Chapter 3

New Automated Usability Evaluation Methods

3.1 Introduction

As discussed in Chapter 2, automated usability evaluation (AUE) methods are promising complements to non-automated methods, such as heuristic evaluation and usability testing. AUE methods enable an evaluator to identify potential usability problems quickly and inexpensively compared to non-automated methods and can decrease the overall cost of the evaluation phase [John and Kieras 1996; Nielsen 1993]. Despite the potential benefits of AUE methods, this field is greatly underexplored as shown by the survey in Chapter 2.

Performance evaluation (PE) encompasses established methodologies for measuring the performance (e.g., speed, throughput, and response time) of a system and for understanding the cause of measured performance [Jain 1991]. Since computer systems were first invented in the 1950's, system designers and analysts have used these methodologies extensively to compare and improve the performance of hardware and software systems.

The intent of this chapter is to illustrate how PE provides insight about new methods for automated usability assessment. As such, the chapter systematically compares the two methodologies. It provides background for PE, discusses the mapping between the two methodologies, and introduces two example applications. It then describes how to apply PE to UE for three different classes of evaluation: measurement, simulation, and analytical modeling. In each case, the discussion illustrates the potential for new automated usability evaluation methods based on this comparison.

3.2 The Performance Evaluation Process

System designers, analysts, and high performance computing experts have used performance evaluation techniques extensively to improve and compare the performance of hardware and software systems. More recently, human performance and process engineers began to use these methodologies to understand and improve the performance of humans and work practices. Performance in these contexts is a measure of the speed at which a system (e.g., a computer, person, or process) operates and/or its total effectiveness, including throughput, response time, availability, and reliability. Performance evaluation encompasses methodologies for measuring and for understanding the cause of measured performance. Although this methodology has been utilized in many domains, this chapter focuses on its application in the computer hardware and software domain.

1. Specify performance evaluation goals.
2. Define system boundary.
3. List system services and outcomes.
4. Select performance metrics.
5. List system and workload parameters.
6. Select factors and their levels.
7. Select evaluation method(s).
8. Select workload.
9. Implement evaluation program.
10. Design experiments.
11. Capture performance data.
12. Analyze and interpret performance data.
13. Critique system to suggest improvements.
14. Iterate the process if necessary.
15. Present results.

Figure 3.1: Activities that may occur during the performance evaluation process.

Performance evaluation is a process that entails many activities, including determining performance metrics, measuring performance, and analyzing measured performance. Jain [1991] and Law and Kelton [1991] present similar ten-step systematic approaches to performance evaluation; these steps were adapted for this discussion. Figure 3.1 depicts a fifteen-step process for conducting a performance evaluation. This process is similar to the one outlined for usability evaluation in Chapter 2. Jain [1991] and Law and Kelton [1991] provide detailed discussions of the steps comprising the performance evaluation process. Hence, they are not discussed in this chapter.

3.3 Overview of Performance Evaluation Methods

PE consists of three broad classes of evaluation methods: measurement, simulation, and analytical modeling [Jain 1991]. A key feature of all of these approaches is that they enable an analyst to automatically generate quantitative performance data. Such data can: (i) help designers explore design alternatives; (ii) help analysts tune system performance; and (iii) help consumers purchase systems that satisfy their performance requirements. The methods differ along several dimensions, including the system stage at which they are applicable, cost, accuracy, time, and resource requirements. Table 3.1 compares the methods along these dimensions; the table is modeled after the comparison in [Jain 1991]. The columns reflect the order of importance of dimensions.

Measurement has the potential to be the most accurate and credible evaluation method; however, it is only applicable to an existing system – either the target system or one similar to it. Simulation is usually more accurate and credible than analytical modeling, since it allows

Method	Stage	Time	Resources
Analytical Modeling	Any	Small	Analysts
Simulation	Any	Medium	Computer Languages
Measurement	Post-prototype	Varies	Instruments

Method	Accuracy	Trade-off	Cost	Credibility
Analytical Modeling	Low	Easy	Small	Low
Simulation	Moderate	Moderate	Medium	Medium
Measurement	Varies	Difficult	High	High

Table 3.1: Summary of the performance evaluation method classes.

the analyst to incorporate greater system detail. However, simulations require considerably more time to develop than analytical models. Analytical modeling is usually the least accurate and consequently the least credible evaluation method because of simplifications and assumptions made to produce the model. However, this method can be valuable for comparing alternatives during early system design.

Due to the various trade-offs associated with each evaluation method, it is advisable to employ two or three methods simultaneously [Jain 1991; Law and Kelton 1991; Sauer and Chandy 1981]. Typically, an analyst may use simulation and analytical modeling together to verify and validate the results of each one. It is especially advisable to use analytical modeling and/or simulation along with measurement, since measurement is highly susceptible to experimental errors.

The remaining sections discuss analytical modeling, simulation, and measurement methods in more detail.

3.4 Measurement Methods

Capturing real system performance is by far the most credible, yet most expensive performance evaluation method. It requires careful selection of performance metrics, a means of running a workload, and a means of capturing performance measures for the workload. The following sections discuss these aspects.

3.4.1 Measurement Methods: Selecting Performance Metrics

Performance metrics are a crucial component of this type of assessment and care must be taken when selecting measures. The goal is to choose a minimal number of metrics that reveal the maximum amount of relevant performance detail for the system under study. In instances where multiple users share the system under evaluation (e.g., distributed computing), the analyst must give consideration to both individual and global metrics. Individual metrics reflect performance for each user, while global metrics reflect the system-wide performance.

Jain [1991] suggests using the list of possible service outcomes (correct response[1], incorrect response, or nonresponse) to guide the selection of performance metrics. Jain distinguishes three categories of metrics – speed, reliability, and availability – corresponding to the three possible outcomes. Jain also distinguishes between metrics that measure individual (single user) and global

[1]Response is a generic term applicable to a wide range of services (e.g., query processing, I/O processing, CPU processing, etc.).

Outcome	Metric Type	Example Metric	Scope	Performance Goal
Correct Response	Speed	response time	IG	LB
		throughput	IG	HB
		utilization	G	NB
Incorrect Response	Reliability	probability of error	G	LB
		time between errors	G	LB
Nonresponse	Availability	probability of failure	G	LB
		time between failures	G	LB

Table 3.2: Metrics associated with possible outcomes. Reliability and availability metrics are typically captured over a time period (e.g., weeks or months) and are used to assess system performance across users versus for individual users. The scope of a metric is an individual user (I), global or across users (G), or both. Performance goals include: lower is better (LB), higher is better (HB), and nominal or middle is better (NB).

(across users) performance. Table 3.2 summarizes these distinctions along with example metrics. The scope of each metric – individual or global – is denoted with an I and G, respectively. The following performance goals are also appropriately associated with each example metric: LB - lower is better; HB - higher is better; and NB - nominal or middle is better. Below is a discussion of each of the metric types.

- **Speed:** a measure of system response time, throughput, or utilization. Response time is the time between a user's request and the system's response to the request. Throughput is the rate (in requests per unit of time) at which the system services requests. Utilization is the fraction of time the system is busy servicing requests.
- **Reliability:** a measure of the fraction of time the system correctly services users' requests.
- **Availability:** a measure of the fraction of time the system is available to service user requests.

It is possible that an analyst may need to employ several metrics, individual and global, for each system service; hence, the number of metrics can grow proportionally (e.g., twice the number of services to be evaluated). To reduce the number of metrics, Jain suggests selecting a subset of metrics with low variability, nonredundancy, and completeness as discussed below.

- **Low Variability:** metrics that are not a ratio of two or more variables.
- **Nonredundancy:** metrics that do not convey essentially the same information.
- **Completeness:** metrics that reflect all of the possible outcomes of a service.

3.4.2 Measurement Methods: Running a Workload

In order to capture performance data, the analyst must first load a workload onto the system via a load driver. Internal drivers, live operators, and remote terminal emulators are the major types of load drivers and are discussed below.

- **Internal Driver:** encapsulates mechanisms for loading and running the workload into one program, as is the case in benchmarks. One problem with this approach is that loading the workload may affect system performance during execution. Nonetheless, benchmarks are a commonly used internal load driver. They are discussed in more detail below.

- **Live Operators:** real users submit requests to the system. The use of live operators is extremely costly and hard to control.

- **Remote Terminal Emulators:** programs that run on separate computers and submit requests to the system under study. This is one of the most popular and desirable load drivers used; it is not nearly as costly as using live operators, and it eliminates the interference problem of internal drivers.

Measurement Methods: Running a Workload – Benchmarking

Benchmarking [Dowd 1993; Jain 1991; Weicker 1990], is one of the most commonly used measurement techniques for comparing two or more systems. Benchmarking entails using a high-level, portable program with a realistic workload and adequate problem size to automatically quantify performance aspects in a reproducible manner. A single such program is referred to as a benchmark, while multiple programs used jointly are referred to as a benchmark suite. Benchmarks automatically specify performance metrics and workloads. As such, they simplify the performance evaluation process to some degree.

Benchmarks differ along several dimensions, including the application domain, the code type used, and the type of execution measured as summarized below.

- **Application Domain:** the benchmark workload and problem size typifies a specific application domain (e.g., scientific computing, graphics, or databases) and measures performance aspects germane to this domain. For example, a scientific computing benchmark may assess floating-point computation speed, whereas a database benchmark may assess transaction completion speed or throughput.

- **Code Type:** the actual code executed and measured can be a real application (i.e., it exercises system resources as fully and realistically as possible), a kernel (i.e., the computation "core" of an application), or a synthetic program designed to model activity of a typical program or to mimic a real workload.

- **Execution Type:** the benchmark code can measure several execution types - single stream, throughput, or interactive. Single stream benchmarking measures the time to execute one or more benchmarks individually. Throughput benchmarking collects measurements while multiple programs run concurrently. Use of the term throughput here is distinct from its use as a performance measure by both single stream and interactive benchmarks. In these instances multiple programs are not running concurrently; hence, they are not considered throughput benchmarks. Throughput benchmarks are not as common nowadays as they were with batch processing systems. Interactive benchmarking measures response time or throughput in a client/server environment; this execution type requires a second program to simulate user requests to the server.

Several industry benchmarks provide objective measures of system performance. Standard benchmarks help buyers to make informed purchases and vendors to prioritize optimization in their systems. One drawback of these benchmarks is that they lack orthogonality because they sometimes measure many things at once; this makes it difficult to draw concrete conclusions about performance. Table 3.3 summarizes several commonly used industry benchmarks. There are published performance results from a myriad of systems for each of these benchmarks; thus, enabling comparison across systems.

Benchmark	Application Domain	Code Type	Execution Type
Linpack	scientific computing	application	single stream
STREAM	scientific computing	synthetic	single stream
TPC-C	database	synthetic	interactive
TPC-D	database	synthetic	interactive
SPEC Web 96	Web	synthetic	interactive

Benchmark	What's Measured	How It's Measured	Metric
Linpack	floating point speed	solving linear equation	MFlops
STREAM	memory bandwidth	performing vector operations	MB/s
TPC-C	server throughput	servicing complex transactions	tpmC
TPC-D	server throughput	servicing complex db queries	QphD
SPEC Web 96	server throughput	servicing HTTP GET requests	SPECweb96

Table 3.3: Summary of commonly used industry benchmarks. The What's Measured column reflects the performance aspect measured by each benchmark. The How It's Measured column describes the workload used for measuring this performance aspect, while the Metric column lists the measurement returned by each benchmark.

3.4.3 Measurement Methods: Capturing Performance Metrics

Analysts use several types of monitors (e.g., hardware counters and software timers) as well as accounting logs to capture performance data as discussed below.

- **Monitors:** measure system performance at some level. Hardware monitors, such as counters, measure low-level performance aspects (e.g., signals on buses and instruction execution time). Software monitors, such as code instrumented with timing calls, measure high-level performance aspects (e.g., queue lengths and time to execute a block of code). There are also firmware monitors, such as a processor microcode instrumented with timing calls, that measure performance aspects of network interface cards and other external system components. Sometimes analysts use multiple hardware, software and/or firmware monitors simultaneously as a hybrid monitor.

- **Accounting Logs:** another form of software monitors that automatically capture performance data during program execution. Hence, they do not require system instrumentation as is the case for monitors. Usually, compiling a program with certain flags enables accounting.

Tracing and sampling are the primary measurement techniques used in monitors and accounting logs. Tracing actually produces a time-stamped record of requests as they move through various stages of the system and/or of event occurrences in the system. Sampling or timing entails reading a clock at specified times to compute elapsed time between timing events.

3.5 Analytical Modeling Methods

An analytical model consists of mathematical or logical relationships that represent the analyst's assumptions about how a system works. By solving this model, the analyst can predict system performance. Analytical modeling approaches range in complexity from simple "back of the envelope" calculations to formal queuing theory as discussed below.

Simplistic Models. "Back of the envelope" and "front of the terminal" [Sauer and Chandy 1981] calculations are two relatively simple analytical modeling approaches. In the "back of the envelope" approach the system model is extremely crude such that it facilitates solving the model via unaided calculation. An analyst may solve slightly more complicated models using modeling software or a simple mathematical environment, such as Matlab; this is considered to be "front of the terminal" calculation. Since simplistic models are very abstract representations of systems, the accuracy of the results produced are highly questionable. Nonetheless, this approach is very appropriate during the design stage of a system.

Informal Queuing Theory. In this approach an analyst develops a queuing model (i.e., a model of the times a request spends in various system resource queues) that captures all of the significant aspects of the system. The analyst then uses this model to get an approximation (e.g., using numerical methods) to the model solution (i.e., response time). The model is usually significantly more detailed than the previous approach but less thorough and accurate than formal queuing theory. The latter case is due to the use of approximate versus exact system parameters.

Formal Queuing Theory. Formal queuing theory requires more accurate specification of the following six system parameters: interarrival times of requests; service time at each queue; number of resources or servers; system capacity (i.e., number of requests that can be serviced); population size or the maximum number of requests; and the service discipline or policy for servicing requests, such as first come first served. This approach can also be used to model multiple queues in a system as a queuing network. The objective of formal queuing theory is to solve models for parameters that impact performance to answer questions, such as the number of servers required to fulfill the current demand or appropriate sizes of queue buffers to prevent overflow.

3.6 Simulation Methods

Some models may be too complex to solve using analytical modeling or may have no analytical solution. In these cases the analyst can create a simulation (i.e., a computer program) to exercise the model with various inputs to see the resulting affects on output measures of performance. The analyst accomplishes this with a system model, a program or simulator that behaves like the model, and detailed input data to the program. Simulation allows the analyst to create arbitrarily detailed models of systems. Consequently, performance analysts consider simulation to be more credible and accurate than analytical modeling. Nonetheless, simulation requires considerably more time to develop as well as compute resources to run.

One of the most important aspects of a simulation is the underlying simulation model. Simulation models vary along three major dimensions as discussed below.

- **System Evolution:** time-independent models are a representation of a system at a particular time that is totally independent of time, whereas time-dependent models represent a system as it evolves over time.
- **System Parameters:** deterministic models do not contain probabilistic (i.e., generated based on random numbers) input components, while probabilistic models may contain some random input components.
- **Number of System States:** finite models have a countable number of system states, whereas infinite models have an uncountable number of states.

These aspects of the model largely govern the type of simulation the analyst uses during in performance studies. Below is a discussion of several commonly used simulation approaches.

Discrete-Event. In a discrete-event simulation the state variables of the system change instantaneously in response to events.

Continuous-Event. In this simulation approach the state variables of a system change continuously with respect to time.

Combined Discrete-Continuous. A simulation can use both the discrete-event and continuous-event approaches. For example, a discrete system change may occur when a continuous variable reaches a threshold value.

Monte Carlo. A Monte Carlo simulation uses random numbers to solve stochastic or deterministic models without time dependence. Analysts use this simulation approach to model probabilistic phenomenon that do not change characteristics with time, such as cancer cell growth.

Any of these simulation approaches, with the exception of Monte Carlo simulation, can be driven by a time-ordered record of events taken from a real system. This is referred to as trace-driven simulation. Analysts consider this type of simulation to be the most credible and accurate.

Several problems associated with all of these simulation approaches include: an inappropriate level of system detail, poor random number generators and seeds, no verification or validation of models, too short simulation runs, and too long simulation runs due to the model complexity.

3.7 Mapping Between Performance and Usability Evaluation

As previously discussed, performance evaluation encompasses established methodologies for measuring the performance (e.g., speed, throughput, and response time) of a system and for understanding the cause of measured performance [Jain 1991]. PE consists of three broad classes of evaluation methods: measurement, analytical modeling, and simulation. A key feature of all of these approaches is that they enable an analyst to automatically generate quantitative performance data.

Usability evaluation, on the other hand, encompasses methodologies for measuring usability aspects (e.g., effectiveness, efficiency, and satisfaction) of an interface and for identifying specific problems [Nielsen 1993]. As discussed in Chapter 2, UE consists of five classes of methods: testing, inspection, inquiry, analytical modeling, and simulation. Although there has been some work to automate these approaches, automated UE methods are greatly underexplored. Furthermore, only 29% of existing methods support the same level of automation as PE methods (i.e., automated capture, analysis, or critique without requiring interface usage).

The remainder of this chapter discusses how PE can be used as a guiding framework for developing new automated UE methods for both WIMP (Windows, Icons, Menus and Pointers) and Web interfaces. Similarly to PE, automated UE methods can provide additional support to evaluators using non-automated methods and for designers exploring design alternatives. These methods are even more crucial in instances where performance is a major consideration.

Sections 2.3 and 3.2 show the UE and PE processes to be quite similar. Furthermore, there is a mapping between the methods used within each domain. Table 3.4 summarizes this mapping. PE measurement, analytical modeling, and simulation methods are used as a framework for this discussion. The following section presents two applications used throughout the comparison of PE and UE.

PE	UE
measurement	testing, inspection, inquiry
analytical modeling	analytical modeling
simulation	simulation

Table 3.4: Mapping between performance evaluation (PE) and usability evaluation (UE) method classes.

Figure 3.2: Address label example in Microsoft Word.

3.7.1 Example Applications and Tasks

Two example applications and representative tasks are referred to throughout this discussion. The first application is a word processor with a task of creating address labels – six to a page where each label has a rectangular border. Figure 3.2 depicts a sample set of address labels in Microsoft Word.

It is assumed that the user loaded the application prior to beginning the task and that the cursor is at the top left margin of a blank document when the user begins the label creation task. Analysis of this task within Microsoft Word 97 revealed that it requires an expert user 55 steps to complete (see Appendix B). A replicated analysis in Microsoft Word 2000 derived the same number of steps. Figure 3.3 shows the first thirteen steps required to create the first label, along with the corresponding high-level goals; Appendix B contains the high-level goals for all 55 steps in the task sequence. A high-level task structure similar to NGOMSL [John and Kieras 1996] is used in discussions.

The second application is an information-centric Web site typical of large-scale federal and state agencies. The task for this application is to navigate from a site entry point to a page that contains some target information. Figure 3.4 depicts this example with a sample navigation path. Unlike the label creation example, there is no clear step-by-step procedure for completing the task. The user could start from any page within the site and follow various paths to the target information. Hence, it is not possible to specify an explicit task structure as specified for the label creation example without restricting users to traversing one navigation path through the site. It is assumed that users are unfamiliar with the site and that locating the information is a one-time

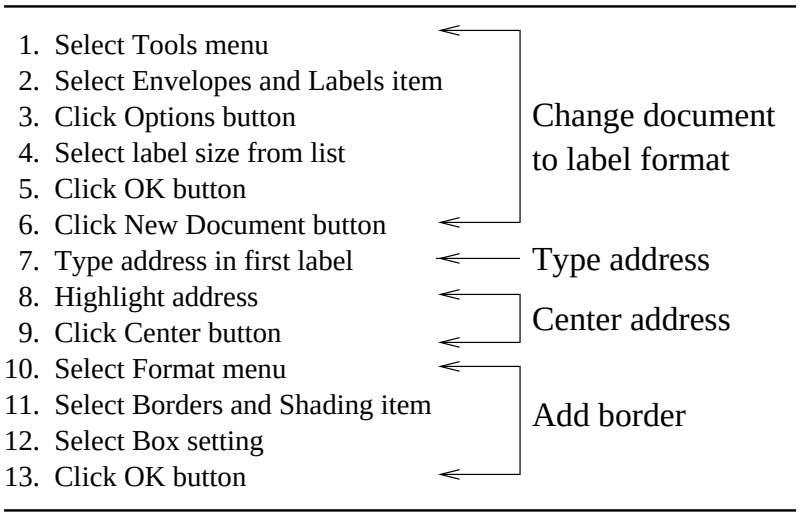

Figure 3.3: Label creation steps in Microsoft Word 97 and 2000.

Measurement Workload	**Measurement Type**	
	Monitoring	Profiling
Benchmark		
Granularity	fine	coarse
Interference	yes	yes
Real User		
Granularity	fine	coarse
Interference	no	no

Table 3.5: Characteristics of measurement techniques in the performance evaluation domain.

task. Thus, bookmarking is not used to access the information. It also assumed that users enter the site via a search engine or external link.

The remainder of this chapter demonstrates how to determine the number of errors and the navigation time for the label creation and site navigation tasks, respectively.

3.8 New UE Measurement Methods

3.8.1 Measurement in Performance Evaluation

Measurement is by far the most credible, yet most expensive PE method [Jain 1991]. It requires a means of running a workload on a system as well as a means of capturing quantitative performance data while the workload runs. A performance analyst usually derives the workload from current or anticipated system use. Capturing quantitative data for the workload enables the analyst to identify performance bottlenecks, tune system performance, and forecast future performance. Table 3.5 summarizes relevant techniques for running workloads and capturing quantitative performance data. Section 3.4 discusses these approaches in more detail.

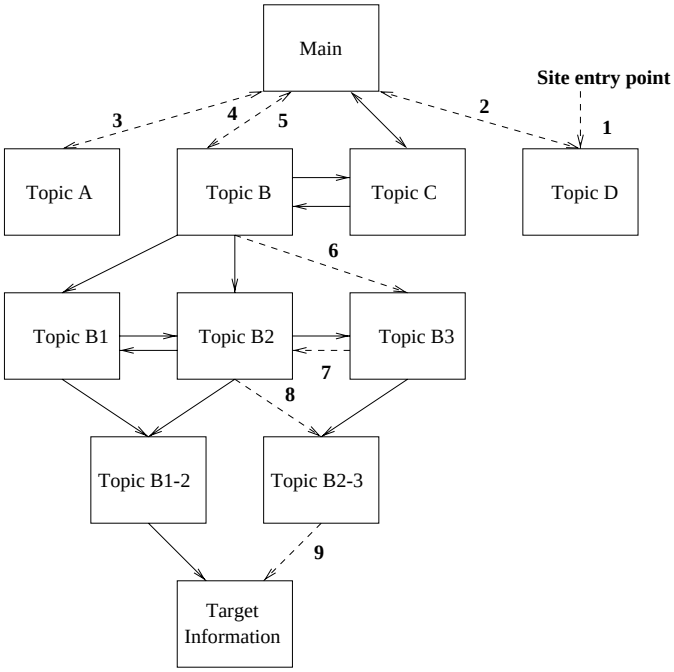

Figure 3.4: Information-centric Web site example.

3.8.2 Measurement in Usability Evaluation

Inspection, testing, and inquiry UE methods are equivalent to PE measurement techniques; most require an interface or prototype to exist to capture measurements of some form (e.g., number of heuristic violations, task completion time, and subjective rating). Most evaluation methods surveyed in Chapter 2 do not produce quantitative data. However, methods that do produce quantitative data, such as performance measurement[2], use both profiling and monitoring techniques (e.g., high-level and low-level logging) similarly to their PE counterparts, and all of these methods require a real user (an evaluator or test participant).

Although monitoring with benchmarks is the predominate approach in the PE domain, it is unused in the UE domain. The closest approximation is replaying previously-captured usage traces in an interface [Neal and Simons 1983; Zettlemoyer *et al.* 1999]. Early work with Playback [Neal and Simons 1983] involved recording actions performed on the keyboard during usability testing and then sending the recorded commands back to the application. The evaluator could then observe and analyze the recorded interaction.

More recent work automatically generates usage traces to drive replay tools for Motif-based UIs [Kasik and George 1996]. The goal of this work is to use a small number of input parameters to inexpensively generate a large number of test scripts that a tester can then use to find weak spots and application failures during the design phase. The authors implemented a prototype system that enables a designer to generate an expert user test script and then insert deviation commands at different points within the script. The system uses a genetic algorithm to choose user behavior during the deviation points as a means for simulating a novice user learning by experimentation.

Recent work in agent technology captures widget-level usage traces and automatically

[2]Performance measurement in this context refers to usability testing methods rather than the performance evaluation method.

analyzes user actions during replay [Zettlemoyer *et al.* 1999]. The IBOT system interacts with Windows operating systems to capture low-level window events (e.g., keyboard and mouse actions) and screen buffer information (i.e., a screen image that can be processed to automatically identify widgets). The system then combines this information into interface abstractions (e.g., menu select and menubar search operations) that it can use to infer UI activities at a high-level. The IBOT system can also perform the same operations as a real user by adding window events to the system event queue. Similar work has been done in the software engineering field for generating test data values from source code [Jasper *et al.* 1994].

Guideline review based on quantitative measures is a somewhat distant approximation of PE benchmarking. Automated analysis tools, such as AIDE (semi-Automated Interface Designer and Evaluator) [Sears 1995], compute quantitative measures for WIMP UIs and compare them to validated thresholds. Similar, underdeveloped approaches exist for Web UI assessment (e.g., HyperAT [Theng and Marsden 1998], Gentler [Thimbleby 1997], and the Rating Game [Stein 1997]). Thresholds either do not exist or have not been empirically validated.

3.8.3 Applying PE Measurement Methods to UE

As previously stated, benchmarking is widely used in PE to automatically capture quantitative performance data on computer systems. Usability benchmarks, especially benchmarks that can be executed within any UI, is a promising open area of research. Nonetheless, there are three major challenges to making this a reality in the UE domain. Brief discussions of these challenges and potential solutions are below; the next section provides more depth on these issues.

The first challenge is generating usage traces without conducting usability testing or reusing traces generated during usability testing. One approach was discussed above [Neal and Simons 1983]; however, more work needs to be done to automatically generate traces that represent a wider range of users and tasks to complement real traces. In particular, a genetic algorithm could simulate usage styles other than a novice user learning-by-experimentation [Kasik and George 1996]. It may also be beneficial to implement a hybrid trace generation scheme wherein traditional random number generation is used to explore the outer limits of a UI [Kasik and George 1996]. Data from real users, such as typical errors and their frequencies, could serve as input in both cases to maximize the realism of generated tasks.

The second challenge is making such traces portable to any UI. In the PE domain this is accomplished by writing hardware-independent programs in a common programming language, such as C or Fortran. Analysts then compile these programs with the appropriate compiler flags and execute them to capture performance data. There needs to be a similar means of creating portable usage traces. One way may entail mapping high-level tasks captured in a trace into specific interface operations. The USINE system [Lecerof and Paternò 1998] provides insight on accomplishing this. In this system, evaluators create task models expressing temporal relationships between steps and create a table specifying mappings between log file entries and the task model. USINE uses this information to identify task sequences in log files that violate temporal relationships. Another approach employed in the IBOT system is to use system-level calls (e.g., mouse and keyboard event messages); this simplifies porting to other operating systems. The authors claim that the IBOT agent can interact with any off-the-shelf application because it is independent of and external to the UI.

A proposed solution would combine both the USINE and IBOT approaches. Similarly to USINE, a task programming tool could prompt evaluators for application steps corresponding to each task within a generated trace. Ideally, the evaluator could use programming-by-demonstration [Myers 1992] to record application steps. The task programming tool could translate application

Challenge	PE	UE
Executable Workload	software programs	usage traces (real & generated)
Portability	hardware-independent software programs	high-level traces with UI mapping, benchmark program generation
Quantitative Metrics	execution time, standard metrics	number of errors, navigation time

Table 3.6: Summary of the challenges in using PE measurement techniques within the UE domain. The PE column describes how these challenges are resolved within the PE domain, and the UE column describes potential ways to resolve these challenges within the UE domain.

sequences into a format, such as system-level calls, that could be subsequently executed within the application. A benchmark generation tool could then translate the usage trace and mapped application sequences into a program for execution similarly to replay tools; each application sequence could be expressed as a separate function in the program. The Java programming language is promising for the task programming tool, the benchmark generation tool, as well as the generated benchmark programs.

The final challenge is finding a good set of quantitative metrics to capture while executing a trace. Task completion time, the number and types of errors, and other typical UI metrics may suffice for this. One of the drawbacks of relying solely on quantitative metrics is that they do not capture subjective information, such as user preferences given a choice of UIs with comparable performance and features.

Usability benchmarks are appropriate for evaluating existing UIs or working prototypes (as opposed to designs). Evaluators can use benchmark results to facilitate identifying potential usability problems in two ways: (i) To compare the results of an expert user trace to results from those generated by a trace generation program. This may illustrate potential design improvements to mitigate performance bottlenecks, decrease the occurrence of errors, and reduce task completion time. (ii) To compare results to those reported for comparable UIs or alternative designs. This is useful for competitive analysis and for studying design tradeoffs. Both of these uses are consistent with benchmark analysis in the PE domain.

Table 3.6 summarizes the challenges with using benchmarking in the UE area. The next section describes usability benchmarks for the example tasks in more detail.

3.8.4 Example Usability Benchmark

As previously discussed, executable and portable usage traces and a set of quantitative performance metrics are required to construct a benchmark. To generate high-level usage traces, the evaluator could specify a high-level representation of the label creation task sequence previously discussed. Figure 3.5 depicts the nine high-level steps that correspond to the 55-step Microsoft Word sequence. It is also possible to process a real user trace to create a high-level task sequence.

Figure 3.6 demonstrates a procedure for using the high-level task sequence as input for constructing and executing a usability benchmark. The evaluator could use the high-level trace as a task template in which deviation points could be identified similarly to the work done in [Kasik and George 1996]. The trace generation program would then generate plausible variations of the task sequence that represent alternative user behaviors during task completion. Figure 3.7 shows the type of output that the program might generate – an example in which a user mistakenly enters text before changing the document format, corrects the mistake, and completes the task as specified by the task template. This approach enables the evaluator to amplify the results taken

1. Change document to label format
2. Enter text
3. Center text
4. Add square border to text
5. Copy text
6. Move cursor to next label
7. Paste text
8. Add square border to text
9. Repeat steps 6-8 for remaining 4 labels

Figure 3.5: High-level steps for the label creation task.

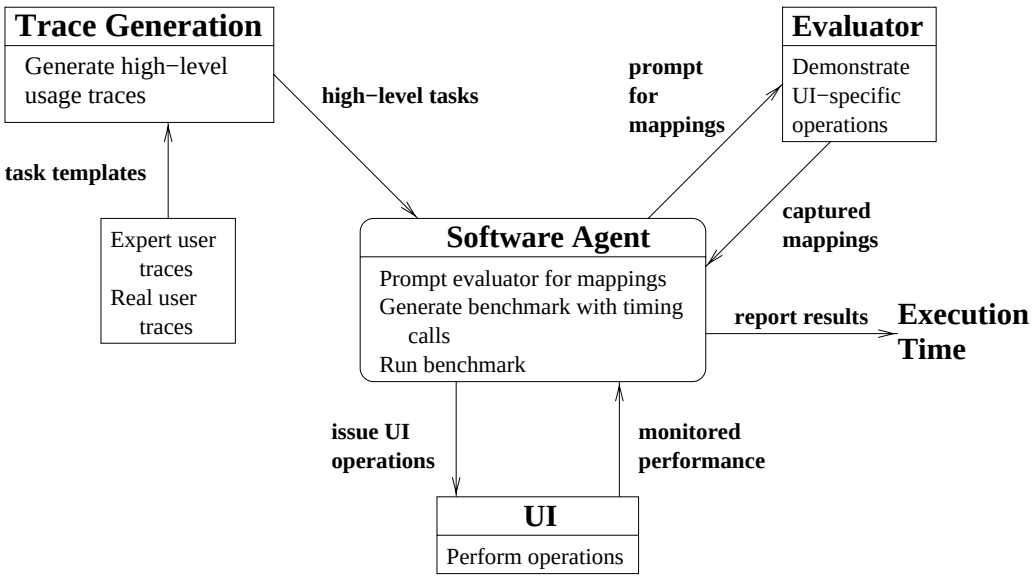

Figure 3.6: The proposed usability benchmarking procedure.

from a small number of inputs or test subjects to generate behavior equivalent to a larger number of users and wider UI coverage.

An evaluator or designer could then map all of the high-level tasks in the generated traces into equivalent UI-specific operations as depicted in Figure 3.6. Figure 3.3 shows one such mapping for creating the first label in Microsoft Word. Programming-by-demonstration [Myers 1992] is one way to facilitate this mapping. An agent program could prompt the designer to demonstrate a task sequence, such as delete document, and record the steps to facilitate playback. If there is more than one way to accomplish a task, then the designer could demonstrate multiple methods for a task and specify a frequency for each, similarly to GOMS [John and Kieras 1996] task representations.

Given the high-level usage traces and UI mappings, the agent could then automatically generate executable benchmark programs to replay in a UI as depicted in Figure 3.6. To facilitate error analysis, the agent could also generate a designer benchmark (i.e., a benchmark representing the correct sequences of operations to complete a task) from the task template and UI mappings. The agent would then "run" (i.e., send operations to the UI) this designer benchmark once to record

1. Enter text
2. Change document to label format
3. Delete document
4. Create new document
5. Change document to label format
6. Enter text

⋮

Figure 3.7: Example trace for the label creation task.

1. Select Topic D page (Site entry point)
2. Select Main link
3. Select Topic A link
4. Click Back button
5. Select Topic B link
6. Select Topic B3 link
7. Select Topic B2 link
8. Select Topic B2-3 link
9. Select Target Information link

Figure 3.8: Example trace for the information-seeking task.

system state after each operation. The agent would repeat this procedure for each generated benchmark, record discrepancies (i.e., differences in resulting system state between the designer benchmark and other benchmarks) as errors, note error locations, and report whether the task completed successfully[3]. The agent could also aggregate data over multiple traces to provide more conclusive error analysis data.

Similar approaches using generated benchmark programs as well as quantitative measures could be applied to the information-centric Web site example. Generating traces for multiple navigation paths (and in some cases all possible paths) is the most crucial component for this example. An algorithm can determine plausible paths based on the navigation structure and content of the links and pages. Chi *et al.* [2000] also demonstrates a methodology for generating plausible navigation paths based on information scent. Genetic algorithm modeling could also be employed for this. Again, as input the evaluator could use real user traces from Web server logs or a designer-generated trace. Since navigation operations (e.g., select a link, click the back or forward button, etc.) are standard in Web browsers, it may be possible to eliminate the mapping operation that was required for the WIMP example. Hence, the genetic algorithm could generate executable benchmarks directly as depicted in Figure 3.8. Figure 3.4 shows the corresponding navigation path.

A software agent could simulate navigation, reading, form completion, and other user behavior within the actual site and report navigation timing data. WebCriteria's Site Profile tool [Web Criteria 1999] uses a similar approach to simulate a user's information-seeking behavior within a model of an implemented Web site. Site Profile uses a standard Web user model to follow an

[3] Actually "running" both benchmarks in a UI is required, since it may not be possible to ascertain successful task completion by comparing two task sequences. This can only be accomplished by comparing system state.

explicit navigation path through the site and computes an accessibility metric based on predictions of load time and other performance measures. This approach suffers from two major limitations: it uses a single user model; and it requires specification of an explicit navigation path. The proposed approach with automatically-generated navigation paths would not have these limitations.

Another benchmarking approach for Web sites could entail computing quantitative measures for Web pages and sites and comparing these measures to validated thresholds or profiles of highly-rated sites. As previously discussed, HyperAt, Gentler, and the Rating Game compute quantitative measures (e.g., number of links, number of words, and breadth and depth of each page), but validated thresholds have not been established. This dissertation presents an empirical framework for using quantitative measures to develop profiles of highly-rated sites. Such profiles could also be used to determine validated thresholds. Subsequent chapters discuss the methodology, profiles, and threshold derivations in more detail.

3.9 New UE Analytical Modeling Methods

3.9.1 Analytical Modeling in Performance Evaluation

Analytical modeling entails building mathematical or logical relationships to describe how an existing or proposed system works. The analyst solves a model to predict system performance. Such predictions are useful for studying design alternatives and for tuning system performance in the same manner as measurement and simulation. The major difference is that analytical modeling is much cheaper and faster to use albeit not as credible [Jain 1991].

Most analytical models do not adhere to a formal framework, such as queuing theory (see Section 3.5). These models use parameterized workloads (e.g., request characteristics determined by probability distributions) and vary in complexity. Some models may be solved with simple calculations, while others may require the use of software.

3.9.2 Analytical Modeling in Usability Evaluation

The survey in Chapter 2 revealed analytical modeling methods for WIMP UIs, but not for Web interfaces. GOMS analysis [John and Kieras 1996] is one of the most widely-used analytical modeling approaches, but there are two major drawbacks of GOMS and other UE analytical modeling approaches: (i) They employ a single user model, typically an expert user, and (ii) They require clearly-defined tasks. The latter is appropriate for WIMP UIs but does not work well for information-centric Web sites for reasons previously discussed.

One way to address the first problem is to construct tasks representative of non-expert users with a programming-by-demonstration facility embedded within a UIDE. CRITIQUE [Hudson *et al.* 1999] is one such tool; it automatically generates a GOMS structure for a task demonstrated within a UIDE. Constructing tasks representative of non-expert users requires the evaluator or designer to anticipate actions of novice and other types of users. However, this information is usually only discovered during usability testing. Thus, it has a strong non-automated component.

3.9.3 Applying PE Analytical Modeling to UE

Another approach to address both of the problems discussed above can be derived from the PE analytical modeling framework. Recall from Section 3.3 that PE analytical models predict system performance based on input parameters. To study different usage scenarios, the analyst simply changes the input parameters, not the underlying system model. Analytical modeling in the

Challenge	PE	UE
Modeling System	parameterized model	parameterized model
Multiple Usage Scenarios	vary input parameters	usage traces (real & generated), vary input parameters

Table 3.7: Summary of the challenges in using PE analytical modeling techniques in the UE domain. The PE column describes how these challenges are resolved within the PE domain, and the UE column describes potential ways to resolve these challenges within the UE domain.

UE domain is usually performed in a manner contrary to the PE counterpart (especially techniques using GOMS). These approaches require the evaluator to change the underlying system model rather than the input parameters to study different usage scenarios. Below is a brief discussion of challenges and potential solutions for addressing this problem; the next section presents a more in depth solution.

It would be better to construct an abstracted model (i.e., no task details) of a UI that encapsulates the most important system parameters for predicting performance (e.g., baseline time to enter information in a dialog box or scan a Web page for novice and expert users). For example, the designer could construct a basic graphical UI model to predict the number of errors for a high-level task sequence based on the number of operations in a task, the probability of errors, and other relevant interface parameters. The designer could then vary the input parameters (e.g., the number of dialog boxes required by a task and that predictions are to be based on a novice user) to the model. Each variation of input parameters corresponds to a different design or different user model. This allows for quick, coarse comparison of alternative designs. Previous work on GOMS and usability studies could be used to construct this basic UI model. Such models inherently support ill-defined task sequences, since they only require specification of key parameters for tasks. Although predictions from the model would be crude, such predictions have proven to be invaluable for making design decisions in PE [Jain 1991].

Besides constructing an abstract model, the designer must determine appropriate system and input parameters. If an interface exists, either as an implemented system or a model, then the designer could process generated or real usage traces to abstract the required input parameters. If an interface or interface model does not exist, then the designer must specify required input parameters manually.

Cognitive Task Analysis (CTA) [May and Barnard 1994] employs a modeling approach that is similar to the one proposed. CTA requires the evaluator to input an interface description to an underlying theoretical model for analysis. The theoretical model, an expert system based on Interacting Cognitive Subsystems (ICS [Barnard 1987]; discussed in Section 2.9), generates predictions about performance and usability problems similarly to a cognitive walkthrough. The CTA system prompts the evaluator for interface details from which it generates predictions and a report detailing the theoretical basis of predictions. Users have reported experiencing difficulties with developing interface descriptions. An approach based on quantitative input parameters should simplify this process.

Table 3.7 summarizes the challenges for using analytical modeling in the UE domain as it is used in PE. Analytical modeling is most appropriate for performing quick, coarse evaluations of various design alternatives.

3.9.4 Example Analytical Models

Analytical modeling is appropriate for comparing high-level designs in order to inform design decisions. Below are example comparison scenarios for the label creation and Web site navigation tasks.

Label Creation: A designer wants to develop a wizard for walking users through the cumbersome label creation process. The designer is considering one wizard design that has only three steps (dialog boxes) but requires the user to specify multiple things (e.g., various label and page settings) during each step. The designer is also considering a design with five steps wherein the user specifies fewer things during each step. The designer wants to know which of the designs would be easier to use.

Web Site Navigation: A designer needs to determine how to divide some content over multiple pages. The designer is considering dividing the content over five large pages as well as over eight medium pages. The designer wants to know which of the designs would be easier to navigate, especially for first-time visitors.

Equation 3.1 demonstrates a way to predict task completion time; this calculation could be embedded within a simple UI model and used to generate predictions for the label creation wizard. This model could contain baseline or average times for completing various interface operations (*task_type*), such as entering information in a dialog box or performing a generic task. The model could also incorporate high-level task complexity (*complexity_adj*) and user (*user_type*) models. For example, if the designer specified prediction based on a novice user, then the model could adjust the baseline time for tasks (e.g., increase by 15%). Similarly, if the designer specified that tasks were highly complex, then the model could further adjust the baseline time. To generate predictions, the designer need only specify the task type (dialog task), the task complexity (medium for the first design and low for the second), and the user type (novice). The designer could vary input parameters to generate predictions for other scenarios. The UI model could also include an equation similar to Equation 3.1 for predicting the number of errors.

$$\begin{aligned} T &= num\_tasks * (task\_time[task\_type] * \\ &\quad complexity\_adj[complexity\_type] * user\_adj[user\_type]) \end{aligned} \tag{3.1}$$

Equation 3.2 demonstrates a similar way to predict navigation time; this calculation could be embedded within a simple Web navigation model and used to generate predictions for the content organization approaches. This model is based on the taxonomy of Web tasks discussed in [Byrne *et al.* 1999]. Similarly to the previous example, the designer could specify input parameters for the number of pages (five in the first design and eight in the second), the complexity of pages (high in the first design and medium in the second), and the user type (novice). The UI model could also include an equation similar to Equation 3.2 for predicting the number of errors.

$$\begin{aligned} T &= num\_pages * ((navigate\_time * complexity\_adj[complexity\_type] * \\ &\quad user\_adj[user\_type]) + (read\_time * complexity\_adj[complexity\_type] * \\ &\quad user\_adj[user\_type]) + (think\_time * complexity\_adj[complexity\_type] * \\ &\quad user\_adj[user\_type])) \end{aligned} \tag{3.2}$$

Assuming models of graphical interface usage and Web site navigation existed, designers could quickly, albeit possibly not very accurately, compare designs and use predictions to inform decisions. Using empirical data to develop system parameters should improve the accuracy of such models.

3.10 New UE Simulation Methods

3.10.1 Simulation in Performance Evaluation

In simulation, the evaluator constructs a detailed model of a system to reproduce its behavior [Jain 1991]. Typically, a computer program (known as a simulator) exercises this underlying model with various input parameters and reports resulting system performance. Unlike an analytical model, which represents a high-level abstraction of system behavior, a simulator mimics system behavior. Hence, it is possible to use actual execution traces to drive a simulator, which is not possible with analytical models (see below). Consequently, analysts regard simulation results as more credible and accurate than analytical modeling results [Jain 1991]. Simulators also allow for the study of alternative designs before actually implementing the system. This is not possible with measurement techniques because the system must exist to capture performance data.

The underlying simulation model is one of the major differences among the various simulation approaches. The defining characteristics of these models are: the way the system evolves (time dependent or time independent), how its parameters are generated, and the number of states it can evolve into. In time-independent evolution, the system does not change characteristics based on time (i.e., time is not a system parameter); the opposite is true of time-dependent evolution. System parameters can be *fixed* (i.e., set to specific values) or *probabilistic* (i.e., randomly generated from probability distributions). Finally, simulation models can have a *finite* or countable number of system states or an *infinite* or uncountable number.

Another distinguishing feature of a simulator is its workload format. The workload may be in the form of fixed or probabilistic parameters that dictate the occurrence of various system events or an execution trace captured on a real system. Performance analysts consider trace-driven simulations to be the most credible and accurate [Jain 1991].

Table 3.8 summarizes characteristics for two frequently-used simulation approaches, discrete-event and Monte Carlo simulation. Discrete-event simulations model a system as it evolves over time by changing system state variables instantaneously in response to events. Analysts use this approach to simulate many aspects of computer systems, such as the processing subsystem, operating system, and various resource scheduling algorithms. Execution traces are often used in discrete-event simulations, since it is relatively easy to log system events.

Monte Carlo simulations model probabilistic phenomena that do not change characteristics with time. Analysts use this approach to conduct what-if analysis (i.e., predict resulting performance due to system resource changes) and to tune system parameters. Due to the random nature of Monte Carlo simulations, execution traces are not usually used with this approach. However, a Monte Carlo simulator may output an execution trace to facilitate analysis of its results.

3.10.2 Simulation in Usability Evaluation

Of the simulation methods surveyed in Chapter 2, all can be characterized as discrete-event simulations, except for information scent modeling [Chi *et al.* 2000]; information scent modeling is a close approximation to Monte Carlo simulation. Typical events modeled in these simulators include keystrokes, mouse clicks, hand and eye movements, as well as retrieving information from

Simulation Workload	Simulation Type	
	Discrete-event	**Monte Carlo**
Parameter Evolution	time-dependent	time-independent
Parameters	probabilistic	probabilistic
# of States	finite	finite
Trace Evolution	time-dependent	—
Parameters	probabilistic	—
# of States	finite	—

Table 3.8: Characteristics of simulation techniques in the performance evaluation domain.

Challenge	PE	UE
Modeling System	software program, simulation environment	UI development environment
Capturing Traces	record during measurement	usage traces (real & generated)
Using Traces	simulator reads & "executes"	simulate UI behavior realistically
Multiple Usage Scenarios	vary simulator parameters, multiple traces	usage traces (real & generated)

Table 3.9: Summary of the challenges in using PE simulation techniques in the UE domain. The PE column describes how these challenges are resolved within the PE domain, and the UE column describes potential ways to resolve these challenges within the UE domain.

memory. All of these simulation methods, except AMME (Automatic Mental Model Evaluator) [Rauterberg and Aeppili 1995], use fixed or probabilistic system parameters instead of usage traces; AMME constructs a petri net from usage traces.

3.10.3 Applying PE Simulation to UE

The underexplored simulation areas, discrete-event simulation with usage traces and Monte Carlo simulation, are promising research areas for automated UE. Several techniques exist for capturing traces or log files of interface usage [Neal and Simons 1983; Zettlemoyer *et al.* 1999]. As previously discussed, tools exist for automatically generating usage traces for WIMP [Kasik and George 1996] and Web [Chi *et al.* 2000] interfaces. One approach is to use real traces to drive a detailed UI simulation in the same manner discussed for measurement; AMME provides an example of this type of simulation. Such simulators would enable designers to perform what-if analysis and study alternative designs with realistic usage data.

Monte Carlo simulation could also contribute substantially to automated UE. Most simulations in the UE domain rely on a single user model, typically an expert user. One solution is to integrate the technique for automatically generating plausible usage traces into a Monte Carlo simulator; information scent modeling is a similar example of this type of simulation. Such a simulator could mimic uncertain behavior characteristic of novice users. This would enable designers to perform what-if analysis and study design alternatives with realistic usage data. Furthermore, the simulation run could be recorded for future use with a discrete-event simulator.

Table 3.9 summarizes challenges for using simulation in the UE domain as it is used in PE. The next section contains an in depth discussion of simulation solutions for the example tasks.

3.10.4 Example Simulators

For both of the example tasks, the assumption is that the UI is in the early design stages and consequently not available for running workloads. Hence, the designer must first construct a model to mimic UI behavior for each operation. The simplest way to accomplish this would be to expand a UI development environment (UIDE) or UI management system (UIMS) to support simulation. These environments enable a designer to specify a UI at a high-level (e.g., a model) and automatically generate an implementation.

Trace-driven discrete-event simulation is appropriate for simulating interface tasks such as the label creation example. However, all of the discrete-event simulators, except AMME, do not support executing usage traces. The drawback of AMME is that it requires log files captured during interface usage. The requirement of interface usage can be mitigated with a number of techniques previously discussed for automatically generating usage traces. Such traces could be saved in a format that a discrete-event simulator can process. In particular, traces need to be augmented with timing information.

The major difference between Monte Carlo and discrete-event simulation, especially simulation driven by usage traces, is that the task sequence is not pre-determined in a Monte Carlo simulation. This type of simulation is appropriate for the information-centric Web site example as described in the following section.

Monte Carlo Simulator for Information-Centric Web Sites

As previously mentioned, Monte Carlo simulation is appropriate for simulating information-centric Web sites, since this methodology does not require explicit task sequences. The survey in Chapter 2 revealed two related simulation methods: WebCriteria's Site Profile [Web Criteria 1999] and information scent modeling [Chi *et al.* 2000]. Site Profile attempts to mimic a user's information-seeking behavior within a model of an implemented site. It uses a idealist Web user model (called Max) that follows an explicit navigation path through the site, estimates page load and navigation times for the shortest path between the starting and ending points, and measures content freshness and page composition (amount of text and graphics). Currently, it does not use other user models, attempt to predict navigation paths, or consider the impact of other page features, such as the number of colors or fonts, in estimating navigation time. This simulation approach is more consistent with discrete-event simulation than Monte Carlo simulation, since it uses explicit navigation paths.

Information scent modeling is more consistent with Monte Carlo simulation and was developed for generating and capturing navigation paths for subsequent visualization. This approach creates a model of an existing site that embeds information about the similarity of content among pages, server log data, and linking structure. The evaluator specifies starting points in the site and information needs (target pages) as input to the simulator. The simulation models a number of agents (hypothetical users) traversing the links and content of the site model. At each page, the model considers information "scent" (i.e., common keywords between an agent's goal and content on linked pages) in making navigation decisions. Navigation decisions are controlled probabilistically such that most agents traverse higher-scent links (i.e., closest match to information goal) and some agents traverse lower-scent links. Simulated agents stop when they reach the target pages or after an arbitrary amount of effort (e.g., maximum number of links or browsing time).

The simulator records navigation paths and reports the proportion of agents that reached target pages. The authors comparison of actual and simulated navigation paths for Xerox's corporate site revealed a close match when scent is "clearly visible" (meaning links are not embedded in

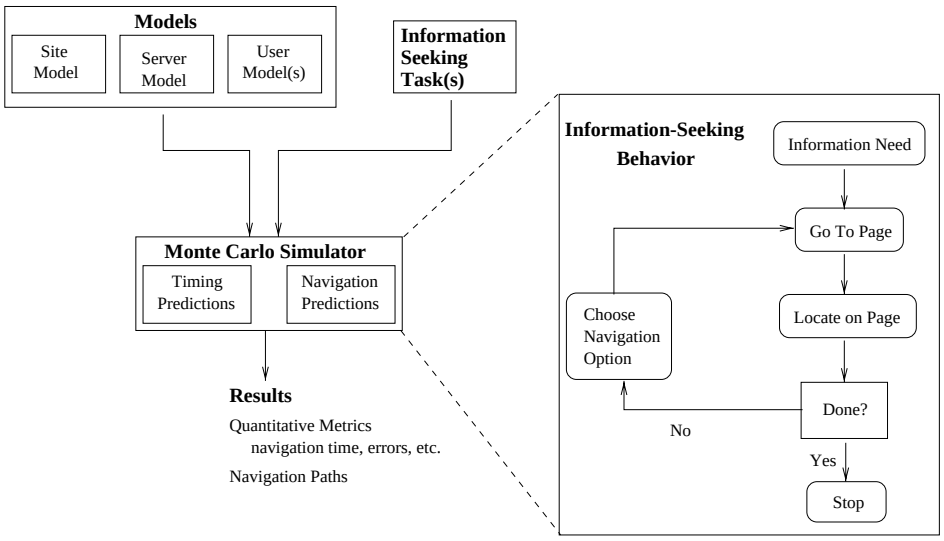

Figure 3.9: Proposed simulation architecture.

long text passages or obstructed by images). Since the site model does not consider actual page elements, the simulator cannot account for the impact of various page aspects, such as the amount of text or reading complexity, on navigation choices. Hence, this approach may enable only crude approximations of user behavior for sites with complex pages.

This section details a Monte Carlo simulation approach to address the limitations of the Site Profile and information scent modeling methodologies; some of this discussion was previously published as a poster [Ivory 2000]. Figure 3.9 depicts a proposed simulation architecture and underlying model of information-seeking behavior based on a study by Byrne *et al.* [1999].

A typical design scenario entails the designer initially creating several designs (i.e., site models) either by specifying information about each page, including the page title, metadata, page complexity and link structure, or importing this information from an existing site. The page complexity measure needs to consider various page features, such as the number of words, links, colors, fonts, reading complexity, etc.; the benchmarking approach discussed in Chapter 4 as well as the profiles developed in Chapter 6 provides insight for automatically determining a page complexity measure for existing sites, while estimates could be used for non-existent sites. The designer would also specify details about the server's latency and load (server model) and users' information tasks (e.g., destination pages and associated topics of interest). Finally, the designer would specify models of anticipated users with key parameters, such as the reading speed, connection speed, probability that a user will complete a task, read a page, make an error, etc. The designer could also specify constraints in the user model, such as an upper bound on navigation time or a small screen size, using production rules. It is possible to develop user models based on observed user behavior and reuse these models for simulation studies.

After specifying these models, the designer would then run the simulator for each design. Each run of the simulator would require the following steps. First, pick a starting page. There are three different models for how to do this: (1) User specified, (2) Randomly chosen independent of task (this may be used for assessing reachability), and (3) Chosen based on the task (this is equivalent to following a link returned by a search engine, a link from an external page, or a link from a usage trace). Next, repeat the steps below until either (a) the target page is reached; or (b) a stopping criteria is reached (e.g., maximum navigation time, all paths from starting point

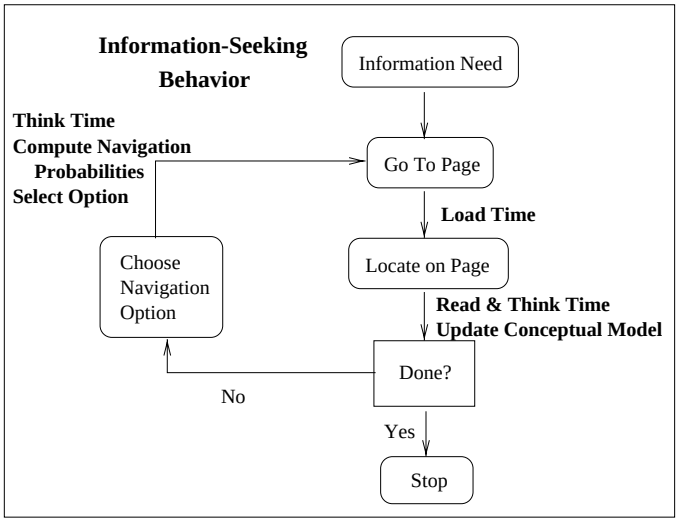

Figure 3.10: Simulator behavior during a run.

exhausted, or maximum number of links traversed). These steps are depicted in Figure 3.10.

Assuming the starting page is already loaded in the user's browser, the steps for simulating navigation are:

1. Read the page
   - This entails computing a read time based on the following:
     (a) page complexity (e.g., amount of text and reading difficulty)
     (b) page visitation (if previously viewed, then less read time)
   - Update the system clock after computing the read time.
2. Make a decision
   - Think time considers time for
     (a) deciding if target is reached (if so, end simulation run)
     (b) if not, deciding what to do next (this should be proportional to the number of options). This entails deciding on a navigation option (e.g., following a link or using the back button) as follows:
        i. survey options (create a list of choices)
        ii. compute a probability for selecting each option based on criteria described below; this procedure is not followed for the discrete-event mode, since the choice is dictated by the navigation trace.
        iii. prune options: if the probability for an option falls below a certain threshold, then eliminate the option.
        iv. choose option: if there is more than one option remaining, then the Monte Carlo algorithm determines the choice; this choice could be based on history (i.e., a user model or record of previous choices) or random.
   - Update the system clock after computing the think time.
3. Navigate

- This entails making the selected page the current page and marking it as visited. It also entails estimating a link traversal time and updating the system clock. This could be a fixed page download time or a variable that takes into account network variations, caching, and characteristics of the page itself.

Step 2b above requires computation of a probability for selecting a next move based on certain criteria. These include:

1. Metadata match: do a pairwise comparison of metadata between the current and potential page to compute a score indicating relatedness of content (it should be possible to pre-compute these for all links and store them in a structure); do a pairwise comparison of metadata between the target information and next page to compute another match probability; multiply the probabilities to compute a final match probability.

   Another possibility is to maintain a composite metadata analysis that's updated at each link. This composite analysis would take into account metadata on the previous pages traversed and the target information. This composite metadata could then be used in the pairwise comparison to compute a final match probability.

   Information foraging theory [Pirolli 1997; Pirolli *et al.* 1996] could be used for the metadata match algorithm. This approach has been used to present users with relevant Web pages in a site [Pirolli *et al.* 1996] and for information scent modeling [Chi *et al.* 2000].

2. Page visitation: if a page has been visited before, adjust selection probability according to the user model under use.

3. User model: adjust the match probability based on the simulated user. For example, if we are simulating a user learning by exploration, then the system state would reflect the user's current learning. If a page under consideration is consistent with a user's learning, then we would increase the match probability. Note that this criteria is different from (2), since it affects pages that have not been visited before.

The simulator could report simulated navigation time along with navigation traces to facilitate analysis and to possibly use with a discrete-event simulator. The designer could use simulation results for multiple site designs to determine the best information architecture for a site and to inform design improvements.

3.11 Summary

Using PE as a guiding framework provides much insight into creating new fully-automated (i.e., does not require interface use) UE methods. The analysis showed that the major challenges to address include: automatically generating high-level usage traces; mapping these high-level traces into UI operations; constructing UI models; and simulating UI behavior as realistically as possible. This chapter described solutions to these challenges for two example tasks.

The proposed simulation and analytical modeling approaches should be useful for helping designers choose among design alternatives before committing to expensive development costs. The proposed usability benchmarks should help assess implemented systems globally, across a wide range of tasks.

The remainder of this dissertation focuses on using PE as a guiding framework for developing a measurement approach for assessing Web interface quality. Specifically, it describes the

development of a Web interface benchmark consisting of over 150 page-level and site-level measures, the development of statistical models of highly-rated interfaces from these quantitative measures, and the application of these models in assessing Web interface quality and Web design guidelines.