Evaluating User Interface Systems Research

Dan R. Olsen Jr.
Brigham Young University
Computer Science Department, Provo, Utah, USA
olsen@cs.byu.edu,

ABSTRACT
The development of user interface systems has languished with the stability of desktop computing. Future systems, however, that are off-the-desktop, nomadic or physical in nature will involve new devices and new software systems for creating interactive applications. Simple usability testing is not adequate for evaluating complex systems. The problems with evaluating systems work are explored and a set of criteria for evaluating new UI systems work is presented.

ACM Classification Keywords
H.5.2 User Interfaces

General Terms:
Human Factors

Author Keywords:
User Interface Systems Evaluation

INTRODUCTION

In the early days of graphical user interfaces, the creation of new architectures for interactive systems was a lively and healthy area of research. This has declined in recent years. There are three reasons for this decline in new systems ideas. The first is that, unlike those early days, there are essentially three stable platforms (Windows, Mac, Linux) upon which virtually all software is built and those platforms have dictated the user interface architecture. This is in contrast to the state of UI research 15 years ago when there were many competing toolkits and platforms. The second is that the stability of these platforms has lead to a new generation of researchers who lack skills in toolkit or windowing system architecture and design. The third reason is the lack of appropriate criteria for evaluating systems architectures. This paper addresses the last question of "How should we evaluate new user interface systems so that true progress is being made?"

UIST'07, October 7–10, 2007, Newport, Rhode Island, USA.

WHY UI SYSTEMS RESEARCH?

Before addressing the evaluation question we must first consider the value of user interface systems research. The systems we have are stable. Applications are being written. Work is progressing. The users are happy (sort of). Why then does the world need yet another windowing system?

Forces for change

A very important reason for new UI systems architectures is that many of the hardware and operating system assumptions that drove the designs of early systems no longer hold. Saving a byte of memory, the time criticality of dispatching an input event to the right window or lack of CPU power for geometric and image transformations are no longer an issue. Yet those assumptions are built into the functionality of existing systems. The constraints of screen size are rapidly falling and we are finding that interaction in a 10M pixel space is very different from interaction in a 250K pixel space.

Our assumptions about users and their expertise have radically changed. Most of our windowing systems are designed to deal with a populace who had never used a graphical user interface. That assumption is no longer valid. The rising generation is completely comfortable with computing technology in a variety of forms and is increasingly comfortable with change.

Our existing system models are barriers to the inclusion of many of the interactive techniques that have been developed. Research as shown that manipulating the mouse gain can improve selection in various spaces [1] yet this does not fit smoothly into any UI system model. Cameras and touch tables produce inputs that are the size of a hand or finger rather than a point, yet we force such techniques into the standard mouse point model because that is all that our systems support. Multiple input points and multiple users are all discarded when compressing everything into the mouse/keyboard input model. Lots of good research into input techniques will never be deployed until better systems models are created to unify these techniques for application developers.

The advent of new interactive platforms also drives a need for new systems architectures. The WWW forms a huge base of interactive use, yet its interaction model is primitive and the toolkits built around it are difficult. People are increasingly moving their digital lives to PDAs, cell phones

and other portable platforms. Many people live and work across many platforms and interact with many people, yet our UI systems architectures support none of this.

In the early days of the Mac and Windows, an industry leader said "Almost none of our customers own a mouse or a graphics card. The installed market is too large and too entrenched to change." Barely 5 years later that company had fallen from market dominance to near irrelevance because the market had moved to screen/keyboard/mouse. We now stand at a similar position. Systems based on one screen, one keyboard and one mouse are the new equivalent of command-line interfaces. We need new UI systems solutions. Our current systems architectures are beginning to impede progress rather than empower it.

Value added by UI systems architecture

Before addressing the evaluation of research claims we should look at the value UI systems architectures bring to the table.

Reduce development viscosity

A good UI toolkit will reduce the time it takes to create a new solution. The faster new solutions can be created, the easier it is to try more solutions on users. The more solutions that users experience, the more effective the UI design process will be.

Least resistance to good solutions

UI programmers, like most programmers, are optimizers. They tend to follow the path of least resistance. Apple spent a lot of time and effort writing a style manual and evangelizing the value of a common look and feel. None of this effort had the impact of providing a standard widget set built into the system that was available for free. The toolkit made adoption of a common look and feel much cheaper and easier than custom solutions.

A related concept is that toolkits can encapsulate and simplify expertise. When exploration of a space of possibilities finally settles on a few good solutions, these can be packaged into a toolkit to simplify the development of future systems.

Lower skill barriers

Bill Buxton's Menulay [2] demonstrated that large portions of the UI design problem could be handled by drawing rather than code. Systems like HyperCard and Visual Basic commercialized these ideas and allowed people with a much different set of skills to participate in UI development. The right toolkit design meant that artists and designers rather than programmers were dictating the visual appearance of user interfaces. Good toolkit design can expand the set of people who can effectively create new applications.

Power in common infrastructure

Though the current mouse/keyboard event model is standing in the way of many new interactive devices and techniques, it has also empowered many new techniques. Pen-based interfaces have benefited from pretending to be a mouse. Though the pen has many unique advantages, by pretending to be a mouse it is usable for most existing applications. In our current event model, any device that can produce 2D input events is usable by any existing application. This has real power in supporting new combinations to create new solutions. What we need is an upgrade to this common denominator to empower more advanced techniques.

The HTTP/HTML standard is a clear example of the power of common infrastructure. By adhering to this simple standard any user can acquire a browser that will give them access to a vast number of services. Conversely any new service that adheres to the standard can gain access to multitudes of potential customers.

UI toolkits and system architectures define the ways in which interactive components are combined. The power of such techniques is also their curse. The importance of such standards makes the cost of changing them very high. However, if we do not search out and develop new ways to combine UI technologies we stunt our potential.

Enabling scale

Laying stable foundations makes possible larger more powerful solutions than ever before. The set of applications that can be economically built without a good toolkit is substantially smaller than with one.

EVALUATION ERRORS

Before exploring better ways to evaluate interactive software architectures, it is helpful to look at some ways that misapplied evaluation methods can damage the field. There are three to be discussed here: the usability trap, the fatal flaw fallacy and legacy code.

The Usability trap

When evaluating interactive systems the first concept that comes to mind is usability. There are several common measures of usability such as time to complete a standard task, time to reach a certain level of proficiency and minimize number of errors. Usability measures have driven a great deal of good research in the CHI community.

Many usability experiments are built on three key assumptions. The first is "walk up and use." This assumes that all potential users have minimal training. This is a great goal for home appliances and for software tools used by many people. This goal also works if there is a large pool of potential users with shared expertise, such as secretaries or chemists. The assumption is that anyone with the shared expertise should be able to walk up and use the system. The "walk up and use" assumption does not work well for problem domains that require substantial specialized expertise, such as user interface programming or design.

The second is the standardized task assumption. To make valid comparisons between systems one must have a task

that is reasonably similar between the two systems and does not have many confounding complexities. The task of entering a string of text is fairly standard across any set of text entry techniques. There are not many ways to approach the task. A task of painting a picture or designing a circuit is not a good task for a usability experiment. There are too many variables in the task to make valid comparisons. Individual differences in creativity and skill create too many variations in the potential user population. A task that is suitable for a usability experiment must have low inherent variability so that any variance can be assigned to the differing techniques being tested not to variations in approach to the task or user expertise.

The third assumption is scale of the problem. The economics of usability testing are such that it must be possible to complete any test in 1-2 hours. More extensive testing may allow multiple sessions over many weeks but any practical experiment cannot afford thousands or tens of thousands of experiment hours. A usability test of a toolkit that requires 10 programmers over 6 months would incur at least $300,000 in user subject compensation for just that one test. The cost is high and the statistical significance very low.

Usability testing is attractive because it can produce a statistically valid, clearly explained, easily compared result. However, toolkit and UI architecture work rarely meets any of the three assumptions of usability testing.

Usability testing of interactive tools and architectures

Any UI toolkit that addresses substantive problems will require expertise in using the toolkit. By definition any new toolkit will have no population that possesses that expertise. Any comparison between a new toolkit and an existing toolkit will be confounded by familiarity with the existing toolkit and the need for expertise in using the new one. The only way to eliminate this confound is to find a population that is equally ignorant of both systems. This produces comparison results that are not representative of the intended populations of either tool.

The standardized task assumption is also violated in systems architecture research. Any problem that requires a system architecture or a toolkit in its solution is by nature complex. UI applications are known to be complex with many possible paths to a solution. Meaningful comparisons between two tools for a realistically complex problem are confounded in so many ways as to make statistical comparisons more fantasy than fact.

The task scale assumption is also violated by UI systems work. Building a significant application using two different tools, even if valid comparisons were possible, would be very costly. Performing many iterations on a toolkit solution using this evaluation technique would be out of the question.

The usability trap lies in how we respond to nature of usability measurement for systems architectures and toolkits. There are those who respond with "If it can't be measured it is not research." This reduces research to the study of the trivially measurable. There is a slightly different response which is "Focus on the measurable it is easier to publish." This limits our power to effect significant change. A much more interesting response would be "If not usability then how do we evaluate systems?" This is the topic of this paper.

The fatal flaw fallacy

When evaluating small interactive techniques or specific behaviors it is good practice to carefully examine all of the possible ways in which the technique or its validation might be in error. If such a fatal flaw is discovered then the reported results may not be valid and the work should be remanded to the authors for repair.

This search for fatal flaws is devastating for systems research. It is virtually impossible for a small team of researchers to recreate all of the capabilities of existing systems or to completely examine all of the eventualities of new concepts. The farther such a team reaches into new territory the more compromises will be required and the more supporting ideas must be left unresolved. If a new systems approach is attempted, the omission of some important feature is guaranteed. The existence of a fatal flaw is a given. If the evaluation of the work is focused on "what does it not do" no research system will ever pass. Flaw analysis will frequently be a barrier to new systems research.

Legacy code

In the late 1970s many people objected to new UI architectures because of the vast amount of legacy code written with command-line or text screen interfaces. When the Macintosh hit the market followed by Windows suddenly all that legacy code became irrelevant. The WWW has precipitated a vast rewrite of the ways in which companies deal with the public. Many desktop applications have been rewritten for cell phones and PDAs. And yet people still invoke the "legacy code" standard for UI systems research.

If a toolkit can run legacy applications while providing some new advance that is a good thing. If a new architecture necessitates rewriting applications, that is just the price of progress. The legacy code requirement is a barrier to progress.

EVALUATING EFFECTIVENESS OF SYSTEMS AND TOOLS

To find ways to evaluate systems and tools we must revisit the claims that are made. Our list of advantages presented for toolkits and systems provides us a framework of possible claims. Given a set of possible claims we can then outline ways to demonstrate each of them.

STU: Situations, Tasks and Users

Every new piece of interactive technology addresses a particular set of users, performing some set of tasks, in some set of situations. It is critical that interactive innovation be clearly set in a context of situations, tasks and users. The STU context forms a framework for evaluating the quality of a system innovation.

In systems and toolkit work there are frequently two different STU contexts. The users of a toolkit are developers. The task is to design quality interfaces that fit some end user STU context. For example we may develop tools for programmers (users) who are creating applications (task) for doctors (end user) accessing medical records (end task) while making hospital rounds (end situation). These tools might be designed to allow programmers (users) to do development while following doctors around (situation).

Importance

Before all other claims a system, toolkit or interactive technique must demonstrate importance. Tools are invariably associated with expertise gained over time. People will not discard a familiar tool and its associated expertise for a 1% improvement. In most cases at least a 100% improvement is required for someone to change tools. Without establishing the importance of the problem and its proposed solution, nothing else matters.

Importance analysis proceeds directly from the intended STU context. The first question is the importance of the user population (U). Populations can be important because they are large (every sentient being over the age of 10). User populations may be important because their performance is critical to the well being of others (doctors, emergency personnel, or teachers). User populations may be important because of tremendous need (disabled, poverty). Every population is important to itself. We must give some consideration to the population's importance to society in general. Sports fans in the bleachers behind left field at Yankee stadium would not be a very important population no matter how fervid they may be.

Assuming that the user population is important we must then evaluate how important the target tasks (T) are to that user population. Importance might be established by how frequently the task occurs. It might also be established by looking at the consequences of not being able to do the task. It is a serious problem if a doctor cannot access a patient's records.

We must look at the set of situations (S). How often do the target users find themselves in these situations and do they need to perform these tasks (T) in those situations? We must also look at the importance of the STU context as a whole. Many doctors go scuba diving. Doctors are important. Scuba diving is a novel situation but accessing patient records while scuba diving does not seem particularly important.

Lastly we must consider the difference that our new technology will make. Changing work practice requires a lot of effort. Although data analysis is good, if the difference offered by our new technology must be carefully analyzed for statistical significance in order to establish an improvement, then it is probably not important.

Problem not previously solved

This is one of the more compelling claims for a tool. This claim says that there is a STU context that has no current solution. It is a powerful claim to demonstrate that T can be performed effectively with a new tool. Usability testing is irrelevant when comparing what can be done against what cannot.

Such a claim is seriously weakened if there is only a single task T. What has been created is not a design tool but an application. That application may be important to the user population U, in which case it may be more appropriately published in the literature of population U. This novelty claim is strengthened if population U is very, very large (i.e. everyone who can read). The larger and more diverse the STU context is, the stronger the claim to the importance of a solution. The World Wide Web could make the claim that previously it was impossible for non-programmers to interconnect many disparate information resources.

Generality

The new solution claim is much stronger if there are several populations U_i that each have tasks T_i that do not have effective solutions with existing technology. If the new tool can solve all of T_i then a claim for a general tool is quite strong. The generality of the new solution claim is strengthened as the populations U_i are increasingly diverse from each other.

The problem with this claim is that proving solutions for all T_i is not possible. In fact the more general the tool the less likely one can demonstrate all of the possible solutions for which the tool is useful. The best proof of a generality claim is the diversity of the STU contexts for which a solution is demonstrated. If one has used the tool to solve three diverse problems then one can argue that the tool solves most of the problems lying in the space between the demonstrated solutions. The greater the diversity and the larger the number of demonstrated solutions, the stronger the generality claim.

Reduce solution viscosity

One of the important attributes of good tools is that they foster good design by reducing the effort required to iterate on many possible solutions. The more cumbersome the tool, the greater the viscosity in the design process with fewer and less diverse alternatives being explored. There are at least three ways in which a tool can reduce solution viscosity: flexibility, expressive leverage and expressive match.

Flexibility

A UI tool is flexible if it is possible to make rapid design changes that can then be evaluated by users. This has been the claim of interpreted programming languages. By eliminating the compile step, it is faster to try many different solutions. By evaluating code created at run-time more flexible solutions are possible. Visual UI design tools exhibit flexibility when one can rapidly make changes to the appearance of the user interface.

The flexibility claim is relatively easy to support. Define some interesting and diverse set of possible design changes and show that such changes take significantly less effort in the new tool relative to the competition.

Expressive Leverage

Expressive leverage is where a designer can accomplish more by expressing less. The dominant cost of any design processes is the making, expression and evaluation of choices. Expressive leverage is achieved when a tool reduces the total number of choices that a designer must make to express a desired solution. There are several ways this might be done.

Eliminating repetitive choices is an easy technique for achieving leverage. The simplest mechanism is reuse. The claim is that a large class of STU contexts include the expression of choice Y, yet Y is the same across all of these solutions. A tool that encapsulates Y so that the choice is made only once is more expressive. One must show that every solution in the class of solutions includes Y and that every instance of Y is similar or show that the differences are easily parameterized. The size/importance of the STU contexts in which choice Y must be made is key to this claim.

There are pitfalls to the "generalize and reuse" strategy for expressive leverage. In many cases the manipulation of the parameters is more complex than redoing a custom instance of Y. This comes when there is more variation in Y than was originally anticipated. This also comes when one generalizes too far.

A good example of over-generalization is the Table widget. At first glance it has rows, columns and cells. Columns have headers and cells contain text. It is relatively easy to write such a widget and its interface to the data for the model is quite straightforward. The temptation, however, is to add fonts, colors, special layout rules, arbitrary data types for the cells, icons for the columns and specialized cell editors with four or five different ways of embedding those editors into the cells/table layout. Suddenly the correct specification of the parameters to the Table widget is more complex and less predictable than implementing it from scratch. A generalize and reuse strategy must demonstrate that there has been a clear reduction in the number of choices that a designer must make and that the implications of those choices are clear to the designer.

A second means for reducing the required number of design choices comes from observing that many choices are not independent but are almost always implied by other choices. Having made choices A, B and C, then design choices U, V and W are determined. We get our expressive leverage by automatically computing U, V and W from A, B and C rather than requiring the designer to express them. An example would be the automatic calculation of complimentary colors once a background color is selected. A variation is that default values U, V and W are computed and then only manually changed when necessary. In the common cases the complexities of U, V and W are gone.

A third means for achieving expressive leverage comes from the progress of technology. For example the GIGO event handling system [3] was widely acclaimed in its time because of its small memory footprint and that event dispatch that could be accomplished in less than 10 machine instructions. For years various event dispatch mechanisms were proposed to optimize the interactive loop. This need for optimal event dispatch imposed various design choices on programmers to achieve the desired speed. When 500 KHz processors became 3GHz processors these design choices became irrelevant. Eliminating such choices and simplifying what a designer must do becomes possible as more memory, processor and communications power becomes available.

Claims of increased expressive leverage may often be unjustly discounted by reviewers. Most of the leverage comes from some insight about the way in which systems have been used or developed. Once the insight is exposed, the implications are obvious. It is very tempting for reviewers to site the obvious nature of the implication and thus discount the value of the insight. It is incumbent on the reviewer to demonstrate that the insight is either trivial or was already known.

Expressive Match

Tools for creating new user interfaces can be improved by increasing the expressive match of the system. Expressive match is an estimate of how close the means for expressing design choices are to the problem being solved. For example one can express a color in hexadecimal or one can pop up a color picker that displays the color space in various ways and shows the color currently selected. Both are completely accurate means for expressing color, but the color picker is a much closer match to the design problem.

Most interface design environments provide a tool for placing widgets in a form and dragging them around until the resulting layout is visually appealing and readily understood. Encoding the coordinates of each widget in C++ or a text file will accomplish the same goal, but the visual tool is a better expressive match for the task of creating a usable and pleasing layout.

There are several requirements when making a claim of greater expressive match. One must demonstrate that the

new form of expression is actually a better match. If the problems are small enough one can compare times to create a design or express a set of choices. This may or may not fall into the usability trap described above. Frequently greater expressive match is tied to a claim to lower skill barriers.

Another test for expressive match is a "design flaw challenge". A design that is deficient in some way is encoded in the two different forms. Users are each given one of the forms of expression and asked to locate and remedy the flaw. Time, errors, difficulties and success rates can be used to compare two forms of expression. In many cases such challenges are trivially obvious. "Match this color" can be given in hex or in a color picker. For most people the task is virtually impossible in hex and even for experienced designers there is a lot of trial and error required. Similarly a flaw in the input handling of some widget can be posed. Some designers are given state diagrams and some are given the equivalent Java/Swing code. Each is challenged to find and remedy the flaw.

Greater expressive match sometimes introduces difficulty in integrating multiple forms of representation and their associated tools. A long term challenge in the UI community has been the integration of various visual representations with program code. Despite years of research and dozens of alternative proposals, algorithms and data structures are still best represented in a general-purpose programming language. State machines showed real promise as a representation for input, yet they always reached expressive limitations that required a transition to code and the transition was awkward. Visual Basic addressed this problem by integrating a full interpreted programming language into its UI layout tool. The key point is that any new form of expression must be computationally complete or must have a clear integration mechanism with a programming language. When there are many different forms of expression, the tools must show how they integrate effectively with each other. In many cases the integration effort swamps the benefits of the new tool or notation.

Empowering new design participants

The previous set of claims focused on the speed or ease with which a user interface could be designed. Tools can also make a contribution by introducing new populations to the UI design process. Frequently this is done by dealing with expressive leverage and expressive match issues, but the claims are different. The "new design participants" claim is that there is some population U who would benefit by being more directly involved with the UI design process. It has long been claimed that empowering artists will lead to better visual designs. Participatory design advocates the involvement of end-users in the design process.

The first part of the claim must be to describe some population of participants and show why they should be involved in the UI creation process. Secondly one must establish why existing tools are not acceptable for this population. This may include lack of appropriate training, different norms of expression or design goals that are not supported by existing tools. Lastly one must demonstrate that the new tools are accessible, easier or more effective for this desired population.

The simplest criterion is that the existing tools simply could not be used by the target population at all. Demonstrating that artists cannot reliably encode colors in hex is an easy claim to make. A second claim is that the new tools are easier for this population. Here usability tests are possible. However, the representation may not scale to full-sized UI designs and the usability trap will be reintroduced. There is also an importance issue. If there is not a substantial improvement in ease or usability to this population or the population is only peripherally involved then there just may not be much value in the new tool. People will only change tools for substantial (2 times or better) improvements. The last claim is that the resulting UI designs are "better" when the new tools are used by this population. The question of defining "better" is problematic.

Power in combination

Many tools demonstrate their effectiveness by supporting combinations of more basic building blocks. There are two basic variations of this claim. The first is an inductive claim that an infinite set of solutions can be built from primitives and their combinations. The second is the N to 1 reduction. Both of these approaches are based on clearly defining mechanisms for combining pieces of design to create a more powerful whole.

Inductive Combination

This is the basis for grammars, UI component trees and a variety of other innovations. The idea is that there is some set of primitive design components and some mechanism for combining them into more complex designs. Some simple set of string tokens can be combined using non-terminals and production rules to create an infinite number of different programming languages. Simple widgets that each perform some interactive task can be combined using panes, tabs, tables and other visual constructors to create an infinitely diverse set of UI designs.

Making this sort of claim requires that one show that the set of primitives is either relatively small or can be easily extended. User interface toolkits use both of these strategies. There is usually a small set of primitive widgets (less than 30) from which many interesting designs can be built. In addition, most toolkits provide a mechanism for creating new primitives that can be readily integrated into the set. The power comes from the means for combining these pieces into more complex designs. The induction to an infinite set of possibilities comes when any of these combinations can be used wherever a primitive solution can be used.

This claim introduces the question of coverage. For example, context-free grammars define an infinite set of textual languages. However, it can be readily proved that they cannot fully represent most programming languages. It can also be readily shown that combining menus, scroll-bars, labels and text boxes into forms can produce an infinite number of UI designs. However, many UI designs (painting, drawing or animation time lines) will not fit this model. Virtually any combination scheme will leave out some design solutions. Therefore one must show that the set of designs within the system are interesting and that there are mechanisms outside the system for addressing issues that are not covered. Compilers added symbol tables to context-free grammars to make them effective. UI toolkits generally provide a means for adding new primitive components for extensibility.

Simplifying Interconnection

There are many situations where components must communicate and integrate with each other. This interconnectivity is critical to the deployment of new technology. A system of N components that must work together imposes a serious burden on the N+1 component. If every component must implement an interconnection with every other component then the N+1 component must include N interconnections with other pieces. A good interconnection model will reduce the cost of a new component from N to 1. With a good interconnection model the new component must only implement the standard interface. It will then be integrated with all N existing components.

An early example was the use of pipes in UNIX. Applications would read from standard-in and write to standard-out. Pipes could connect the standard-out of one program directly to the standard-in of another, allowing applications to be plugged together in any number of useful ways without new code.

A more compelling example is the World Wide Web. When one creates a new web browser one automatically inherits the value of all of the web sites that have been created. When one creates a new web site one inherits the value of all of the browsers that have been installed. Adding a new browser or a new site incurs only the cost of that site and not the cost of creating, connecting or deploying the remainder of the needed infrastructure. The power is in the connectivity architecture that reduces cost from N (one for every related component) to 1.

To make the interconnection claim, one must show 1) that there is an interesting diversity of choices on both sides of the connective architecture, 2) that they are all cleanly supported by the architecture, and 3) that the space of combinations is interesting and non-trivial.

Ease of Combination

The fact that architectural components can connect is generally not sufficient. It is important that the interconnection be simple and straightforward. In most UI toolkits a new widget must only implement the *redraw()* and *resize()* methods to integrate with the rest of the system. Additional event-handling methods will provide interactivity but these two are sufficient. Many of the interconnection complexities are hidden in inherited code. The WWW has a similar ease of combination. The HTTP protocol that connects browsers to servers is very straightforward and easy to learn. My personal counter example is SOAP which took the relatively simple HTTP and XML techniques and created a very complex model for interconnecting with web services. This model is very difficult to use without additional layers of code generation to obscure its complexity.

Can it scale up?

An important question that must be asked of every new UI system is whether it can scale up to large problems. This was the fundamental drawback of state machines for describing user interface dialogs. For simple examples like dragging a rubber-band line, the state machine dialog was clear and direct. However, for any reasonable application the representation acquired hundreds of states interconnecting in hundreds of ways that were impossible to visualize, present on a screen, or debug. Constraint systems have similar problems. They nicely model small local relationships yet produce serious debugging challenges when hundreds of constraints are all being evaluated simultaneously. Any new UI system must either show that it can scale up to the size of realistic problems or that such scaling is irrelevant because there is an important class of smaller problems that the new system addresses. To evaluate this criteria one must try the system on a reasonably large problem and show that the advantages of the new model still hold.

SUMMARY

User interface technology, like any other science, moves forward based on the ability to evaluate new improvements to ensure that progress is being made. However, simple metrics can produce simplistic progress that is not necessarily meaningful. Complex systems generally do not yield to simple controlled experimentation. This is mostly due to the fact that good systems deal in complexity and complexity confounds controlled experimentation. This paper shows a variety of alternative standards by which complex systems can be compared and evaluated. These criteria are not novel but recently have been out of favor. We must avoid the trap of only creating what a usability test can measure. We must also avoid the trap of requiring new systems to meet all of the evaluations required above. This would recreate the fatal flaw fallacy. We must look to our evaluation strategy to answer the fundamental question "Has important progress been made?" If the answer is yes then we happily take our share of the new knowledge and move forward to fill in the gaps.

ACKNOWLEGEMENTS

Though the opinions are the author's own they have been influenced by discussions with Scott Hudson, James Landay, Saul Greenberg and Ben Bederson.

REFERENCES

1. Blanch, R., Guiard, Y., and Beaudouin-Lafon, M., "Semantic Pointing: Improving Target Acquisition with Control-Display Ratio Adaptation" *Human Factors in Computing Systems (CHI '04)*, ACM (2004), pp 519-526.
2. Buxton, W., Lamb, M. R., Sherman, D., and Smith, K. C., "Towards a Comprehensive User Interface Management System" *Computer Graphics (SIGGRAPH '83)*, ACM, (1983), pp. 35-42.
3. Rosenthal, D. S. H., "Managing Graphical Resources" *Computer Graphics 17(1)*, ACM (1983), pp. 38-45.