The Programming Walkthrough: A Structured Method for Assessing the Writability of Programming Languages

Brigham Bell, Wayne Citrin, Clayton Lewis, John Rieman, Robert Weaver,
Nick Wilde, Benjamin Zorn

University of Colorado
Boulder, Colorado

→ Spreadsheet as good example of latent demand for programming when presented in right way.

Corresponding Author: Clayton Lewis
Department of Computer Science,
Campus Box 430
University of Colorado
Boulder, CO 80302
(303) 492-6657, clayton@cs.colorado.edu

DRAFT: please do not cite.
Comments are welcome.

Abstract
We describe a method for assessing the writability of a programming language by examining the steps required in creating sample programs and weighing the knowledge required to choose among alternative steps. We present experience in applying the method in four language design projects, and argue that the method is a useful supplement to existing approaches.

1. Introduction

One objective of language design is to make programs easy to write; that is, a language should have good *writability*. Part of a language designer's task, therefore, is to understand how the character of a language influences the time, effort, and knowledge required to write programs.

The cost of writing programs is of course only one of the costs involved in the larger programming enterprise. In some environments the costs of maintaining programs far outweighs the costs of initial development, so that the effects of language design on readability and related attributes may dominate some designers' thinking (see, for example Ichbiah et al, 1979; Ada, 1984). Nevertheless, writability has been and continues to be an important design concern.

We present a structured method for exploring the writability of programming languages during the design process. First, we summarize existing approaches to writability. Then we describe our method, the *programming walkthrough*. We then discuss our experiences in applying the programming walkthrough in four language design projects. Finally we weigh the costs and benefits of the method as a supplement to current approaches.

1.1. Current Approaches to Designing for Writability

Attention to writability in design has more often been implicit than explicit. Nevertheless, a number of general approaches to the problem can be distinguished in reports of design projects.

Adoption of proven representations. One way to make programs easier to write is to incorporate in a language existing representations of proven effectiveness. Thus the FORTRAN designers adapted traditional algebraic formulae as a key part of their design (Backus, 1981), a choice that has been followed in most succeeding designs. For problems for which appropriate mathematical formulations are available, a significant part of the task of writing a FORTRAN program is greatly simplified. Recently Citrin (1991a, 1991b) has called explicitly for the use of this approach to guide the design of visual programming languages: such languages, he argues, should exploit existing graphical representations in specific problem domains.

Intuition guided by experience with an evolving sequence of languages. At least since Von Neumann it has been recognized that programming requires facilities beyond those provided by classical mathematics. More generally it is true that writing programs for any application domain requires concepts not in that domain, and so designers must provide features that cannot simply be borrowed from mathematics or elsewhere. Faced with choices about constructs for iteration, for example, where can a designer find guidance? One source is experience with previous language designs. The work of a designer who has produced many designs, like Niklaus Wirth, shows this approach in very refined form. Each successive Wirth design shows the impact of deep reflection on its predecessors, with features being added, removed, or adjusted in reaction to emergent difficulties or opportunities to achieve more with less (Wirth, 1971, 1984, 1985, 1989).

Intuition about the role of abstract characteristics of designs in making it easy to write programs. Experience with the progression of designs that have appeared over the years may be codified in the form of abstract claims about what is good and bad in design. Designers and commentators commonly believe that simple designs are to be preferred to complex ones (Feuer & Gehani, 1984; Hoare, 1981; Pratt, 1984). Orthogonality in a design, in which the behavior of combinations of features can be readily predicted from knowledge of the features and general principles of combination, is also seen as beneficial (van Wijngaarden, 1969).

Because writability often remains an implicit goal, it is not always clear how these or other abstract characteristics are meant to relate to writability, as opposed to other aspects of a design. But some writers are explicit about this: Sebesta (1989) includes simplicity and orthogonality in a list of characteristics supporting writability, and MacLennan (1987) sketches a psychological argument that simple, orthogonal designs are easier for programmers to remember than others.

Attention to the process of writing programs. Increasing writability by tuning a language design to support the process of writing programs is a natural idea. The designers of BASIC took this approach: they analyzed the ideas needed to write and run programs in existing languages and looked for opportunities to eliminate ideas by simplifying the process (Kurtz, 1981). Examples of such design initiatives were the elimination of the distinction between integer and non-integer variables and the elimination of the concepts of compilation and object code. As these examples show, these designers took a comprehensive view of the programming process and worked to simplify all of it.

A rather different approach, still under the heading of attention to process, can be seen in the work of Dijkstra and other proponents of programming by successive refinement (Dijkstra, 1976; Gries, 1981; Wirth, 1971). Here mental processes in programming are discussed quite explicitly: programming consists of the construction of a succession of problem representations linking an original problem statement with a program. While much of the analysis of this process has not been directed at the design of languages to support it, some has been. For example, Dijkstra's *alternative* command has no *else* clause, a deliberate omission that is intended to compel a complete analysis of alternatives during the development of a program (Gries, 1981).

1.2. Refining the Focus on Process

The programming walkthrough method that we describe in this paper also belongs in the class of approaches that focus on the programming process. It can be seen as a structured version of what the BASIC designers did, providing an organized way to develop an inventory of the ideas required in writing programs with alternative language designs. It differs from the successive refinement approach in two ways. First, it is more permissive. The method requires no commitment to any particular programming process, and could be applied to successive refinement or any other process a designer might favor. Second, the approach pushes further (but still not very far) into psychology. Not only the major stages in the development of a program, but also the knowledge required to make choices in development, are examined explicitly. We argue that analysis at this lower level exposes differences between language design alternatives that can provide useful guidance in design.

This explicit analysis of mental processes also distinguishes the programming walkthrough from the approaches to writability placed in other categories above. In all of these approaches, mental operations figure only implicitly, but (we argue) an explicit description of mental processes provides a useful perspective on intuitions that otherwise remain rather vague. For example, it encourages one to think about just how simplicity might support writability, and hence provides a basis for determining what kinds of simplicity are valuable and which are not.

Adapted from cognitive walkthroughs.

2. The Programming Walkthrough Method

2.1. The process of writing a program.

Any design approach that focuses on the programming process needs an organizing view of that process. We presume that writing a program consists of a series of *steps*, mostly mental but some involving physical actions like typing or reading. We presume that the steps that are taken are drawn from a collection of possibilities, in general, so that steps must usually be *chosen* on some basis. We presume that choices are guided by *knowledge* of many different kinds, including knowledge of the problem domain, knowledge of problem-solving techniques, and knowledge of specific language features.

This picture is a simplified summary of the findings of a number of psychological studies of programming: the work of Mayer and colleagues on BASIC (Mayer, 1981; Bayman & Mayer, 1983), Soloway and colleagues on PASCAL (Soloway, 1986; Soloway & Ehrlich, 1984; Spohrer, Soloway, & Pope, 1985; Rist, 1986), and Anderson and colleagues on LISP (Anderson, Farrell, & Sauers, 1984; Anderson & Jeffries, 1985; Anderson & Skwarecki, 1986). All of these studies bring out the role of knowledge of various kinds in guiding the programming process, and especially knowledge that goes beyond a specification of the syntax and semantics of a language. For example, Soloway and colleagues document the existence of many specific programming *plans*,- ways of deploying the features of Pascal for particular purposes, such as the use of an accumulator variable in summing an array. A student who knows Pascal, but does not know these plans, has a difficult time writing programs. Similarly Anderson and colleagues specify what one needs to know to write recursive programs in Lisp: knowing the syntax and semantics of Lisp is only a part of the knowledge required. Ideas about the stereotypical structure of recursive programs, and how to identify and specify base cases and recursive cases are also needed.

Within this broad analysis, how can writability be assessed for a language design? The key steps in our approach, the programming walkthrough method, are (1) to describe the steps required in writing a program for one or more problems, and (2) to describe the knowledge needed to select these steps, and then (3) to examine these descriptions of the programming process with the following questions in mind:

- How long is the process? Are there opportunities to eliminate steps by changing the design?
- Are there steps for which it was not possible to describe knowledge that would guide their selection? These steps will require extensive problem-solving by programmers.
- Are there steps that require knowledge that programmers are unlikely to have, where this knowledge is extensive or involves difficult concepts? Such steps will be hard unless extensive training or indoctrination is possible.

2.2. Example of a Walkthrough

2.2.1 The CMPL Language

To make the preceding explanation more concrete, we describe how the walkthrough was applied in evaluating the macro language CMPL. CMPL, the Core Macro Processing Language (Maurich & Zorn, in progress), is intended to provide all the functionality found in cpp or M4, plus two additional features. As originally designed the language provided a typed macro facility, which would allow the programmer to define syntax extensions to the language at compile time. The parameters and context of macro calls would be checked for syntactic correctness by the macro processor. The language also allows internal macro variables and computation while processing macros.

After some initial design work CMPL was evaluated in several programming walkthrough sessions. As a result of that evaluation, substantial changes were made in the language design. We present part of one of those sessions to illustrate how the programming walkthrough is done.

First, we provide enough of the description of CMPL to make our example understandable. Macro definitions in CMPL have the following basic structure:

```
define ResultDeclaration InputTemplate
begin
   ComputationSection
result
   ResultSection
end
```

In this definition, ResultDeclaration represents the type declaration of the result of the macro and is either of the form "[]" (if the macro is typeless) or the form "[TypeName]" (if the macro is typed). In CMPL, most types are syntax classes of the base language, such as "<Expression>" or "<Statement>". These types must be delimited with angle brackets. There are, in addition, several types provided by CMPL itself, such as "int"; these do not have angle brackets around them.

The InputTemplate part of the definition describes the format of a macro call, consisting of literals (which much be matched exactly in the macro call) and argument declarations. Argument declarations are delimited by square brackets and are composed of CMPL keywords, which provide special functionality, type names, and argument names. Keywords and type names must be followed by colons. A simple argument declaration might be "[<exp>: start]". A typeless version of the same argument would be "[start]". An example of a keyword would be "rest", the CMPL keyword for a list. An example of the use of this keyword is:

```
[rest: delimit "$[ ], $[ ]": int: input]
```

This declares the argument "input" to be a list of type "int" delimited by commas. The "$[ ]" indicates optional whitespace permitted in the list.

The ComputationSection of the definition contains internal CMPL variable declarations and computations. We will not describe this part further since we do not use it in the following example.

Finally, the ResultSection is the template for the text string returned by the macro execution. Since it is a string, it must be enclosed in quotes. It consists of literals and CMPL arguments and variables. Arguments and variables are prepended by a "$" and delimited by square brackets. A simple example is ""$[inputA] + $[inputB]"". The type of a variable or argument can be changed (similar to a cast in C) by prepending the token for the variable or argument with "([Type])", where "type" is either a TypeName, a CMPL keyword that affects the type (such as "`rest:`"), or a legal combination of these.

2.2.2. A Walkthrough for CMPL

The programming walkthrough sessions for CMPL were done by four of us with no experience in CMPL and by the designers of the language. The designers provided a written description of CMPL and a suite of 23 sample problems. While the designers watched, the four of us then worked through the first few problems a step at a time, being careful to identify the reason for each step we took.

We worked through three problems in two sessions, each about two hours long. During the walkthroughs, the designers provided help or explanations when asked, but didn't volunteer assistance. After the second session the designers called a halt to the evaluation and went back to modify CMPL based on what they had learned.

Here is an abbreviated description of the session in which we analyzed the first problem. For space reasons we give only an outline of what took place. We have omitted all the false starts, all the mistakes, and almost all the lengthy discussions we engaged in.

The problem to be solved with CMPL was this: write a macro that takes arbitrarily many integer arguments and produces a sum expression. For this walkthrough the designers did not provide a preferred solution, though they had one in mind.

After considerable discussion on how to start, we agreed on a basic explanation for what a macro is, and we set out a three-part definition of a macro definition. These became the first two items in our list of knowledge needed to write CMPL macros:

I. A macro is an input to output translation in which the input must satisfy certain pre-defined conditions.

II. A macro definition has three parts:
   A. A description of the decomposition of the input into parts (some of which can be named),
   B. Any computation necessary to describe the output, and
   C. A description of the output.

This knowledge provides a skeleton for a macro to be filled in, and we identified establishing this skeleton as the first step in the programming process. The next step seemed to be filling in the first part of the skeleton, but here we confronted a choice between using a typeless or typed macro, leading to a lengthy discussion of the difference between these. We elected to use a typeless macro because it seemed simpler to us, but we could not formulate a principle that would help us make the right choice. We therefore noted this as a step for which it would be difficult to specify adequate guiding knowledge.

At that point, someone suggested that a reasonable next step was to examine sample input and output, and we added this knowledge to our list:

III. To begin, first write out a sample input and its corresponding output.

Applying this advice we wrote:

```
Input: add(1, 2, 3, 4)
Output: 1 + 2 + 3 + 4
```

Since we were concentrating on input and output and the relationship between them, we tried to categorize the possible relationships. This resulted in another item of knowledge for our list:

IV. Then, look at the relationship between your sample input and sample output. Try to fit it into one of the following:
   A. No relationship.
   B. Simple rearrangement (plus fewer or additional literals).
   C. (B) plus simple substitution.
   D. More complicated than the earlier three.

We then wrote out a principle for mapping the input to the first part of a macro definition:

V. Describe the input so that the pieces determined in (IV.) have names (or are otherwise accessible) and the pre-defined conditions on the input can be checked.
   A. Within this limit, keep the input description as simple as possible.

It became apparent to us that the only CMPL construct capable of matching an arbitrary sequence of integers was a list. Our input began to take shape in the next step, guided by this principle:

```
add( [rest: delimit "$[ ], $[ ]": input])
```

This describes the macro call as the keyword "add" followed by a list (for which CMPL uses the term "rest:"), delimited by commas and allowing optional white space around the commas, to be called "input". Some of us objected that "rest" seemed a poor name here, and we added the following to our knowledge inventory to flag this use as one programmers would need to know about:

VI. The CMPL keyword "rest" matches a list.
   A. The CMPL keyword "delimit" allows you to specify the delimiter that separates the elements of that list.

There was then a lengthy discussion over what type of input type checking we wanted to do. This gave rise to four more items of guiding knowledge:

VII. Here are the various types of pre-defined conditions that can be checked on the input:
   A. Types of arguments.
   B. One level of list form checking.

VIII. More complicated checks than those specified in (VI.) can be specified in the computation section.

IX. For each argument in the input, you can specify a type that it must match.
   A. You do not have to specify a type.

X. As a rule of thumb, the tighter the type checking you can specify on the input, the better.

On this basis, we decided we would not only check the input for the proper delimiters (already done above) but we would also test to see that the elements of the list were integers. At the next step our input became:

```
add( [rest: delimit "$[ ], $[ ]": int: input])
```

This form differs from the previous in including the type "`int:`", which constrains the elements of the list. At this point one of us noted that the only difference between the argument list "input" and the required result form was the delimiter: replacing the commas by plus signs would do the job.Since the delimiters are not part of the list value in CMPL, this could be done simply by coercing the type of the list to change the delimiter from a ", " to a " + ". We added another item of knowledge to point out the possibility of this approach.

XI. You can change the type of a list to another list type with a coercion.
   A. Using this technique, you can change what the delimiter is.

Our complete macro definition then became:

```
define[]add( [rest: delimit "$[ ], $[ ]": int: input])
begin
result
   "[([rest: delimit " + ":]) input]"
end
```

Here the result form coerces input to be a list delimited by plus signs. This solution was quite different from what the designers had in mind. They expected that the result form would be constructed by code in the ComputationSection of the macro definition. This code would build a result string by a recursive process that pieced together the arguments in the macro call separated by plus signs. The solution developed in the walkthrough required no such computation, and has an empty Computation Section.

2.2.3 Analyzing the Results from the CMPL Walkthrough

What was learned from this part of the CMPL evaluation? First, the walkthrough produced an overall picture of what a macro is and how one should go about writing one, embodied in a number of specific items of information that could guide users. This material might be used in documenting the language for users, but more immediately it enabled the designers to see the context in which the various features of the language would be employed.

Another result from the walkthrough was that choosing between typed and typeless macros was highlighted as problematic, in that the analysts could not immediately formulate knowledge adequate to guide this choice. Further, the designers were not able to motivate the choice without substantially complicating the overall picture of what a macro is and how it should be written that emerged from the walkthrough. Because this overall picture looked good to the designers, they opted to make important changes to the design that eliminated the typed-typeless distinction while capturing its intended function in another way.

A third issue highlighted by the walkthrough, the difference between the designers' expected solution for the problem and the solution developed in the walkthrough was not seen as a problem. Rather, the new solution was seen as a logical outcome of examining the relationship between the form of a macro call and the form of the result, as called for in item IV on the knowledge list, and noting that only the delimiter in the argument list needed to be changed to produce the desired result. So the designers decided not to try to change the design or add new items of knowledge to push users toward the original expected solution.

Finally, the minor issue of the use of "`rest`" where "`list`" would be more transparent was noted.

2.3. Cookbook Procedure for the Programming Walkthrough

Here is a more complete description of the walkthrough, with some details of the method added.

1. Define the language. The language need not be implemented, but a reasonably complete definition is required, at least of those features to be examined in the walkthrough.

2. Define a suite of sample problems. A walkthrough examines the programming process involved in solving one or more specific problems. These problems should capture what the designer hopes to achieve in the design. Problems that might be used to demonstrate the value of novel language features are good candidates. Problems that are representative of real problems in the application domain intended by the designer should also be examined. If a realistic problem is large it will not be possible to analyze the complete programming process in detail, but crucial parts of it can be picked out for attention. Statements of the problems should be examined to make sure they do not include unrealistic clues to programming methods, for example, references to concepts that are defined in the language but not in the intended application domain. Thus a problem about manipulating lists

would not be appropriate for evaluating a language intended for users who wouldn't be expected to know what lists are.

3. Outline solutions to the problems. The designer should decide what kind of program should result when a programmer tackles each of the sample problems. The value of these target solutions rests on a subtle but important argument. To be useful in design, writability analysis should relate to the designer's *intentions* in the design: simply knowing whether or not the design has good writability is not as useful as knowing whether the features the designer intended to produce writability are working or not, and why they are not working if they are not. Thus the walkthrough analysis should focus on determining whether the design is likely to work as intended, meaning that the path to an intended solution should get special scrutiny.

The designer can decide whether and when to reveal the intended solutions to any other members of the walkthrough team. The designer might choose to withhold solutions until it becomes clear whether the team, working without knowledge of the solution, heads in the expected direction.

4. Convene the walkthrough analysts. A designer working alone can perform a walkthrough, but our experience suggests the value of getting other people to help. Outsiders can help the designer to do a more complete analysis, not skipping over steps that seem (from an inside perspective) uninteresting or obvious.

5. Work through the steps in the programming process. For each sample problem the analysts now try to identify the steps that take the programmer from the problem statement to the target program. Keep in mind that key steps, especially early in the process, may be mental reformulations not tied to writing any code and perhaps not influenced much by any aspect of the language design. The analysts should ask, "What do I have do first?", "What do I do next?", and, crucially, "How do I know that's the right thing to do here?" The answers to these questions are the raw outputs from the walkthrough: analysts may want to make a list of the steps and the knowledge that guides each one. Arguments like "Shouldn't I do it this way instead?", or

"Haven't we skipped over a decision here?", or "So how did I know I wasn't supposed to use this feature... from the definition it looks applicable," are to be encouraged, and the list of steps and motivating knowledge revised and elaborated until the analysts feel they have captured the process.

Note that the analysts are not trying to document their own mental processes in solving the problems. Rather, they are trying to outline the mental processes of a hypothetical user, who might have quite different knowledge. Thus the fact that all the analysts find a step easy does not mean that nothing needs to be said about it. Even easy steps must be examined at least enough to see what knowledge is needed to make them easy.

It may happen that as the analysts work through aproblem it appears that the designer's target program is unlikely to be reached. The analysts can proceed in either or both of two ways. They can push on toward the target, noting the obstacles in the way, in the form of steps that are unlikely to be chosen. This provides the designer with specific information about the difficulties that must be dealt with to realize the original intent of the design. Or, with the designer's consent, they can explore paths leading to other solutions. If one of these proves sufficiently easy, and the resulting program is appropriate, the designer might revise his or her thinking about how the language should be applied, and hence what features should be provided.

6. Analyze the process. Writability problems can now be identified, either incrementally while working through the process or in later discussions after the process has been fully described. As indicated earlier, writability problems can take the form of large numbers of steps required for problems, or necessary steps for which adequate guidance is hard to identify, or steps for which adequate guiding knowledge is extensive or esoteric.

Beyond these problems the analysts should look for at three other aspects of the results. First, are there conflicts among points of guiding knowledge, or or between guiding knowledge and other knowledge users may bring to the situation? Second, are there steps whose correctness is actually unclear? If so the language design must be incomplete or ambiguous. Third, does the guiding knowledge clarify or modify the designer's view of how the language should be used? The designer should determine whether and how the process developed in the walkthrough differs from what he or she would have expected.

The CMPL walkthroughs evaluated a single design for the language. But in other applications, described below, we have used the method to compare alternative designs. Here the analysis focuses on comparing the walkthrough results for the two designs, to decide if one version has fewer or less problematic steps, for example, or if knowledge required in one version is not needed in another.

7. Apply the results. Designers can act on walkthrough results in a number of different ways. They can redesign the language to avoid lengthy sequences of steps or steps that are difficult to guide. They can provide documentation for the language that will include guiding knowledge users would otherwise lack. They can clarify or modify the language definition to avoid ambiguities.

If the walkthrough reveals a new approach to using the language, not anticipated by the designer two responses are possible. First, the designer can accept the new approach and look for opportunities to make the design better support the new approach. Second, the designer can try to push users in the intended direction, and away from the new approach, by changing the design or documentation of the language.

3. Experience with the Method

We have applied the programming walkthrough method to four different languages, including CMPL. We outline the results here to show what can be learned using the method, and also to illustrate how the method can be adapted for use in different design contexts.

3.1. More on CMPL

The CMPL walkthroughs, including the example just discussed produced results that can be assigned to three categories. First, as described earlier, *the walkthrough clarified basic concepts of how the language would be viewed and used.* The preliminary design of CMPL before the walkthrough combined conventional macro processing techniques with extensible language ideas, without considering how these facilities would be used together. The greatest effect of the walkthrough process on the language design was the perspective it placed on the interaction of these two modes of macro expansion. In particular, while the designers saw typeless and typed macros as involving very different concepts and implementation technology, the overall approach to defining macros that emerged from the walkthroughs did not support this distinction.

This mismatch led the designers to adopt a new view of types in the language design. In a typical programming language, types are specified by a set of values and a set of operations on those values. In CMPL, because it is a text transformation language, types have an additional aspect which is their syntactic representation. For example, consider the CMPL type (in the revised design) ``<expression>:string." In all respects, variables of this type can be manipulated as strings (i.e., they can be concatenated, substrings can be taken, etc). However, the constraint is imposed that the string must have the syntactic structure of an expression (i.e., it must be parsable to expression). If a variable of this type is read in or printed out and the structure is not an expression, an error is generated.

To the best of the designers' knowledge, assigning these semantics to types in CMPL is a unique approach to defining a macro-processing language and represents a large step toward making the use of macro expansion more structured and less error-prone. In the resulting semantics, types with associated syntax classes can be used anywhere types can, and syntax checking goes on at macro expansion time without the need for language extension though added base-language grammar rules.

It is clear that this design change resulted from the different perspectives of the analysts performing the walkthrough and the designers of the language. By looking at language features purely as tools to solve problems, the analysts provided a perspective that had escaped the language designers themselves.

A second category of result from the walkthroughs was the identification of *minor flaws in existing features,* such as the use of "rest" rather than "list". The designers felt that these flaws would have been difficult for them to discover without the walkthrough, because they were (obviously) fully aware of what was intended in all such cases.

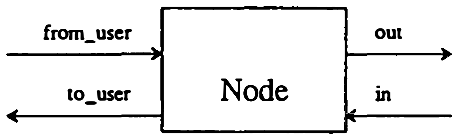
Finally, the CMPL walkthrough showed evidence of *features that needed to be added to* the language, such as a string manipulation operation to perform substitution. While these were useful results, the designers believed that the same features would have been added in the normal course of the language's development.

The designers noted that the results from the walkthrough were especially valuable because they were available very early in the design process. Had the evaluation been done later it would have been painful to contemplate major revisions to the design such as some of the results called for. As it was, the results could be incorporated in what was still the early conceptual phase of developing the design.

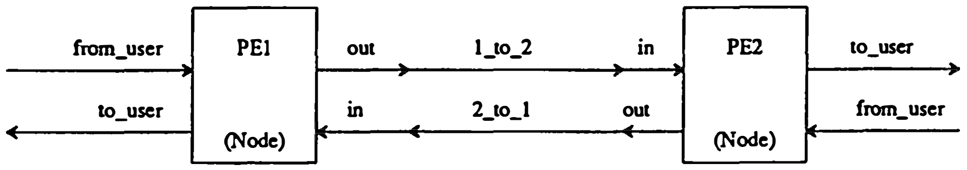
3.2. A Visual Language for Communication Protocols: MFD

MFD (named for Message Flow Diagrams) is a visual language for specifying communication protocols. As the name suggests, it lets users base a specification on message flow diagrams of a style common and familiar in the networking literature. The user describes a protocol by creating message flow diagrams for a set of scenarios that capture the behavior of the protocol. Events specified in the diagrams are identified, and rules capturing their conditions of occurrence are automatically created. These rules form an executable specification of the protocol.

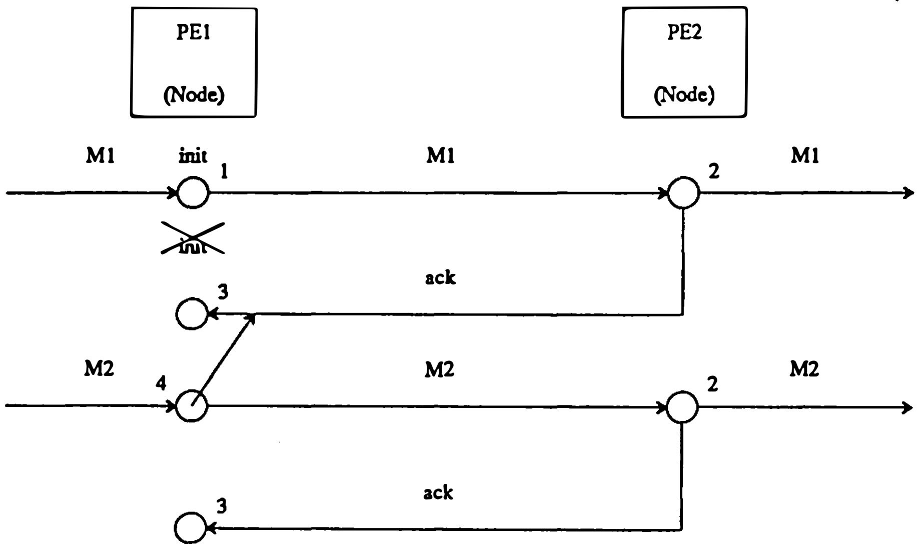
Figure 1 shows an MFD specification for a simple stop-and-wait protocol. The heart of the specification is the message flow diagram near the bottom of the figure. The diagrams above serve to describe the nodes that appear in the



PE Type Declaration



Prototype Configuration Diagram



Message-Flow Diagram

1) init, rcv(from_user,M1) -> send(out,M1), not(init).

2) rcv(in,M1) -> send(out,ack), send(to_user,M1).

3) rcv(in,ack) -> true.

4) rcvd(in,ack,-1), rcv(from_user,M2)-> send(out,M2)

Figure 1 - MFD version of a stop-and-wait protocol, with the resulting rules of behavior

message flow diagram and how they are connected.

Circles in the message flow diagram show events in a scenario, ranged in columns beneath the nodes that perform them. Time runs down the page, with message exchange shown by labelled horizontal arrows. The word **init** denotes a piece of state information for the node in whose column it appears; its presence means that the **init** condition must hold for the event below it to occur. The crossed-out **init** below this event indicates that the **init** condition no longer holds after the event occurs. The upward pointing arrow connecting event 4 to the message entering event 3 denotes a history condition: event 4 cannot occur unless that message has been received.

The protocol behavior can now be read off the diagram. If node PE1 is in the **init** state, and it receives a message from its user, it sends the message to PE2 and leaves the **init** state. If node PE2 receives a message from PE1 it passes it on to its user and sends an **ack** message to PE1. PE1 does nothing when it receives an **ack**, but the event shown as 4 cannot occur unless an **ack** has just been received. That is, if PE1 receives a message from its user and it has just received an **ack** it passes the message to PE2. This behavior is described in the rules shown at the bottom of the figure, which MFD would generate.

The original design of MFD was closely based on the message-flow diagrams displayed by the Cara programming environment (Cockburn, Citrin, Hauser, & von Kaenel, 1990), which in turn was based on a study of documentation and interviews with experienced protocol designers. The Cara diagrams are not a programming language, and their meaning is ambiguous. MFD resolves these ambiguities with graphical features where possible, otherwise with textual annotations.

MFD was subjected to a walkthrough after the language manual had been completed, but before the language was implemented. The problems used were to specify two simple protocols, the alternating bit protocol and a sliding window protocol (Tanenbaum, 1981), from textual descriptions. Four of us, none of whom are experienced protocol designers, read the MFD documentation and jointly attempted to specify the protocols. The designer did not provide MFD solutions. The language designer (Citrin) was present, but did not intervene unless requested by the others.

There were several major points of difficulty revealed by the walkthrough. As happened with CMPL, *the walkthrough clarified basic concepts of how the language would be viewed and used*. In particular, it emerged that users have a choice between using the history mechanism to indicate conditions for events or creating and using state information like that represented by **init** in the example. While the designer intended that history should be used, there was no clear motivation to do so, and the analysts opted not to. The designer must now develop effective motivation for the use of history or must accept that the language may not be used as intended.

The walkthroughs also revealed some *ambiguities in the design*. When a history arrow was drawn it was not clear how much of the context pointed to would be included in the condition specified by the arrow. In the example, is the condition expressed by the history arrow from node 4 simply that an ack be received, or must it be received by an event satisfying any conditions on event 3, since event 3 is the receiving event shown? The designer's intent was that only the identity of the message, and not any attributes of the event, be included in the condition, and this has now been clarified in the language definition. Other ambiguities arose with the meanings of a kind of annotation not shown in the example. The design was changed to eliminate one of the ambiguous annotations and clarify another so as to fit the interpretation that emerged as most natural from the walkthrough.

Finally, the walkthrough revealed the opportunity to *simplify* the way complex messages were defined and referenced. While the mechanisms in the original design were workable, the walkthrough showed that they were complex and required the creation of unnecessary notation by the user. Improved solutions have been incorporated in a revised design.

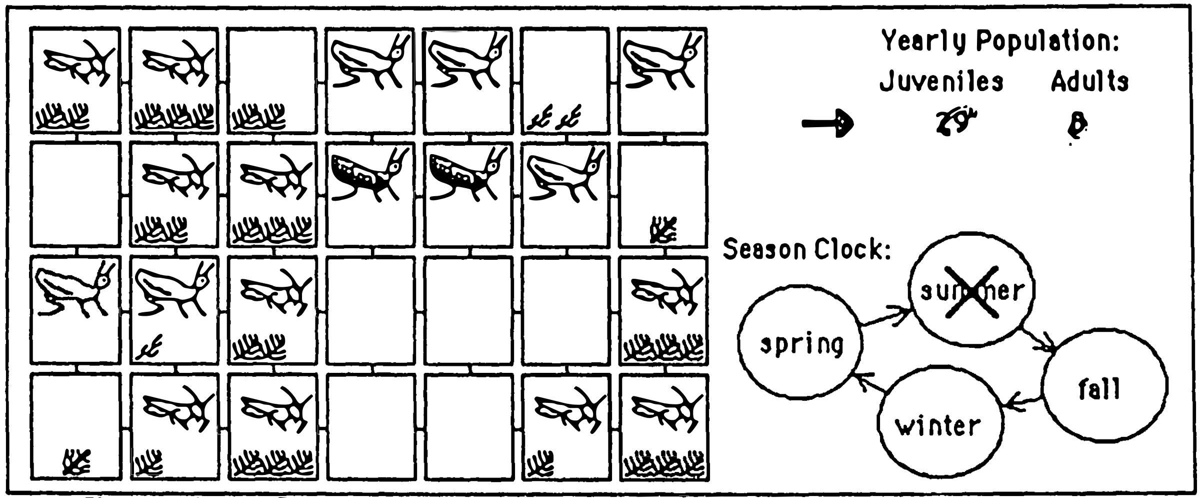
3.3. A Language for Graphical Simulations: ChemTrains

The walkthrough evaluations of CMPL and MFD occurred early in design as "one shot" looks at the state of the designs. In contrast, walkthroughs have been incorporated as an integral part of the ongoing development of the ChemTrains language. The ChemTrains walkthroughs also differ from those already discussed in that they were done by the designers (three of us, Bell, Lewis, and Rieman) with no outside participation.

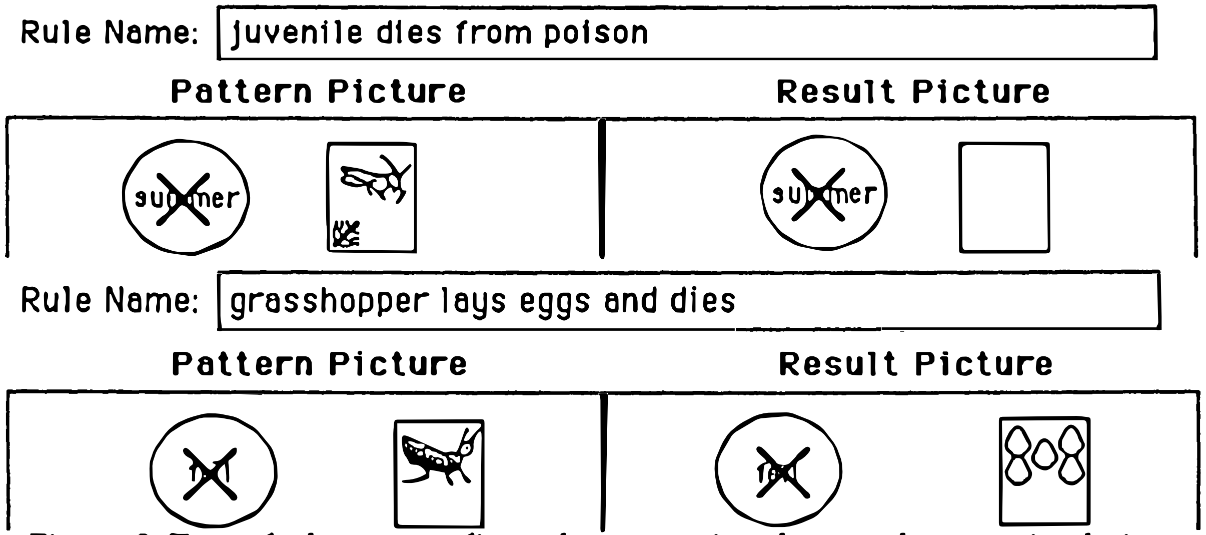
The ChemTrains language provides nonprogrammers with a simple but powerful interactive visual programming environment for modelling qualitatively defined systems, such as a Turing machine or document flow in an organization (Lewis, Rieman, & Bell, 1991). To support users with little or no training, the system is intentionally simple. Every simulation is represented using only three classes of graphical entities: objects, containers, and paths. Behavior of these entities is controlled by "if-then" production rules, also described graphically by the user. The language similar to BitPict (Furness, 1991), another rule-based visual language, except that patterns in ChemTrains contain objects and relationships among them rather than pixel configurations as in BitPict.

Figure 2 shows part of a ChemTrains program for simulating grasshopper population dynamics. The upper portion of the figure shows a snapshot of the simulation, while the lower portion shows two rules. The first rule fires when it is summer and a juvenile grasshopper encounters a poisoned food plant; when the rule fires the grasshopper and the plant are deleted. The second rule makes adult females lay eggs and die in the fall.

After a period of unstructured development and implementation of a simple prototype, the designers made a concerted effort to define a set of language features that would be both powerful and easy to learn and use. These deliberations, which were focussed on six specific "target problems" that the language should support, yielded three alternative designs for the language, representing three different design philosophies. The "ZeroTrains" design opted for simplicity, avoiding special features when combinations of primitive features could provide the same result; "OPSTrains" chose features



A ChemTrains simulation display of Grasshoppers in a field.


Rule Name: juvenile dies from poison

Pattern Picture

Result Picture

Rule Name: grasshopper lays eggs and dies

Pattern Picture

Result Picture

Figure 2: Two of the twenty-five rules governing the grasshopper simulation.

that provided power, influenced by experience with the OPS family of rule-based systems (Forgy, 1984); and "ShowTrains" aimed for concreteness, allowing the user to specify rules by demonstration.

3.3.1 The Initial ChemTrains Walkthroughs

Programming walkthroughs were performed on each of the six target problems for each of the three competing designs. Each of the three designers working on the project acted as analyst for a single design, developing the list of required knowledge and writing up the walkthroughs.

Examining the required knowledge and the problem-solving recorded in the walkthroughs for the six target problems revealed a consistent pattern of differences among the three designs, clearly attributable to particular design decisions. The comparison was critically influential in resolving the designer's differences of intuition as to which was the best design. Overall, the power-oriented OPSTrains design was found to be easiest to write programs in. ShowTrains placed second, while ZeroTrains placed last, with each solution requiring work directly attributable to getting around ZeroTrains' "simplifying" assumptions.

The walkthroughs demonstrated several things that intuitive arguments had left unresolved. Most strikingly, *the walkthroughs contradicted the designers' intuitive belief that a simple language would be easy to program.* On the contrary, the simplicity of ZeroTrains made it harder to program. Conversely, the greater complexity of OPSTrains was shown to make it more useable. The additional features in OPSTrains directly matched the problem-solving needs of the programmer, while the simple building blocks of ZeroTrains required both excessive knowledge of techniques and creative problem solving to reach a solution.

Another unexpected result was the effect of variables, a power-oriented feature unique to OPSTrains. The designers were not surprised that variables permitted more economical solutions. But they had been expected to exact a price, recognizable as extensive additional knowledge required to use them.

In fact, this was not the case. Adequate knowledge to guide OPSTrains programmers simply suggests first writing a rule with no variables, then marking some constants as variables to make the rule more general. This is the same idea behind the successful Query-by Example design (Zloof, 1975). The walkthroughs indicated that *a little bit of knowledge could make an apparently difficult language feature easy to use.*

3.3.2. Validating the Walkthroughs

Although the walkthroughs and the design process had focussed on six target problems, the designers had attempted to create a language that was generally applicable. To test whether this had been achieved, the designers did another set of walkthroughs for each design, using four new problems that fell within the language's general design goals.

The second set of walkthroughs confirmed the central result of the first analysis: the OPSTrains design was the easiest to use. But several of the new problems were more complex than the first set, requiring features that had not been considered in the design or the initial walkthroughs, and all three designs had difficulties that the initial walkthroughs had not predicted. None of the designs, for example, gave precise control over timing and synchronicity of rule firings. Similarly, none allowed objects to be rotated. As with CMPL and MFD, these walkthroughs again showed evidence of features that needed to be added to the language.

As a final validation of the walkthrough process, a prototype of the most successful design, OPSTrains, was implemented and subjected to user testing. The techniques of the language were described to the users, and they were asked to create an animated simulation showing the phase changes of material in a beaker over an adjustable Bunsen burner.

Like the second set of walkthroughs, the user testing generally confirmed the walkthrough analysis, but also pointed up some shortcomings. First, the inventory of required knowledge was incomplete in a few places, especially with regard to very basic language issues that the designers had long been familiar with. Second, our cursory description of the required knowledge was sometimes inadequate to overcome users' expectations about how they should proceed. Third, simply describing to the user a technique for accomplishing some result was not always convincing; some users balked at following techniques without some deeper understanding of their effects. (The walkthroughs are described in more detail in Rieman, Bell, and Lewis, 1991; the user testing is covered in Bell, Rieman, and Lewis, 1991.)

3.3.3. Iterative Design with Feedback from Walkthroughs

One of us (Bell) has continued with another round of design work on ChemTrains to address limitations in the previous version. In particular, Bell set out to add support for numerical computation and modularization of large programs. In this round of design the programming walkthrough was integrated into the design process, rather than being used to compare or evaluate relatively finished designs. The process can be likened to iterative design with feedback from rapid prototyping (Buxton and Shneiderman, 1980) with prototyping replaced by walkthrough evaluation.

This process allowed the designer to search the ChemTrains design space by examining either simple or drastic design changes at each iteration. Since the cost of generating and evaluating new designs was small, backtracking was inexpensive.

Walkthrough analysis at each stage not only provided feedback on the alternatives that were evaluated but also suggested many new alternatives. An inquiry that began with 35 identified design choices expanded over three months of work to include 95, with most of the additional alternatives emerging directlty from the walkthrough analysis. A representation of the design space as a list of yes/no questions enabled the designer to scan design alternatives quickly, to define new designs precisely, and compare designs easily.

At the outset it was clear that doing repeated walkthroughs with a suite of some twenty problems would be tedious. The designer lightened the work by analysing only the problems most clearly affected by a design change, and focussing individual walkthroughs on just parts of solutions that were affected. This narrow focus had the side benefit of permitting the designer to detect very subtle differences in writability between alternatives being examined on the same small part of a larger problem.

3.4 . A Language for Parallel Numerical Computing: DINO

DINO is a language for programming distributed memory parallel computers, designed primarily for doing regular numerical problems in a data parallel fashion. The language was developed by a team including one of the present authors; see Rosing, Schnabel and Weaver (1991). DINO's design reflects the philosophy that the programmer must say how a problem is to be parallelized. To this end, it attempts to provide the programmer with high-level constructs for distributing data to processors and specifying inter-processor communication.

Like many new languages, DINO evolved through the efforts of a small group of developers who were also its primary users. It was incrementally developed over about three years, first as a C++ application and later as a language with its own compiler to provide better performance. Usability decisions primarily reflected the developers' subjective experiences. The language design was well established at the time the walkthroughs were performed.

The DINO walkthroughs were aimed at specifically exploring two alternative language constructs. In the current DINO, interprocessor communication is designated by appending a "#" to a reference to a variable that has been distributed across processors. If the reference occurs in a write context, the communication will be a send; if the reference occurs in a read context, the communication will be a receive. In either case, the affected variable will be automatically updated. The walkthroughs were intended to explore an alternative syntax ("Send(X)" and "Recv(X)") that explicitly separated communication of a variable from reads and writes to local memory.

The walkthroughs were performed by two of the authors (Weaver & Lewis, 1990). We began the process by informally developing a first approximation of the items of knowledge needed for general use of the language. We then selected a suite of problems designed to highlight the difference between the two alternative ways of specifying inter-processor communication. Walkthroughs were conducted for each problem with each set of constructs. The entire process was relatively informal and took two afternoons.

The walkthrough analysis yielded several results. First, it indicated that more guiding knowledge was needed for the "#" alternative than for the Send/Recv alternative. This suggests that this alternative would be harder to learn and use. The designers could respond to this finding by adopting the Send/Recv approach, or by seeking ways to make the "#" alternative work better. Much of the complexity of the "#" approach comes from the need to force sends and receives at points where there are no corresponding writes or reads, a situation that requires several specific techniques. The designers could identify what type of problem leads to these situations, then propose other language features that deal with them without the need for these techniques. Thus the walkthrough analysis can help in *developing design alternatives* as well as choosing among them.

The walkthrough results raised another interesting point. While the "#" needs more techniques, it also sometimes *needs less code.* A final choice of design would have to weigh this fact as well as the walkthrough result. Implementation issues would have to be considered as well. The walkthrough analysis did not deliver an ultimate verdict on design alternatives, but it permitted a more fully informed decision to be made.

The walkthrough also turned up information that was not sought. DINO programs normally manipulate arrays in such a way that processing of elements in the interior of an array differs from processing near the edge. We found we needed guiding knowedge to direct the programmer to think first about the processing of interior elements and then deal with the details of edge conditions. It became clear from the examples that the language provided good high-level support for dealing with interior elements, but it provided little help in coping with the edges. Though this issue was not related to the specific comparison we were pursuing it did point up an area of the overall design that might repay attention.

For DINO, in which most of the major design decisions had been finalized before the walkthroughs were performed, the most influential effect of the walkthrough analysis was to produce *an inventory of what users really need to know to use the language.* The documentation prepared for the language included not only the usual definitions of language constructs but also the guiding knowledge we identified in the walkthrough, in explicit form. It also included walkthroughs of two sample problems, to help users understand how the guiding knowledge can be used to solve problems.

4. Discussion

4.1. Potential Yields of the Method

These case studies show that the programming walkthrough can be performed as soon as the definition of a language is available, and it can yield useful information at that time as well as at points much later in the design process. The information produced can include:

- Places in the programming process where many steps are required,
- Choices for which adequate guidance is not available,
- Choices that require extensive or esoteric knowledge,
- Language features whose definitions are ambiguous, and
- A clarified or modified view of how the language might be used.

Designers can apply this information by

- Redesigning the language to provide simpler solutions,
- Redesigning the language to avoid problematic choices,
- Documenting the knowledge needed to use the language,
- Clarifying the definitions of language features,
- Redesigning the language to support a different approach to using it, or

• Choosing between design alternatives by comparing walkthrough results.

A key point supported by the case studies is that programming walkthroughs go beyond the designer's intuition in evaluating language writability. The CMPL, MFD, and ChemTrains walkthroughs all gave results contradicting the designers' intuitive predictions about their languages. This analytical power is a primary advantage of the programming walkthrough over existing approaches to the design of writable languages. It derives from the fact that reasoning about general principles of language use is weaker than reasoning about how features of a language might be used in solving specific, concrete problems, as required by the walkthrough method.

It is possible that the analytical power of the method could be developed further by pressing the walkthrough to break down the programming process into smaller steps than those we have settled on. The work of Green, Bellamy and Parker (1987) in studying the fine details of coding, including the order in which statements are written, and the way in which partial descriptions of code are refined, suggests that insights into language features are available at this level of analysis as well as at the coarser level of analysis we have used.

Given the role of concrete problems in the walkthrough method, it is interesting to contrast the walkthrough with other evaluation techniques that also use concrete problems. Suppose a designer simply works through a series of sample problems, verifying that they can be solved and examining the solutions? This differs from the walkthrough if the designer does not keep an accounting of the knowledge needed to arrive at a solution. Much of the benefit of the programming walkthrough comes from examining that knowledge.

Suppose a designer arranges for test users to solve sample problems, collecting thinking-aloud protocols (as in Bell et al, 1991) as a way of identifying mental steps? The limitation here is that users will only comment on difficulties they themselves have, so only some of the steps in the programming process, and some of the knowledge needed to guide it, will be revealed. Further, users will not try to formulate clear statements of guiding knowledge. Of course if users were directed to discuss all steps, and to enumerate all guiding knowledge, this method becomes a walkthrough.

The use of concrete problems in the walkthrough method produces limitations as well as strengths. The results will only be as good as the problems that are analyzed, and no finite collection of problems can capture all the important considerations in the use of a language. As we noted ealier, for example, the very simple problems used in the initial ChemTrains evaluation caused us to miss key issues that would arise in real use. So the walkthrough must be seen as a supplement to other methods, including those based directly on designers' intuitions, that can assess the importance of language characteristics not exercised in any particular sample problem.

Of course the walkthrough is not a substitute for intuition anyway, since intution is constantly in play in the method, in identifying steps in the programming process and knowledge adequate to guide them. The claimed advantage of the method is not that it replaces intution but that it guides the application of intuition in a productive way.

4.2. Costs versus Benefits

We believe that all the walkthroughs we have described paid back the time and effort invested in them. However, balancing that time and effort against the results achieved suggests that walkthroughs are most effective when performed fairly early in the design cycle, with relatively small (though not necessarily simple) problems.

The DINO analysis came too late in the design cycle to have a major effect on the design, although it did provide important suggestions for the documentation. The ChemTrains walkthroughs that assessed the three different design strategies had a major influence on the language's evolution. But the effort of performing and recording the 30 ChemTrains walkthroughs was considerable, perhaps beyond what many language designers would find acceptable. The walkthroughs for MFD and CMPL showed the best balance between effort and results. Neither of these required more than a day or two of the analysts' work, including one or two walkthrough sessions, writing up the techniques, and follow-up discussions. Yet in each case the walkthrough yielded clear evidence of problems that had escaped the designer's intuition, but that could be fairly easily corrected. Importantly, some of the results led to substantive, not superficial, design revisions in each case.

4.3. Conclusion

The programming walkthrough is a structured method that can refine a designer's intuitions about language writability into a more concrete and focussed understanding. Instead of asking whether a language construct is "natural," the designer can ask what decisions are involved in using the construct and how likely it is that the programmer will possess the knowledge necessary to make the right choices.

References

Ada: past, present, future--An interview with Jean Ichbia, the principal designer of Ada. (1984). *Communications of the ACM 27,* 990-997.

Anderson, J.R., and Jeffries, R. (1985). Novice LISP errors: Undetected losses of information from working memory. *Human-computer Interaction 1,* 107-131.

Anderson, J.R., Farrell, R., and Sauers, R. (1984). Learning to program in LISP. *Cognitive Science 8,* 87-129.

Anderson, J.R., and Skwarecki, E. (1986). The automated tutoring of introductory computer programming. *Communications of the ACM 29,* 842-849.

Backus, J. (1981). The history of FORTRAN I, II, and III. In *History of Programming Languages.* R. L. Wexelblat, Ed., Academic Press, New York.

Bayman, P., and Mayer, R.E. (1983). A diagnosis of beginning programmers' misconceptions of BASIC programming statements. *Communications of the ACM 26*, 677-679.

Bell, B., (in preparation). Using programming walkthroughs to design a visual programming language. Ph.D. dissertation, Department of Computer Science, University of Colorado.

Bell, B., Rieman, J., and Lewis, C. (1991). Usability testing of a graphical programming system: Things we missed in a programming walkthrough. In *Proceedings ACM CHI'91 Conference on Human Factors in Computer Systems* (New Orleans), pp. 7-12.

Buxton, W., and Shniderman, R. (1980). Iteration and the design of the human-computer interface. *Proceedings of the 13th Annual Meeting of the Human Factors Association of Canada*, pp. 72-81.

Citrin, W. (1991a). Visualization-based visual programming. Tech. Rep. CU-CS-535-91, Department of Computer Science, University of Colorado, Boulder, July.1991.

Citrin, W. (1991b). Design considerations for a viaual language for communications architecture specifications. In Proceedings 1991 IEEE Workshop on Visual Languages (Kobe, Japan, Oct. 1991).

Cockburn, A.A.R., Citrin, W., Hauser, R.F., and von Kaenel, J. (1990) An environment for interactive design of communications architectures. In *Proceedings of the 10th International Symposium on Protocol Specification, testing, and Verification* (Ottowa, June 1990).

Dijkstra, E.W. (1976). *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J.

Feuer, A.R., and Gehani, J.H. (1984). A methodology for comparing programming languages. In *Comparing and Assessing Programming Languages: Ada, C, Pascal.* A.R. Feuer and J.H. Gehani, Eds., Prentice-Hall, Englewood Cliffs, N.J.

Forgy, C.L. (1984). OPS5 User's Manual. Tech. Rept. CMU-CS-81-135, Computer Science Department, Carnegie-Mellon University, July, 1984.

Furness, G. W. (1991). New graphical reasoning models for understanding graphical interfaces. In *Proceedings ACM CHI'91 Conference on Human Factors in Computer Systems* (New Orleans), pp. 71-78.

Green, T.R.G., Bellamy, R.K.E., and Parker, J.M. (1987). Parsing and gnisrap: A model of device use. In *Empirical Studies of Programmers: Second Workshop*. G. Olson, S. Sheppard, and E. Soloway, Eds., Ablex Publishing, Norwood, N.J., pp. 132-146.
Gries, D. (1981). *The Science of Programming.* Springer-Verlag, New York.

Hoare, C.A.R. (1981). The emperor's old clothes. *Communications of the ACM, 24,* 75-83 (1981).

Ichbiah, J.D., Barnes, J.G.P., Heliard, J.C., Krieg-Brueckner, B., Roubine, O., and Wichmann, B.A. (1979). Rationale for the design of the Ada programming language. *SIGPLAN Notices 14,* 6 (Part B, June 1979).

Kurtz, T.E. BASIC. (1981). In *History of Programming Languages.* R. L. Wexelblat, Ed., Academic Press, New York.

Lewis, C., Rieman, J., and Bell, B. (1991, in press). Problem-centered design for expressiveness and facility in a graphical programming system. *Human-Computer Interaction 6,* 319-355.

MacLennan, B. (1987). Principles of Programming Languages. Holt, Rinehart, and Winston, New York.

Maurich, S., and Zorn, B. (in progress). The core macro processing language design: Working document.

Mayer, R.E. (1981). The psychology of how novices learn computer programming. Computing Surveys 13, 121-141.

Pratt, T.W. (1984). *Programming Languages: Design and Implementation.* 2nd ed., Prentice-Hall, Englewood Cliffs, N.J.

Rieman, J., Bell, B., and Lewis, C. (1990). ChemTrains design study supplement. Tech. Rep. CU-CS-480-90, Department of Computer Science, University of Colorado, Boulder, June 1990.

Rist, R. S. (1986). Plans in Programming: definition, demonstration, and development. In *Empirical Studies of Programmers.* E. Soloway and S. Iyengar, Eds., Ablex Publishing, Norwood, N.J., pp. 28-47.

Rosing, M., Schnabel, R., and Weaver, R. (1991). The Dino parallel programming language. *Journal of Parallel and Distributed Computation, 13,* 9, 30-42.

Sebesta, R. (1989). *Concepts of Programming Languages.* Benjamin/Cummings Publishing, Redwood City, Calif.

Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM 29,* 850-858.

Soloway, E., and Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering SE-10,* 595-609.

Spoherer, J.C., Soloway, E., and Pope, E. (1985). A goal/plan analysis of buggy Pascal programs. *Human-Computer Interaction 1,* 163-207.

Tanenbaum, A.S. (1981). *Computer Networks.* Prentice-Hall, Englewood Cliffs, N.J.

van Wijngaarden, A., Mailloux, B.J., Peck, J.E.L., and Koster, C.H.A. (1969). Report on the Algorithmic Language ALGOL 68, *Numer. Math. 14,* 79-218.

Weaver, R.P., and Lewis, C. (1990). Examining the usability of parallel language constructs from the programmer's perspective. Tech. Rep. CU-CS-492-90, Department of Computer Science, University of Colorado, Boulder, Oct. 1990.

Wirth, N. (1971). The design of a Pascal compiler. *Software—Practice and Experience 1,* 309-333 (1971).

Wirth, N. (1988). History and goals of Modula-2. *Byte Magazine,* 145-152 (Aug. 1984).

Wirth, N. (1985). From programming language design to computer construction. Communications of the ACM 28, 160-164.

Wirth, N., and Gutknecht, J. (1989). The Oberon system. *Software—Practice and Experience, 19* .

Wirth, N. (1971). Program development by successive refinement. *Communications of the ACM 14,* 221-227.

Zloof, M. (1975). Query by example. *AFIPS Conference Proceedings 44,* pp. 431-432.