Terms for Influence and Significance

Term	Abbreviation	Discussion
Context Influence	CI(node)[order]	Context Influence for a given node relative to a given order of evaluation. CI = NR x NO x NC, where NR, NO, or NC = 1 if NR, NO, or NC = 0.
Node Reception	NR(node)	The number or transitions that show a node receiving information.
Node Output	NO(node)	The number of transitions that show a node supplying information to another node.
Node Connectivity	NC(node)	The total number of connectors for a given node.
Context Significance Influence	SI(node)[order]	The product of the contexts of influence that contribute a given context of interaction. $S = CI_1 \times \ldots \times CI_n$

Systems Significance

Using the information that you gain from the quantitative influence of given contexts of interaction, you can move on to assess the influence the overall system possesses. The influence of the overall system consists of the sum of the influences of the nodes that constitute the system. Figure 12.16 provides a summary view of event contexts and the influences that characterize them.

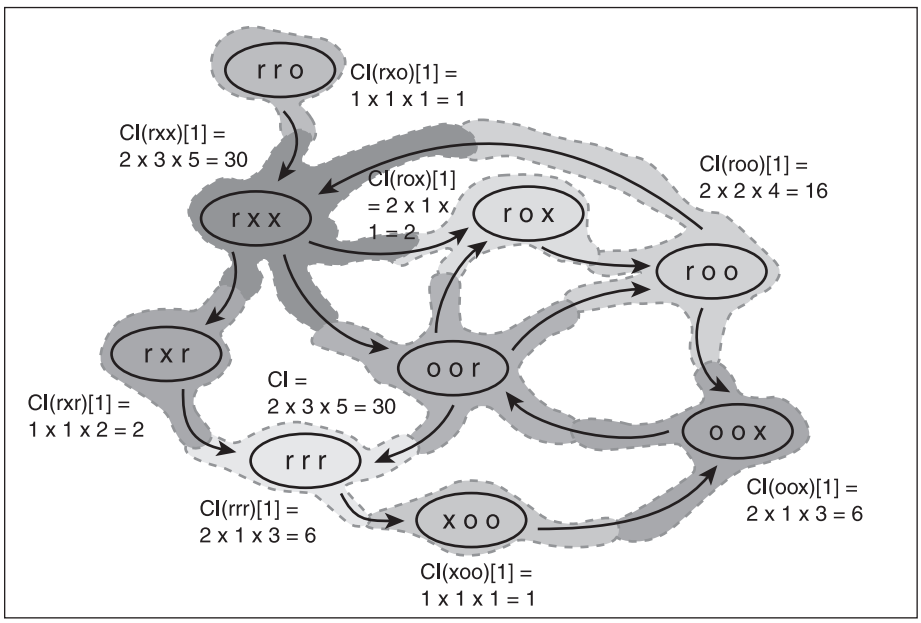


Figure 12.16 Cognitive saturation (SI) for cognition is the sum of the influences of the constituent event contexts.

Path Transitions

Evaluating the significance of a system involves evaluating both its nodes and its transitions. Just as the influence of individual contexts of interactions can be summed to obtain a view of the overall influence of the system, the values of transitions can be combined to evaluate the significance of different paths. A path consists of the set of transitions a given scenario includes. Figure 12.17 depicts a scenario. The scenario encompasses a series of sequentially numbered transitions. The subscripts indicate the order in which the focus of attention approaches the transitions. For the segments T_5 and T_9, you see that path uses the same transitions.

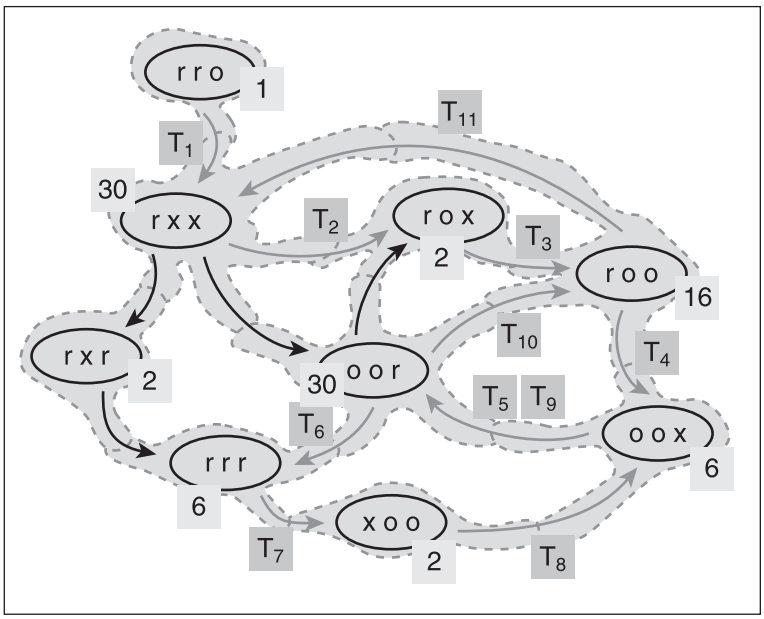


Figure 12.17 Path values are calculated using context significance.

Table 12.2 summarizes the terms that apply to the assessment of transitions and paths. The values for the individual transitions (T) are combined to determine the path value (PV). The path value represents only the paths of a given scenario. Total path value for the system (TPV) represents all the transitions that constitute the system.

Table 12.2 shows you that several types of paths can be characterized in special ways. For example, some paths allow you to move through the transitions a system offers without redundancy. You can refer to such paths as *progressive*. The path shown in Figure 12.17 falls into a second category, that of a *constrained* path. Such a path allows you to repeat transitions up to twice each, and it generally moves forward in a fairly efficient way from start to terminal point.

After constrained paths come *regenerative* paths. Such paths can repeat a given transition a large number of times. The limiting number is a number that allows you to complete a path the system provides. (Picture turning right three times to complete the circuit of a square or seven times to complete the circuit of an octagon.)

A final path is identified as *degenerative*. This form of path is not necessarily doomed to failure, but it allows you to repeat single transitions a number of times far in excess of the number of transitions that characterize the system. The number can even go to infinity.

Table 12.2 Types of Paths

Term	Abbreviation	Discussion
Progressive Path		The order is CI1 -> C12 -> … ->CIn, and no path (transition) is repeated more than once (TC <= 1 >)
Constrained Path		The order is CI1 -> C12 -> … ->CIn, and no path (transition) is repeated more than twice (TC <= 2>)
Regenerative Path		The order is CI1 -> C12 -> … ->CIn, and no path (transition) is repeated more than the context count (CC) for the system (TC < = CC>)
Degenerative Path		The order is CI1 -> C12 -> … ->CIn, and one path (transition) becomes infinite (Tn = Infinity>)
Transition Value	(TV)	The value is the sum of the two contributing contexts. TV = CIn + CIn+1.
Path Value	(PV)	The sum of the path transition values that constitute a given scenario. PV = Sn{TV1 + TV2 +…+ TVn}.
Total Path Value	(TPV)	The square of the sum of the smallest transition value added to the largest transition divided by 2. TPV = (TVmin + TVmad / 2)2

Calculating the Relative Path Value

Drawing from the information Table 12.2 provides, you can determine the total path value the system provides. To accomplish this, calculate a value you can use to represent the sum of all possible paths in the system. This requires that you find a reasonable limitation to the number of paths you consider. Toward this end, you can sum the minimum and maximum transition values, divide this sum by two, and then square the result. This results in the *total path value* for the system (TPV):

$$\text{Total Path Value} = \left(\frac{\text{Minimum Transition Value} + \text{Maximum Transition Value}}{2} \right)^2$$

You can also calculate the *traversed path value* (PV) of a path a scenario designates. Again, drawing from the information that Table 12.2 provides, to accomplish this, you need only to add up the values you find for each transition (T) in the path you have taken. Here is the formula:

$$\text{Traversed Path Value} = T_1 + \ldots + T_n$$

To find the *relative path value* (RPV) of the path taken, you need only to find the ratio of the traversed path value to the total path value:

$$\text{Relative Path Value} = \frac{\text{Traversed Path Value}}{\text{Total Path Value}}$$

Figure 12.18 shows the result of the calculations for the system depicted thus far in this chapter. The relative path value is 0.4156378. The large number of decimal places can be reduced to accord with your needs. In more complex systems such as the one that you deal with a little later on in this chapter, you require a large number of decimal places because your interaction scenario represents a small fraction of the total path value.

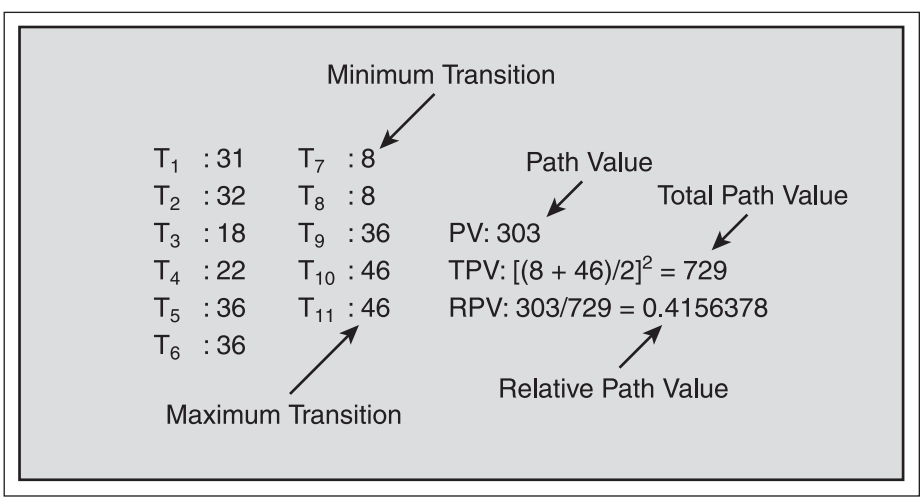


Figure 12.18 Transitions pertain to specific and general path potentials.

Calculating the Relative Context Value

The relative context value is the ratio of the nodes included in a given scenario to the nodes that comprise the entire system. To obtain the relative context value (or significance) of the session, you take the number of contexts the traversed path includes and find the ratio of this value to the number of contexts the system includes:

Total Context Count = 9
Actual Context Count = 8
Relative Context Value = 8/9 = .888

The system Figure 12.13 depicts features 9 event contexts. Since 8 of the event contexts lie in the path the scenario designates, the ration is 8/9, or 0.888.

Calculating the Cognitive Saturation Value

If you know the relative context value and the relative path value, you can then calculate the level of cognitive saturation that characterizes a given user session. The cognitive saturation level is the average of the sum of the relative context value and the relative path value.

$$\frac{\text{Relative Context Value} + \text{Relative Path Value}}{2} = \text{Cognitive Saturation}$$

Given the figures used so far, when you apply this formula you arrive at a saturation of approximately 0.651.

$$\frac{.415 + .888}{2} = \text{Cognitive Saturation}$$

A Trial Run of Inspect

To place the discussion provided thus far in a context that allows you to arrive at a more concrete, tactile grasp of the meaning of cognitive saturation, access the Inspect_Sample folder in the Chapter12_Code folder. In this folder is another, called Inspect_Sample. Open this folder and the Inspect.exe file. Figure 12.19 displays the result.

Inspect is a C# application. You can find the source code and a Microsoft Studio .Net project for *Inspect* in the Inspect folder in the SimulationSrc directory for Chapter 12. Due to limitations of space, this chapter does not provide a discussion of how to implement the code for the application. Instead, the focus is on how to use *Inspect* in conjunction with a game to derive data that you can use to assess the cognitive saturation of the game.

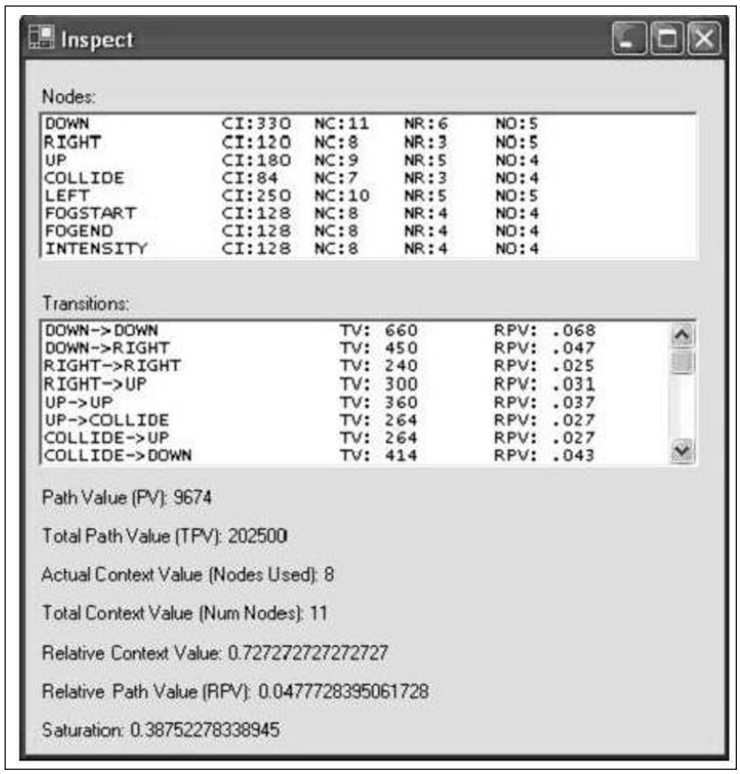


Figure 12.19 *Inspect* provides you with a view of data you can use to assess application performance.

The data that *Inspect* displays consists of five types. Here is a breakdown:

- **Node data.** Each node provides a context of interaction. Nodes communicate with other nodes by passing information to them (NO). They also receive information (NR). They have a general context weight based on their role as connectors (NC). Using the formulas shown in Table 12.2, you can assess the interactive context influence (CI).
- **Path data.** Each path possesses significance on the basis of the two nodes it connects (PV) and overall path to which it contributes (RPV).
- **Summary node data.** Each node offers a point at which you can make a decision regarding what to do next. It is not enough to say that a node offers only an opportunity to make a decision. Nodes can differ in significance depending on the number of choices they offer and the significance of the paths to which the choices lead.
- **Summary path data.** The total path value of the system describes what might be considered the maximum potential that a system offers to extend pathways. Picture this in terms of the number of kilometers of trails a park might offer. After a time, you begin hiking the same trails, even if you do so in differing orders.

- **Cognitive saturation.** The way that nodes and paths merge to shape the interactive potential of a game can be assessed using cognitive saturation. If you finish a user session after interacting with only a few nodes and pathways, then the cognitive saturation of the application is probably fairly low. On the other hand, if you end up exploring many of the nodes and paths available to you during your session of interaction, then the cognitive saturation of the application is probably fairly high.

General Trends in Cognitive Saturation

Figure 12.19 represents data drawn from a single session of play involving *Gold Finder*, which is a slightly altered version of the game you developed in Chapter 9. This session results in a cognitive saturation reading of around 0.39. This figure gains significance when you assess its value relative to data collected from a number of play sessions. Consider Figure 12.20, which graphically represents a trend you might observe if you collect data about multiple sessions.

Assume that you acquire a game and over time have different experiences with it. For example, if at first you abruptly conclude the game after starting to play it, you are likely to experience little satisfaction, and the level of cognitive saturation that characterizes your play session is close to zero. Given this start, consider what happens if you again play the game, this time exploring more options and learning more about its features. You make more choices and follow more paths. The cognitive saturation moves into the middle area of the curve. Chances are that you experience greater satisfaction.

If you practice a few more times, you might find that you discover a multitude of features and paths of interactions. Your level of satisfaction increases, as does the level of cognitive saturation that characterized your session of interaction.

Running Inspect

Now that you have had a chance to examine how cognitive saturation can be pictured on an abstract level, you are ready to engage in some practical exercises. These exercises allow you to accomplish two primary tasks. First, you alter the application you developed in Chapter 9 so that it can write data to a file. The data you write to a file allows you to track your interactions with the application. Second, you alter the application so that it automatically opens *Inspect* to analyze the data you have generated. To begin this activity, access the SimulationSrc folder in the Chapter12_Code folder, and then look in the Listing12_01 folder for the project (*.sln) file. Compile the project. Figure 12.21 illustrates a view of *Gold Finder*, which you last saw in Chapter 9.

To begin generating data, you can follow a test scenario. Software testers employ use case to map test scenarios. Figure 12.22 illustrates a test scenario to follow as you play *Gold Finder*. While you can set up a use case to help you maintain a complex set of interactions during testing, you can also employ the use case as a tool to help you assess the data that you obtain through *Inspect*.

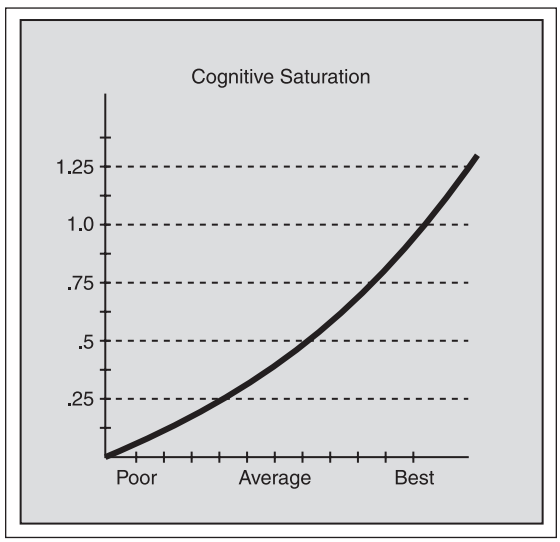


Figure 12.20 Cognitive saturation gauges interaction and satisfaction.

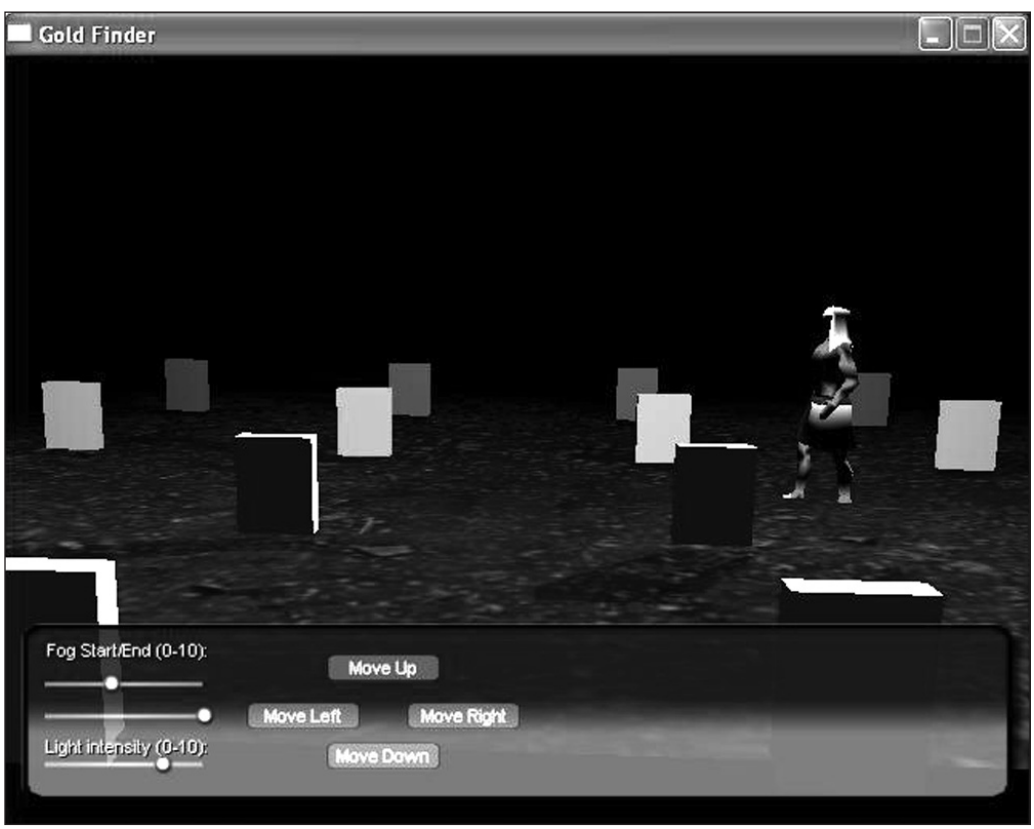


Figure 12.21 The Gold Finder game from Chapter 9 makes a reappearance for testing.

Use Case Name: Play scenario for cognitive saturation testing
Player (Actor) Context (Role): Player
Precondition(s): Game up, ready to play.
Trigger(s): Start of game.
Main Course of Action: 1. Adjust start fog to the middle of the slider. 2. Adjust the end fog to the start of the slider. 3. Set the Light Intensity to high. 4. Move the camera on the z axis until you see six rows down and six across. 5. Walk to the last row, turn right. 6. Walk to the far right row, turn right (so you're facing forward). 7. Walk forward two rows, turn left. 8. Walk all the way across the cemetary, turn right. 9. Walk to the last row, turn right. 10. Walk to the center row, turn right. 11. Walk forward until you are even with the gold stone, turn left. 12. Walk until you collide with the stone.
Alternate Course(s) of Action: 1–12. If you need to correct actions, do so. But keep to the plan.
Exceptional Course(s) of Action: n/a

Figure 12.22 A test scenario for playing *Gold Finder*.

If you follow the test scenario Figure 12.22 provides, when you perform step 12, which involves guiding the character to the golden tombstone, the game displays a message box. The message box informs you the data from your user session has been collected and can be displayed. Figure 12.23 illustrates the message box.

You might have to move other windows on your desktop around before you can see it, but the *Inspect* window appears as soon as the game ends. The game ends when you collide with the gold stone, and the message box informs you that *Inspect* has executed to show you the data your session has generated. To close the message box, click OK. This leaves you with the *Inspect* data window. Figure 12.24 illustrates the *Inspect* data window that results if you follow the use case Figure 12.22 illustrates.

The upper data pane displays data on the nodes or event contexts that *Gold Finder* encompasses. The lower data pane displays data on the transitions that characterize your interactions with the game. At the very bottom, the *Inspect* window displays the data used to calculate the level of cognitive saturation that applies to your use of the application.

Figure 12.23 A message box informs you that the application has generated data and invoked *Inspect*.

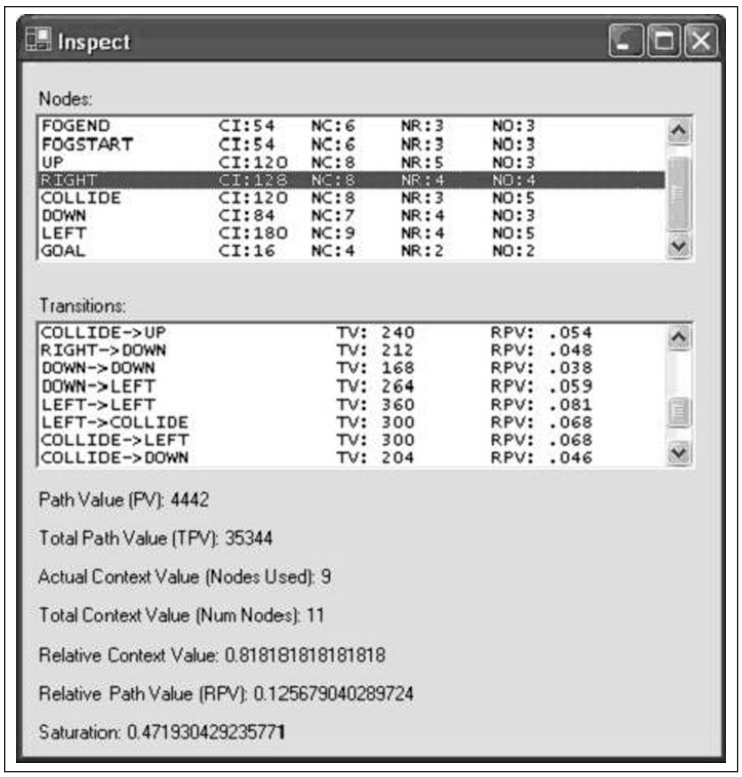


Figure 12.24 Inspect shows you the data that has been logged for your user session with *Gold Finder*.

Assessing Messages

As the earlier sections of this chapter have discussed, to a great extent you can assess interaction in terms of event contexts (nodes) and pathways (transitions). *Gold Finder* provides several contexts of interaction. You can identify the contexts of interaction if you assess the messages the application can process. As is shown in greater detail a little later on in this chapter, the application class for *Gold Finder* features eleven messages. Each of these messages can be the basis of an event context. To fully identify the context, however, you must assess how one event leads to another. As Figures 12.14 through 12.16 illustrate, the significance of an event context depends on the transitions it maintains with other nodes and the complexity of the nodes with which it communicates.

As Figure 12.25 shows, `UP`, `RIGHT`, `LEFT`, and `GOAL` events display the greatest potentials. `INTENSITY` and `FOG` tend to be less complex, as is the game's end (goal) event. Event contexts that tend to character actions frequently performed tend to have greater overall significance than other actions. In this respect, it might be said that frequently performed actions should have greater significance than other actions. Certainly where design of an application is concerned, you work hard to ensure that the users of your application find that the actions they frequently perform are logical and laden with cognitive significance.

Node/Event Context Data

INTENSITY	CI: 30	NC: 5	NR: 2	NO: 3
FOGEND	CI: 54	NC: 6	NR: 3	NO: 3
FOGSTART	CI: 54	NC: 6	NR: 3	NO: 3
UP	CI: 120	NC: 8	NR: 5	NO: 3
RIGHT	CI: 128	NC: 8	NR: 4	NO: 4
COLLIDE	CI: 120	NC: 8	NR: 3	NO: 5
DOWN	CI: 84	NC: 7	NR: 4	NO: 3
LEFT	CI: 180	NC: 9	NR: 4	NO: 5
GOAL	CI: 16	NC: 4	NR: 2	NO: 2

Figure 12.25 Event contexts reveal trends of significance.

In addition to event contexts, the interactive potentials of your application depend on the transitions between event contexts. As Figure 12.26 illustrates, transitions establish paths, and paths have significance according to the value of the nodes they connect. Paths also have significance with respect to the extent to which they account for the total number of possible paths that a system offers.

As Figure 12.26 illustrates, *Inspect* tracks the actual path you follow as you interact with *Gold Finder*. If you examine this data in detail, you can see that it accounts for not only the movement of the arrow keys but also the settings you apply to the lights.

In each case, the records that *Insight* generates relate transitions as passages between event contexts. *Inspect* employs the algorithms discussed earlier in the chapter to arrive at the values of transitions. Transitions that lead to collisions often show high significance. Turns tend to have values that fall into the middle range. Adjustments to the lighting tend to have the lowest significance.

Transition/Path Data

INTENSITY –> INTENSITY	TV: 60	RPV: .014
INTENSITY –> FOGEND	TV: 84	RPV: .019
FOGEND –> FOGEND	TV: 108	RPV: .024
FOGEND –> FOGSTART	TV: 108	RPV: .024
FOGSTART –> FOGSTART	TV: 108	RPV: .024
FOGSTART –> UP	TV: 174	RPV: .039
UP –> UP	TV: 240	RPV: .054
UP –> RIGHT	TV: 248	RPV: .056
RIGHT –> DOWN	TV: 212	RPV: .048
DOWN –> DOWN	TV: 168	RPV: .038
DOWN –> LEFT	TV: 264	RPV: .059
LEFT –> LEFT	TV: 360	RPV: .081
LEFT –> COLLIDE	TV: 300	RPV: .068
COLLIDE –> LEFT	TV: 300	RPV: .068
COLLIDE –> DOWN	TV: 204	RPV: .046
LEFT –> UP	TV: 300	RPV: .068
LEFT –> GOAL	TV: 196	RPV: .044
GOAL –> null	TV: 16	RPV: .004

Figure 12.26 Messages can be used to track paths.

Code Implementation for Testing

To test the *Gold Finder* game, you add a few lines of code to it to generate test data. The lines of code you add complement existing code. Your work begins with the message processing capabilities you created when you developed the game in Chapter 9. After attending to message generation, you add a few lines of code to invoke *Insight*. You do not need to perform any work to create `Insight`. The executable already resides in the Bin directory for the code for this chapter.

Identifying Messages

As discussions in previous chapters have emphasized, you create attributes in the `CMyApplication` class that enable you to track messages. When you create class attributes to track messages, you also have a way to log each message your application processes. You

can log messages because upon declaration you associate each attribute with a unique number. Here is the attribute list for the `CMyApplication` class, which you find in CMyApplication.h:

```
static const int IDC_START = 0;
static const int IDC_END = 1;
static const int IDC_STATIC = 2;
static const int IDC_INTENSITY = 3;
static const int IDC_STATIC2 = 4;
static const int IDC_UP = 5;
static const int IDC_RIGHT = 6;
static const int IDC_DOWN = 7;
static const int IDC_LEFT = 8;
static const int IDC_GOAL = 9;
static const int IDC_COLLIDE = 10;
```

Logging Messages

Whenever you play *Gold Finder*, the actions you take are logged to a file. Logging your actions to a file requires that you perform three tasks. First, you set up a log file. Second, you create a function that writes numbers to the file that identifies your actions. Last, you create a function that signals the termination of a game and invokes *Inspect* so that the data you have logged can be processed and displayed to you.

Setting Up a Log File

To set up a log file, you make a small addition to the `CMyApplication` class. This addition consists of the declaration of `FILE` attribute. The attribute identifies a file stream—an identifier that holds a pointer to which you can write data. Here is the code for the attribute in CMyApplication.h:

```
// Output file
FILE *m_fpOutput;
```

note

As a note for C++ buffs, to include file i/o, you can adopt the newer convention of including C header files using the names they have been assigned under the latest ANSI C++ specification. In this respect, stdio.h becomes cstdio. Since these header files are now in the std namespace, you no longer append *.h to identify them.

To use the file handle, you must first associate it with a file. You perform this work in the `CMyApplication::Initialize()` function. You use the C language `fopen()` function to accomplish this. This function requires two parameters and returns a pointer to a file

stream, which you assign to `m_fpOutput`. The first parameter of `fopen()` identifies the file to which you want to write data. The second identifies the mode in which you want to use the file. The typical modes are read (`"r"`), write (`"w"`), and append (`"a"`). You can also use `"t"` and `"b"` in conjunction with the basic file mode specification to indicate whether you want to write in text or binary mode. In this instance, you use `"wt"`, for "write text":

```
// Open the output file
m_fpOutput = fopen("data.txt","wt");
```

Writing Data to the File

Having designated a stream that you can write to, you can then begin to write data to it. To accomplish this task, you make use of the C language `fprintf()` function, which you call in the scope of the `CMyApplication::OnGUIEvent()` function. The procedure is fairly simple. Each time you invoke the `OnGUIEvent()` function—which is every time you issue a message—you call `fprintf()` to write to the file. Here is a snippet of code to illustrate this activity:

```
void CMyApplication::OnGUIEvent(UINT nEvent, int nControlID,
                                CDXUTControl *pControl)
{
      // Log the message
      if(m_fpOutput)
      {
            fprintf(m_fpOutput, "%d\n", nControlID);
            fflush(m_fpOutput);
      }

//. . . lines left out
}
```

The `fprintf()` function requires two parameters, but it is a function that takes a variable number of arguments. The first identifies the file stream. The second is a character string, possibly containing escape sequences describing the types required for succeeding arguments. In this case, `"%d"` designates a signed or decimal integer. All parameters after the second provide the types specified in the second argument in the correct order to fill in the string.

The data you write consists of the unique identifiers for each of the messages you process when you play the game. The `nControlID` parameter of the `OnGUIEvent()` function provides these identifiers. Each time you issue a message, you call `fprintf()` to write data to the file. After writing the data to the file, you call the C language `fflush()` function to clear the stream for the next message.

If you access Inspect_Sample and open the data.txt file, you can view the result of this activity. The information you log consists of the integers that uniquely identify the messages. Figure 12.27 provides a small sampling of the data.

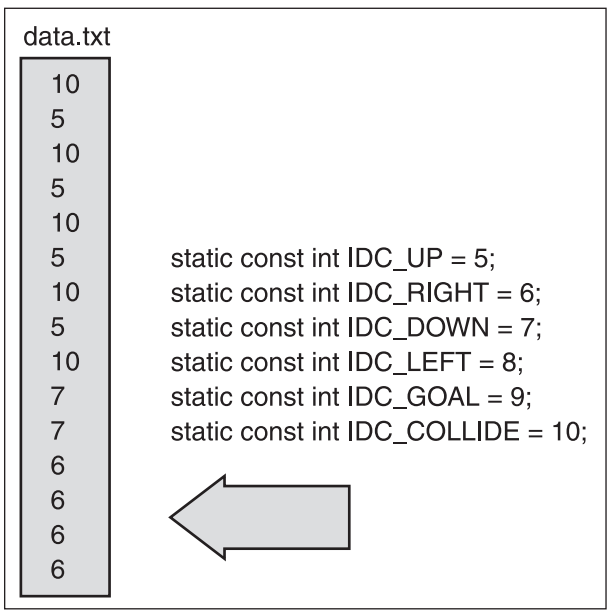


Figure 12.27 The program writes message identifiers sequentially.

Processing Messages

To process messages, you add a set of selection statements to the `OnGUIEvent()` function. The selection statements filter messages and process them accordingly. For most of the section statements, Chapter 9, "Environments of Simulation," discusses the actions you perform to implement message processing. In this context, however, you implement a few statements in special ways. For example, consider how you process messages relating to the end of the game. Here is a snippet of code from the `OnGUIEvent()` function that accomplishes this work:

```
//. . . Lines left out
      else if(GetWorld()->IsCollision(L"Gold", vecNewPos))
      {
            // Log the message for a collision
            if(m_fpOutput)
            {
                  fprintf(m_fpOutput, "%d\n", IDC_GOAL);\
                  fclose(m_fpOutput);
            }
            //../../Bin
            // Run Inspect.exe
            ShellExecute(NULL, L"open", L".\\Inspect.exe",
                              NULL, NULL, SW_HIDE);
            // Open up the inspect application
              MessageBox(NULL, L"You found the gold! \
                                Press OK when finished viewing \
                                data.", L"Winner!", MB_OK);
            pEntity->SetPos(D3DXVECTOR3(0, 0, 0));
```

```
                    CreateStones();
                    // Re-open the file
                    if(m_fpOutput)
                          fclose(m_fpOutput);
                          m_fpOutput = fopen("data2.txt", "wt");
             }
             else
             {
                    pEntity->SetPos(vecNewPos);
             }
      }
      m_pGraphics->SetFog(D3DCOLOR_XRGB(0,0,0), m_fFogStart, m_fFogEnd);
}
```

Unlike other messages, the message that signals the end of the game has a special standing. It occurs only once, and when it occurs, you face a couple of tasks. First, you must announce that the game has ended. Next, given that you want to evaluate the play session using data you have extracted from it, you want to open *Inspect* so that you can process and view the data.

To log the final event of the game, you call the `fprintf()` function one final time to write the `IDC_GOAL` message to the file. In most cases, the game logs this message last, but in some cases, if you do not clear the file, you might see it displayed several times. (The numerical value of the message is `9`.) After writing the `GOAL` message, you call the C library `fclose()` function. The sole parameter for this function is the file stream (`m_fpOutput`).

Calling Inspect

To open *Inspect*, you call a special function that the Win API library includes. This is the `ShellExecute()` function. This function allows you to open one application from another. It provides a handy, safe, and simple approach to opening external executables.

The `ShellExecute()` function requires six parameters. Table 12.3 provides a breakdown of the parameters. Here is the prototype for the function:

```
HINSTANCE ShellExecute( HWND hwnd,
                        LPCTSTR lpOperation,
                        LPCTSTR lpFile,
                        LPCTSTR lpParameters,
                        LPCTSTR lpDirectory,
                        INT nShowCmd
);
```

Table 12.3 ShellExecute Parameters

Parameter	Discussion
`hwnd`	Designates the parent window. In this case, *Inspect* has no parent window, so you assign `NULL`.
`lpOperation`	You assign any of a number of string values to this parameter. Among these are `open`, `find`, `print`, and `explore`. Each of these strings indicates a task. In this instance, you open *Inspect*.
`lpFile`	Designates the file or executable you want to open. In this case, the file is the executable for *Gold Finder*.
`lpParameters`	Designates any values you want to pass to the executable.
`lpDirectory`	Designates the path to the executable.
`nShowCmd`	You can control how the application you open first displays. Among the options are `SW_HIDE`, `SW_MAXIMIZE`, `SW_MINIMIZE`, and `SW_RESTORE`. The initial value is set to `SW_MAXIMIZE`. You might want to change this to `SW_HIDE`.

Inspect is a C# application that you can compile and modify as you wish. Describing how to work with C# and *Inspect* on a programming level lies beyond the scope of this book. However, it might be beneficial to point out that you can find the project for *Inspect* in the SimulationSrc/*Inspect* folder. The project file is named Inspect.sln. If you execute the project file, you can find the algorithms for the calculations in NodeList.cs. Although the code is in C#, you can readily work with it if you know C or C++.

The Message Box

After using the ShellExectute() function to call *Inspect*, you call the MessageBox() function to display a message box. The message box serves as a transition from one application to another. If you intend to engage in frequent testing, you can comment out the message box code. *Inspect* grabs the data it requires when it opens, so you do not have to worry about loss of data.

Following your call to the `MessageBox()` function, your last task involves cleaning up. To clean up, you call the C library `fclose()` function. This function requires only one parameter, which identifies the file stream you have opened (`m_fpOutput`). This closes the file but does not destroy the data in it. Following the call to the `fclose()` function, you include a call to the `fopen()` function. This call to `fopen()` serves to clean up after one session of play and prepare for the next. At this point, the data file is blank.

Conclusion

In this chapter, you have examined a few of the topics that might be associated with testing simulations. This chapter has emphasized a model for simulation that includes event nodes and transitions. Each event node consists of a collection of decisions that establish relationships with other nodes, so you can consider any given node to be an event context. Using a little basic math and some systems diagrams, you can formulate approaches to assigning numerical values to nodes. Given this approach to modeling, it becomes evident that different nodes possess different levels of significance within the context of the system.

Just as nodes possess significance relative to their relationships with other nodes, the relationships between nodes gain significance relative to the values of the nodes they connect. You can refer to the relationships between nodes as transitions. While a transition from one node to another constitutes a simple path, paths within the logical system of a game usually consist of a number of sequentially connected transitions.

The collection of all the nodes in a system constitutes the total node significance of a system. If you compare the number of nodes you navigate through during a given scenario of play to the total number of nodes the system provides, you can determine the relative context value of your session of play. On the other hand, if you consider the paths you navigate during a session of play and compare the value of these paths with the value that represents all the paths the system provides, then you can determine the relative path value of a session of play.

Taken together, relative context value and relative path value allow you to determine the cognitive saturation of a session of play. Measuring the level of cognitive saturation provides you with a way to evaluate how much of the functionality of an application you use during a session of play. No strict standards apply to levels of cognitive saturation, but one general observation to begin with involves considering that if during repeated sessions of play you use only a low percentage of the functionality the system provides, then it might be worthwhile to alter the way you have laid out your game or simulation. You might want to add a few features that induce the user of the game or simulation to more comprehensively explore the features of the game.

This chapter includes a discussion of *Inspect*, an application that processes data generated from *Gold Finder*, the sample game developed in Chapter 9. Using *Inspect* to process data based on the messages generated during a session of play, you can see how levels of cognitive saturation change from session to session. Given a number of sessions of interaction, you can begin to assess whether an application adequately engages its users.

The following books address some of the topics discussed in this chapter:

Virginia Anderson and Lauren Johnson. *Systems Thinking Basics: From Concepts to Causal Loops* (Waltham, Massachusetts: Pegasus Communications, Inc., 1997).

Mat Buckland. *AI Techniques for Game Programming.* (Indianapolis, IN: Premier Press, 2002).

Alistair Cockburn. *Agile Software Development* (Boston: Addison-Wesley, 2002).

Jamshid Gharajedaghi. *Systems Thinking: Managing Chaos and Complexity* (Boston: Butterworth Heinemann, 1999).

Neal Hallford with Jana Hallford. *Swords and Circuitry: A Designer's Guide to Computer Role-Playing Games* (Indianapolis, IN: Premier Press, 2001).

Suguru Ishizaki. *Improvisational Design: Continuous, Responsive Digital Communication.* (Cambridge, Massachusetts: MIT Press, 2003).

Sun-Joo Shin. *The Iconic Logic of Peirce's Graphs* (Cambridge, Massachusetts: MIT Press, 2002).