Reasoning About Interactive Systems

VINCENZO AMBRIOLA AND DAVID NOTKIN

Abstract—Interactive systems have goals and characteristics that differ from those of batch systems. These differences lead to a need for new techniques, methods, and tools for manipulating and constructing interactive systems. We consider the difference in structure between batch and interactive systems, focusing on the distinction between command decomposition and component decomposition. We describe the possible ways of solving a problem using an interactive system using action paths, which account for the relatively unconstrained actions of interactive users. We demonstrate that interaction is not an inherent characteristic of a system but rather a characteristic that depends on the error profile of its users. We consider the requirements that interaction places on the underlying implementation, specifically the need for incrementality and integration. We apply our results to several existing classes of systems.

Index Terms—Incremental computation, integration, interaction, interactive systems, program decomposition, user error profile.

I. INTRODUCTION

Interactive systems are becoming more plentiful. The successes of word processors and spreadsheets, for example, are evidence that interactive systems are attractive to a wide range of users. Interactive systems have evolved with the technological advances that have led to powerful displays and processors. Along with this evolution has arisen a change in the fundamental economics of systems. Previously, systems were planned so as to ensure the processor was used effectively. Now, systems should be planned so as to increase the productivity of users [3], [13].

Due to this new orientation, it is not surprising that the goals and characteristics of interactive systems differ so greatly from those of batch systems that new techniques, methods, and tools are needed. We describe an informal model of the structure, use, and implementation of interactive systems. Each part of the model emphasizes the importance of users when considering interactive systems.

Manuscript received August 30, 1985.

V. Ambriola is with the Dipartimento di Informatica, Universita di Pisa, Pisa 56100, Italy.

D. Notkin is with the Department of Computer Science, University of Washington, Seattle, WA 98195.

IEEE Log Number 8718392.
Table I

<table>
<thead>
<tr>
<th>Batch</th>
<th>Interactive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text Editing</td>
<td>Stream Editors</td>
</tr>
<tr>
<td>Program Development</td>
<td>Compilers</td>
</tr>
<tr>
<td>Desk Calculators</td>
<td>Fortran</td>
</tr>
<tr>
<td>Document Processing</td>
<td>Document Compilers</td>
</tr>
</tbody>
</table>

Fig. 1. An action path.

user—no commands are available. The user has solved the sub-
problem addressed by the third component by applying four com-
mands from the pool of two commands that are provided.

An action path cannot be known before system execution. How-
ever, the possible action paths can be characterized. Consider a
system $S$ with a set of commands $C = \{ C_1, C_2, \ldots, C_n \}$. Each in-
teractive component $C_i$ has a set of $k_i$ commands $C_i = \{ \tau_1, \tau_2, \ldots, \tau_{k_i} \}$. Given this system with the specified component and command decomposition, the set of action paths that a user may use to solve a problem is given by

$$(C_1 \circ C_2 \circ \cdots \circ C_n)^*$$

where each

$$C_i = \begin{cases} 
C_i & \text{if } C_i \text{ is batch} \\
C_i (\tau_1|\tau_2|\cdots|\tau_{k_i})^* & \text{if } C_i \text{ is interactive.}
\end{cases}$$

Consider the case where $S$ has only a single component $C_i$. In
this case, the description of the action paths reduces to

$$C_i (\tau_1|\tau_2|\cdots|\tau_{k_i})^*,$$

which in turn reduces to

$$C_i (\tau_1|\tau_2|\cdots|\tau_{k_i})^*.$$

Systems of this form tend to encompass the wide variety of systems
that are described by the ever-present buzzword “integrated,” since
all processing is done entirely in the context of a single component.

Some interactive systems do not permit users to apply all com-
mands at arbitrary times. Template-based structure editors are ex-
amples of such systems. For these systems, our characterization of
the action paths is inaccurate, but the modification needed to ac-
count for these kinds of restrictions is straightforward.

III. USER-DRIVEN “WITH RESPECT TO”

Arguments over the merits of different approaches to solving a
problem are common. For instance, one group of users prefers text
editors and compilers while another group prefers structure editors
integrated with incremental compilers. Or, for example, there is a
group that prefers WYSIWYG\textsuperscript{G} document processors while another
group prefers document compilers. In many cases, disagreements
about the approaches degenerate to “religious” arguments. A more
orderly way of comparing and contrasting these systems is needed.

Cost is the primary notion used to compare different systems. It
solve the same problem. While other metrics, such as storage space
or memory, can be used, the most common way or comparing costs of
computer systems is speed. The benefits of different systems ran
ultimately be viewed in terms of the time it takes to solve a
problem. This is especially true for interactive systems where the user’s
productivity is the focus.

The reason that the arguments over different systems often de-
gennerates is that the cost of solving a problem using a system varies
enormously from user to user. The cost of using a specific system
may only be discussed with respect to (wrt) a specific user or class
of users. A system may yield acceptable performance for one user,
in which case the system is user-driven wrt that user. The same
system may not yield acceptable performance for another user, in
which case the system is not user-driven wrt this user. For example,
take a satisfied user who uses a screen editor at 1200 baud at
different times and places. If a 9600 baud line is placed in the
office, the “new” user may no longer be in the same system,
the editor from home. The editor is no longer user-driven wrt the user.

This focus on the user clarifies much of the discussion about
novice and expert users. On the one hand, novice users are often
learning to use a system. In this experimental stage, they frequently
make mistakes. Because of this, novices usually benefit the most
from a system in which the cost of each command is very small.
Otherwise, they must first wait a long time to see the results of
their command. Then, if a mistake was made, the effect of the
command must be undone, which may be costly, and a new com-
mand applied, which may also be costly. This makes it difficult for
a system to be user-driven wrt a novice. On the other hand, expert
users are by definition intimately familiar with the system. Experts
make mistakes less frequently, so they are willing to accept greater
cost for each command; also, they are more experienced with
methods for quickly repairing any errors that they do make. This
makes it easier for a system to be user-driven wrt an expert.

The key to this notion is that the error profile of a user affects the
action path that is used to solve a problem. Consider a novice
that uses a system that is user-driven wrt an expert. As the novice
makes frequent errors, a short path used by an expert is applied
only with low probability. Longer paths of expert commands are
applied with high probability, and the cost of the entire solution is
greater since each command costs more. Consider instead an expert
that uses a system that is user-driven wrt a novice. As the expert
makes infrequent errors, a short path of the novice commands is found with high probability. However, this short path is much longer than the path required by the expert
who applies expert commands, and hence the cost of solving the
entire problem is greater. The key is that the probability profile of
a user affects the benefits yielded by a specific system selected to
solve a problem.

The error profile of an expert is more complex than that of a
novice, as their experience makes them aware of the costs of their
actions. Experts about to execute an expensive action will be more
careful, which directly reduces their error profile. Novices tend to
assume, until they have learned by trial and error, that the cost of
each action is constant. Since most systems do not have actions with constant cost, this understanding decreases the productivity of
many novices.

Even if actions with varying costs are considered, the profile is
not sufficient. The variance of the cost of applying commands must
be considered as well. A high variance in the costs of the com-
mands is especially annoying to a novice user that believes that all
commands have similar cost. A command that takes a long time
when a novice does not expect it might even convince the user that
the system is broken rather than slow. Experts may well deal with
high variances better since they have some knowledge about the
cost of commands before they are applied. Providing undo and redo
commands [1], [17] is one approach towards smoothing out the
error profile of users. By making it as easy to fix mistakes as to
make them, the cost of making the errors is reduced and hence the
profile reflects commands with closer to constant cost.

The spectrum of error profiles contains more than just novices
and experts. The road between novice and expert, for instance, is
continuous and not always monotone. This spectrum also includes
intermittent users of systems, who act like “smart novices.” That

\textsuperscript{11}What you see is what you get.”
there are users with different profiles make it difficult to produce systems that are user-driven wrt a wide range of users. That users change their individual error profiles makes it hard to produce systems that are user-driven wrt a given user over time.

IV. THE QUALITY OF AN INTERACTIVE SYSTEM

Considering an interactive system in terms of system structure and error profile makes it easier to understand and address problems with real systems. Assume a user is unproductive when using a specified system. Each of the three pieces—the component decomposition, the command decomposition, and the user’s error profile—must be studied in relation to the difficulty.

If the difficulty is general in that the user cannot successfully work through any part of the system, then the problem is fundamental. The two solutions to this problem are either to train the user (to reduce the error profile) or to modify the system to meet the user’s understanding of problem solving. Discussions of appropriate ways to make this choice are available elsewhere [7].

If the best solution is to train the user, then the solution is beyond the scope of this paper. If the best solution is to restructure the system, several approaches are available. First, related components can be merged together. This is appropriate where high error rates in one component, usually a batch component, overwhelm the benefits of a related component, usually an interactive component. This is the motivation for combining an editor, interpreter, and debugger in many novice programming environments. Second, a single component, usually a batch component, can be decomposed into smaller pieces. This is appropriate where a user is frustrated by the need to apply many short commands that could be described more briefly with another decomposition. This is the reason that experts do not usually like novice programming environments. Third, an entirely new system structure can be developed. This is a different task from modifying an existing system.

Sometimes the user is unproductive due to difficulties with an individual component. If the component is batch, then the only solution to the problem is to make it interactive. If the component is interactive, then either the component should be rewritten as a batch component, or else a different command decomposition for the component is needed. Line-oriented and full-screen text editors are examples of a component with different command decompositions.

This analysis reflects only on the structure and not on the form of the system and its interface. Others have studied principles and techniques for designing the form of these interfaces [4], [8].

The discouraging part of this analysis is exactly what was recognized in the discussion about error profiles: it is extremely difficult to build a system that meets the needs of various users with changing error profiles. In some cases, it is possible to make headway towards supporting some of these modifications in the context of an existing system. One possibility is for the expert to combine multiple commands, if the system has the needed support. For example, many text editors, such as Emacs [14], and many command languages, such as various UNIX® shells, permit the users to define procedures and macros that can be executed in the same way as primitive commands. In this manner, users may help evolve an existing interactive component towards a new component that better meets their needs. Further, they can do this on an individual basis, so that the system can be used in different ways by many users.

V. IMPLEMENTATION OF INTERACTIVE SYSTEMS

Since the composition of components during execution is inexpensive, the entire implementation of a system may be viewed in terms of the implementation of the individual components.

Batch components can be successfully implemented through the use of standard techniques that lead to the refinement

\[\text{decode} \circ \text{compute} \circ \text{encode}.\]

*UNIX is a registered trademark of AT&T Bell Laboratories.

Again, compilers are one of the best known examples of batch components. The decode phase applies parsing techniques to map program text into internal program structures like trees. The compute phase massages these internal structures to solve tasks such as static-semantic checking and code generation. The encode phase maps the internal structures into listings, which can be understood by a user, object images, which can be understood by a loader, and so on. The decode and encode phases are needed since the component is independent of other components: a compiler does not care which text editor is used to produce the input or which loader uses the generated object image.

Implementation of interactive components can be viewed in terms of the implementation of individual commands, each of which can be refined as

\[\text{input} \circ \text{compute} \circ \text{output}.\]

While the implementation of each command is similar in structure to the implementation of a batch component, there is a fundamental difference. Each execution of a batch component accepts no state from another component, except in terms of the input. Each execution of an interactive command, however, can rely on the program state of the previously executed commands, in addition to the input of the command. In other words, the batch component is an independent unit that computes a mathematical function, while the smaller-grain interactive command is a piece of a state transition process. Command-input differs from component-decode as it is not responsible for decoding the entire state of the computation. Similarly, command-output differs from component-encode as it is not responsible for encoding the entire state of the computation.

In most interactive components, the cost of the input phase is inherently small. In many cases, the primary cost is that of searching a table of command names. Commands that require additional input are usually designed with brevity in mind.

The property of incrementality states that the compute phase of an interactive command must have low cost [9]. The compute phase of an interactive command may cost less than the encode phase of a batch component (again due to the difference between functions and state transitions). The compute phase of the batch component generally manipulates all the input (otherwise, a portion of the input would be unnecessary). The compute phase of the interactive command, however, need not manipulate all of the program state. Consider a command that transposes two characters in a sequential file. The command that implements the transposition in an interactive text editor need not look at any characters other than the two being manipulated. A batch component—such as a stream editor—must access the entire file to transpose only two characters.

As an example of the benefits of the incrementality of interactive commands, consider Reps’ incremental attribute grammar algorithm [11]. When a change is made to an attributed tree, an algorithm guaranteed to modify only the minimal number of attribute values is applied. In this way, all nodes whose attribute values will not be affected by the modification will not be considered by the algorithm.

The property of integration states that the intermediate states must be quickly made visible to the user through the output phase of an interactive command [9]. If the mapping from the internal state to the user display is too costly, then the integration decreases, because the user’s attention will wane. However, the output phase can often be faster than the encode phase of a batch component, since output need not depend on the entire state. Instead, just as the compute phase could manipulate a subset of the internal state, the output phase need only modify the user’s display at the points affected by the changes made by compute. In the Descartes project, this appears as a principle requiring strong linkage between the display and the program [12]. Incremental updating of the display is common for text based systems [6], and the approach is made easier because the internal
representation—perhaps a linked list of lines—and the external representation are so closely related. In syntax-directed editors, where the internal representation and the external representation are quite different, a more complex incremental output mapping is required. One of the reasons that many WYSIWYG systems are successful is that this mapping is simple. Garlan has demonstrated an approach and algorithms that lead to incremental output mappings for syntax-directed editors [5]. This success is promising as it shows that at least some more complex structures may be mapped efficiently to the display.

VI. IMPLICATIONS

The relationships among the structure, the user’s error profile, and the implementation comprise an informal model of interactive systems. We now demonstrate that this informal model is natural—that is, that it reflects real interactive systems.

Interaction in Emacs

There is an overwhelming consensus that Emacs [14], a well-known class of full-screen text editors, is both interactive and successful. The reason is that the characters on the screen reflect exactly the state of the buffer being edited. The closeness of the mapping from the internal editor state to the screen, as stated before, allows for speeded redisplay after command execution. Another reason is that most commands can be applied instantaneously, so that the user need not lose attention. Third, the extension mechanism of Emacs makes the system attractive to a wide variety of users. Users of Emacs may define macros, which simply compose commands, or they may define new functions, which can take advantage of the flow-of-control constructs of the underlying extension language. This mechanism allows users to slowly increase the power of the commands that they apply. Actions that take a novice many keystrokes can be applied far more quickly by an expert. The extension mechanism makes Emacs user-driven wrt many classes of users, which contributes to its widespread success.

Program Development Systems

That a text editor is interactive does not mean that a program development system composed of a general-purpose editor and a compiler is user-driven wrt its users. It means only that there is an interactive component in the system; the editor and the compiler are still separate components.

There are two general approaches to reducing this separation. In the first approach, special editor commands have been written to more closely integrate the compiler into the editor. For example, many editors have commands that apply a compiler to a file that is currently being edited. The other approach is that of syntax-directed editing environments, where the editor and the compiler are integrated directly.

Both these approaches relieve the user of exiting or suspending the editor explicitly, which reduces the length of the action path of the solution. Further, errors in the program can often be displayed in the context of the editor, which facilitates the modification of the erroneous program parts. These improved program development systems, however, can no longer be thought of as a simple composition of an editor and a compiler. If a user wishes to use another editor or compiler, for instance, the benefits of integration would require further development.

The Cedar environment is another example of a program development system [16]. One of the prime goals of the system was to permit turnaround of small program modifications in five seconds or less [2]. Although this goal was not met, the system is still thought to be successful. One of the reasons is that many of the programmers who use Cedar "...were not aware of this loss of productivity—they had never had the opportunity to develop their programs in a truly interactive fashion" [16, p. 103]. The author of the quotation also states that the "...philosophy of ‘go slowly, don’t make mistakes because they are expensive to correct’ seems to carry over into the way the ... user interacts with the system." [16, p. 101].

Actually, the programmers who were supposedly losing productivity were not, due to their extremely low error profile. While few people would argue that long turnaround is preferable to short turnaround, our model could have predicted beforehand that the gain of short turnaround for small changes was not an essential goal, given the user community.

Integrated Versus Toolkit Approaches

Two common approaches to developing software systems are the integrated and the toolkit approaches. In the integrated approach, the user has a single interactive system that supports a wide variety of activities. For example, Gandalf is an integrated software development environment in that it supports version control, project management, and incremental program construction [10]. In the toolkit approach, the user has a lot of independent tools, some batch and some interactive, that can be easily linked together. UNIX is the best-known of the toolkit systems.

Our model predicts the ease with which new activities can be supported in each approach. Suppose a designer wishes to add a program verification tool to a system. Consider first the integrated approach, as characterized by Gandalf. Since the Gandalf is tightly integrated, it is difficult to add the verifier without understanding many of the details of the implementation. For instance, the verifier must be based on the internal program tree representation. Further, it is difficult to retain the interactive properties of the system because the computation required to implement the verifier will probably slow down the incremental compiler, if they are to be integrated. So, adding a new activity to an integrated interactive environment is costly. Consider now the toolkit approach, as characterized by UNIX. The verifier can be produced independently of the other components that are available. The verifier is responsible for parsing and error checking, which permits it to be applied to any program text. While it is easier to write and use the verifier, it is a batch component. So, an interactive system that relies on the verifier will become less interactive. In general, any UNIX activity that is described as a sequence of programs connected by pipes is less interactive than the original (perhaps) interactive components.

Separating Applications from Interfaces

The notion of developing programs by separating programs into packages with well-defined interfaces is extremely useful and important. In interactive systems, the two key packages are the application and the interface. A laudable goal in designing these packages is to retain independence between the packages [12]. The advantage of this approach is that either the interface or the application can be replaced without concern for the other.

The decoupling of the application and the interface, however, must be done carefully, as it imposes a level of protocols between the component and the command implementation. To permit arbitrary mappings between commands and implementations may not be desirable. The close connection between the command decomposition and the error profile of the user place requirements on the acceptable costs of the command implementations. Our model indicates that the system structure and the error profile should drive the implementation, rather than the other way around.

Rapid Prototyping

The current emphasis on rapid prototyping also depends on the separating of the applications from the interfaces [18]. In this approach, an interface is developed and is tested using a simple application. Usually, stubs are used wherever procedures should occur in the final application. While this approach is excellent for testing interfaces, it should not be taken as a practical way to develop the final interactive application. The cost of the implementation is closely connected to the success of an interactive system and should not be forgotten during design.

At one level, our worries about the separation of applications
from interfaces and about rapid prototyping are inconsistent. To the designers of the applications and interfaces we warn that the application cannot be built without considering the system structure. To the rapid prototypers we warn that the system structure cannot be built without considering the application's implementation. In a nutshell, to successfully create an interactive system, one must take into account all levels of the problem to be solved, including the user.

VII. CONCLUSION

Our contributions include:
- Distinguishing the difference in structure between interactive and batch systems. In particular, we have shown the distinction between component decomposition—breaking a problem into several subproblems, each of which is solved by a separate software subsystem—and command decomposition—providing a set of commands that a user can interactively select to solve a subproblem.
- Demonstrating that interactivity is not an inherent characteristic of a system. Rather, given a system intended to solve a problem, the degree of interaction must be considered with respect to the user of the system. Specifically, the error profile of the user in combination with the system structure affects the interaction.
- Supporting qualitative analysis of the interactivity of a system in terms of the problem to be solved, the system structure supplied, and the user error profile.
- Describing implementation requirements for constructing interactive systems. Incremental computation and integration of the system are the primary characteristics that are needed.

REFERENCES