

Using the Programming Walkthrough to Aid in Programming Language Design

brigham bell

*U S WEST Advanced Technologies, 4001 Discovery Drive, Suite 270, Boulder, CO
80303, U.S.A.*

wayne citrin, clayton lewis and john rieman

*Department of Computer Science, Campus Box #430, University of Colorado, Boulder
80309-0430, U.S.A.*

robert weaver

*Mathematics and Computer Science, 206 Korman Center, Drexel University, Philadelphia,
PA 19104, U.S.A.*

and

nick wilde and benjamin zorn

*Department of Computer Science, Campus Box #430, University of Colorado, Boulder
80309-0430, U.S.A. (email: zorn@cs.colorado.edu)*

SUMMARY

The programming walkthrough is a method for assessing how easy or hard it will be for users to write programs in a programming language. It is intended to enable language designers to identify problems early in design and to help them choose among alternative designs. We describe the method and present experience in applying it in four language design projects. Results indicate that the method is a useful supplement to existing design approaches.

key words: Programming walkthrough Programming language design Writability Design methods

INTRODUCTION

Many new programming languages, and extensions to existing ones, are being designed. Although some of these languages follow the tradition of mainstream general-purpose language design, many do not, because they are intended to support specific application areas such as document formatting, or because they use novel paradigms such as graphical presentation of programs.

As language designers with non-traditional interests we have found little guidance in the literature on language design. There are extensive and insightful discussions of the features of general-purpose languages such as Pascal (for example Reference 1) but little of this material rises to the level of abstraction needed to guide the design of languages of very different character. The literature focuses on designs,

not on designing, and hence its usefulness falls off sharply as the designs one needs depart from those discussed.

A further limitation of the literature, with respect to our goals as designers, is that it does not address the whole spectrum of design objectives. In particular, there is very little discussion of how to design languages so that programs will be easy to write, yet this is a primary objective of many designs. For example, the purpose of graphical dataflow languages for scientific visualization, such as AVS,² is to permit scientists to specify visualizations easily and quickly. There is little to assist someone confronting this design challenge, though there are extensive discussions of some other objectives, including efficiency of implementation, completeness of specification, and susceptibility to formal analysis.

In our own work we have developed a method that we have found useful in designing languages that fall outside the well-studied paradigms for general-purpose languages, especially where ease of programming is a primary concern. We present this method here, beginning by outlining the existing approaches to ease of programming that can be discerned in the literature. We show how our method, the programming walkthrough, relates to these approaches. We then describe our method in sufficient detail to permit interested designers to apply it in their own work. Finally, we describe our experience in using the method in four language design projects and weigh its costs and benefits as a supplement to current approaches.

Approaches to designing for ease of programming

As we have already noted, there has been little explicit discussion in the literature of ease of programming as a goal in language design. Nevertheless, a number of general approaches to the problem can be distinguished in reports of design projects. Some textbook writers have ventured more general observations than the designers of particular languages, and we note these as well.

1. 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,³ 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^{4,5} 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.
2. Judgement 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 such as Niklaus Wirth, who has produced many designs, 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.^{1,6-8}

3. Consideration of the role of abstract characteristics of languages 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.⁹⁻¹¹ 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.¹²

Because ease of programming often remains an implicit goal, it is not always clear how these or other abstract characteristics are meant to relate to ease of programming, as opposed to other aspects of a design. But some writers are explicit about this: Sebesta¹³ includes simplicity and orthogonality in a list of characteristics supporting ease of programming, and MacLennan¹⁴ sketches a psychological argument that simple, orthogonal designs are easier for programmers to remember than others.

4. Attention to the process of writing programs. Increasing ease of programming by tuning a language design to support the process of writing programs would seem a natural idea. But this approach has not been common, perhaps in part because of its involvement with psychology. One group that did take this approach were the designers of BASIC: they analyzed the ideas needed to write and run programs in existing languages and looked for opportunities to eliminate ideas by simplifying the process.¹⁵ 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.¹⁶⁻¹⁸ 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. Although 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.¹⁷

Refining the focus on the programming 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 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. Secondly, 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. This knowledge includes the general problem-solving techniques investigated by Soloway and others as well as language-specific details. 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 ease of programming placed in other categories above. In all of these approaches, mental operations figure only implicitly, but we argue that an explicit description of mental processes provides a useful perspective on judgements that otherwise remain rather vague. For example, it encourages one to think about just how simplicity might support ease of programming, and hence provides a basis for determining what kinds of simplicity are valuable and which are not.

THE PROGRAMMING WALKTHROUGH METHOD

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 such as typing or reading. We presume that the steps that are taken are drawn from a collection of possibilities, so that steps must 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,^{19,20} Soloway and colleagues on Pascal,²¹⁻²⁴ and Anderson and colleagues on LISP.²⁵⁻²⁷ 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, which are ways of deploying the features of a programming language for particular purposes, such as the use of an accumulator variable in summing an array. A student who knows Pascal, for example, 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 ease of programming be assessed for a language design? The key to our attack is to describe the steps required in writing a program for one or more problems and to describe the guiding knowledge needed to select these steps. These descriptions of the programming process can then be examined with the following questions in mind.

1. How long is the process? Are there opportunities to eliminate steps by changing the design?
2. 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.
3. Are there steps that require unspecified guiding 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.

Outline of the walkthrough process

To make the preceding explanation more concrete, we outline the steps required to conduct a walkthrough.

1. Define the language along with a suite of sample problems and solutions.
2. Convene the walkthrough analysts.
3. Perform the walkthrough of one or more problems. Specifically, the walkthrough is intended to identify the following things:
 - (a) guiding knowledge needed by programmers to solve the problem
 - (b) a sequence of steps that result in a solution to the problem, each step of which is explained by some piece of guiding knowledge.
4. Analyze the process, including looking at what knowledge is needed and how many steps were required.
5. Apply the results. Based on the analysis, modify the language design.

The steps in the walkthrough process are illustrated in an example below and explained in greater detail in the section that follows it.

The CMPL language

Before presenting the walkthrough example in the next section, we first present a basic overview of CMPL²⁸ (Core Macro Processing Language), which is being evaluated by that walkthrough. CMPL is intended to extend the functionality of the C-preprocessor and the Unix M4 macro processor. In its original form, the language provides a typed-macro facility that allows the programmer to define syntax extensions to the language at compile time. The parameters and context of macro calls are checked for syntactic correctness by the macro processor. The language also allows internal macro variables and computation while processing macros.

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). For the purposes of this discussion, CMPL supports primitive types such as `int`.

In the definition, the `<InputTemplate>` describes the format of a macro call, consisting of literals that much be matched exactly in the macro call and argument declarations. Argument declarations are delimited by square brackets and are composed of type names, argument names, and CMPL keywords, which provide special functionality. For, example, declaring the argument `start` of type `int` would be specified as `[int: start]` (the different parts of an argument declaration are separated by colons). An example of a keyword would be `rest`, the CMPL keyword for a list, or `delimit`, which identifies how items in a list are separated. The following illustrates an argument declaration with keywords.

```
[rest: delimit ',': int: input]
```

This code declares the argument `input` to be a list of type `int` delimited by commas.

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 variables including input arguments. Variables are prepended by a `$` and delimited by square brackets. A simple example is

```
[$[inputA] + $[inputB]
```

which prints out a summation of the two input arguments. The type of a variable can be changed, similar to a cast in C, by prepending the token for the variable with `([Type])`, where `Type` is either a `TypeName`, a CMPL keyword that affects the type (such as `rest:`), or a legal combination of these.

Example: a walkthrough for CMPL

The programming walkthrough sessions for CMPL were done by four of us with no experience in CMPL, working with the designers of the language. The designers provided a written description of CMPL and a large suite of 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 to three hours long. During the walkthroughs, the designers provided help or explanations when asked, but did not 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.

The following is an outline of the first CMPL session in which we analyzed the following problem: write a macro that takes arbitrarily many integer arguments and produces a sum expression. In this outline we identify the specific steps required to solve the problem and also the guiding knowledge needed to perform each step. For this walkthrough the designers did not provide a preferred solution, though they had one in mind.

We began the walkthrough by proposing that the CMPL programmer should adopt a simple plan for writing a macro, based on the fact that CMPL macro definitions have four parts:

Step 1

Decompose the task of writing a macro into writing a result declaration, an input template, a computation, and a result description. Then put the pieces together.

Guiding knowledge: These are the parts of a CMPL macro definition.

We encountered a problem immediately: result declarations can be typed or typeless, but we could see no good basis for deciding between these alternatives. So we put:

Step 2

Use the result declaration [].

Guiding knowledge: We don't know! The typeless case seems simpler.

After some discussion we decided that writing the input template required insight into how the result form would be computed, since one role of the input template is to provide names for parts of the input that will be referred to later. To get this insight we proposed:

Step 3

Consider the sample input–output pair:

Input: add(1,2,3,4)

Output: 1+2+3+4

Guiding knowledge: Use sample input and output to guide writing the input, computation, and output parts of the macro.

Step 4

Observe that the output is a reformatting of the argument list in the input.

Guiding knowledge: Compare the output and input to see what parts of the input are rearranged or modified in the output. CMPL allows lists to be delimited by anything, so watch for lists in the input that can be reformatted as part of the output using the language's type casting mechanism.

Step 5

Use the input template

```
add([rest: delimit ' ': input])
```

Guiding knowledge: This is CMPL syntax for a description that accepts the `add()` structure, which will be thrown away, and gives us the name `input` for a list of items delimited by commas. The programmer must know this syntax, including the fact that `rest` is the CMPL keyword for a list. We noted this last point as needlessly obscure.

We noted that this input template does not do any argument type-checking. So we added:

Step 6

Revise the input template to

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

Guiding knowledge: Rule of thumb: tighter type-checking on input is better. CMPL syntax for lists allows an element type to be specified. The type for integers is `int`.

Step 7

Plan to work on the output specification before the computation.

Guiding knowledge: Even though the computation section comes next in the form of a CMPL macro, you don't know what you need to compute without looking first at the output.

Step 8

Write the output description

```
result "$([rest: delimit '+'] input)"
```

Guiding knowledge: The guiding knowledge used at Step 4 already pointed out how the output for this problem can be defined given the list `input`. We are also again assuming knowledge of CMPL's list construct, together with the details of the syntax for the result description.

Step 9

Note that no computation is required.

Guiding knowledge: The computation can be empty if the output can be described directly using parts of the input.

Step 10

Put the pieces together:

```
define [ ] add([rest: delimit ',': int: input])
begin
result
```

```
"$([[rest: delimit '+']) input]"  
end
```

Guiding knowledge: Still working on the plan from Step 1. Programmer must know the overall form of the definition.

In this summary the walkthrough appears more orderly than it actually was. In particular, we did not recognize that the problem could be solved by list reformatting using type casting until we had already devised the input template, including type checking. But the summary shows how we would say a programmer should approach the problem, and what knowledge would be needed to guide the programmer into this approach.

It is essential to realize that the goal of the walkthrough is not to capture how the analysts approach the problem. Rather, as we have recorded in the summary, the goal is to develop a prescription for solving the problem, given the design of the language, and to spell out what knowledge is required to make the prescription work.

In this example the solution we developed 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. But the designers decided not to intervene in the walkthrough to push their intended solution, because the alternative seemed to them to be a more elegant use of the design.

Results from the CMPL walkthrough

What was learned from 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 the distinction between typed and typeless macros was highlighted as problematic, in that the analysts could not formulate knowledge adequate to guide this choice. Further, the designers were not able to motivate the choice without substantially complicating the overall picture, as it emerged from the walkthrough process, of what a macro is and how it should be written. 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.

The difference between the designers' expected solution for the problem and the solution developed in the walkthrough was not seen as a problem in the context of the emerging picture of the process of writing macros. 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 seen in Steps 3 and 4, 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.

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 defined in terms of 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, analysis of ease of programming should relate to the designer's intentions in the design. Simply knowing whether or not the design makes programming easy is not as useful as knowing whether the features the designer intended to make programming easy 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.

What if there are many possible ways to approach a problem in the language? The designer can choose to examine as many of these as are of interest in separate walkthroughs. The points where approaches diverge deserve special attention in the analysis. On what basis will the programmer be able to choose one approach among those available? If some approaches are clearly preferable to others, how will this be apparent to the programmer?

4. Convene the walkthrough analysts. A designer working alone can perform a walkthrough, as we have done in some of the applications we describe below. But we have also used groups of analysts, with designers acting as observers rather than full participants, as we describe below for other applications. Outsiders can help designers do a more complete analysis, not skipping over steps that seem from an inside perspective uninteresting or obvious. They can also detect alternative solution paths that designers may have overlooked, as we illustrate below. On balance we think a solo walkthrough by the designer is better than none, and well worth doing, but a walkthrough with outside participants is preferable.

If there are outside members of the walkthrough team the designer should

not reveal the intended solutions to the problems until it becomes clear whether or not the analysts head off in the expected direction on their own. If they do not the designer might still opt not to reveal the intended solution, if the direction the analysts have found is interesting. If a series of problems is to be analyzed the designer might withhold any solutions until the analysts have made at least an initial attempt on all problems.

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 to 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.

Steps based on guiding knowledge commonly fall into two categories:

- (a) *Framing steps* are ones in which the programmer adopts a plan for attacking the problem or some part of it. Typically the first step of a walkthrough, like Step 1 in the example, is of this kind; Step 3 in the example is another. They are guided by plan knowledge that suggests a way to break down a problem into subproblems, as is illustrated by the guiding knowledge for those steps. Often plan knowledge is largely language independent in content.
- (b) *Application steps* do the work laid out in framing steps. Step 4 and 5 in the example is typical. Application steps are guided by method knowledge that describes how to carry out some part of the plan.

Often the framing of a simple plan and its application occur consecutively. In such a case the analysts may opt to collapse the two steps into one, as seen in Step 6 in the example. The associated guiding knowledge has to include both the plan and the method needed to apply it.

A useful check on the walkthrough is whether each application step has been preceded by or collapsed with a corresponding framing step. If not, the question of how the programmer knows to do what is done has not been addressed, and there is probably a gap in the guiding knowledge.

As noted above, it is important to keep in mind 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 nothing need be said about it. Even easy steps must be examined at least enough to see what knowledge is needed to make them easy.

As the analysts work through a problem it may sometimes appear 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. Problems in programming 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, 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 at two other aspects of the results. First, are there steps whose correctness is actually unclear? If so, the language design must be incomplete or ambiguous. Secondly, 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. In such cases the analysis focuses on comparing the walkthrough results for the two designs, to decide if one version has fewer 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 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. Secondly, 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.

If difficulties can be eliminated by changing the design of the language, well and good. But where this is not possible there is still value in the walkthrough results. Guiding knowledge that is language-independent may be background knowledge that programmers will be assumed to have, and this knowledge can be used to communicate expectations to prospective users. Guiding knowledge that is language-specific, or that the designer does not wish to require as background knowledge, must be covered in documentation. The walkthrough results clarify the content and importance of this information. In particular, the results will show the extent to which the usual lists of language features must be supplemented by information about how to choose among and organize these features.

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.

More on CMPL

The CMPL walkthrough, including the example discussed earlier, was done in two sessions over a period of five to six hours. These sessions 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, whereas 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 the original design the designers viewed typed macros as new grammar rules being dynamically added to the syntax of the underlying language. For example, to add a new operator \$ to C, the programmer would define a BNF syntax rule for the operator and would also need to specify the associativity and precedence of that operator. From this specification, the macro system would have to generate a new, updated parser for the revised grammar. This approach is both hard to use, because the programmer must understand how to write grammars, and hard to implement, because it requires the creation of parsers for new languages incrementally, a task that is difficult to do efficiently.

In the revised design, the programmer would specify that the macro \$ required arguments on either side that parsed to expressions in the base language. The result of the macro must also be an expression in the base language:

```
define [(expression):e1] $ [(expression):e2] returns [(expression)]
```

In this new design, no changes are made to the grammar of the base language. The new approach takes advantage of the user's understanding of type-checking to ensure that macro processing results in well-formed text, and at the same time does not require any sophisticated implementation technology.

To the best of the designer's 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 through added base-language grammar rules.

It is clear that this design change resulted from the interaction between 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. Although 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 language. As it was, the results could be incorporated in what was still the early conceptual phase of the design.

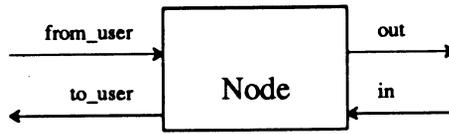
A visual language for communication protocols: MFD

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.

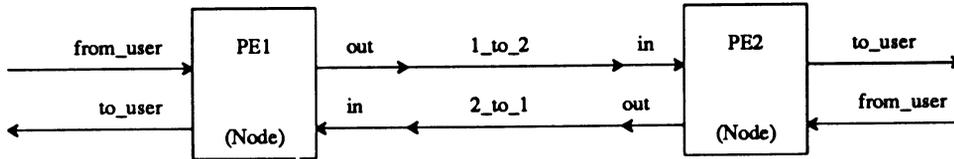
Figure 1 shows an MFD specification for a simple stop-and-wait protocol, which requires that the sender of a message wait until the previous message has been acknowledged before the next message is sent. The heart of the specification is the message flow diagram near the bottom of the Figure. The diagrams above it serve to describe the nodes that appear in the message flow diagram and how they are connected.

Circles in the message flow diagram show events in a scenario, arranged in columns beneath the nodes that perform them. Time runs down the page, with message exchange shown by labeled 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.

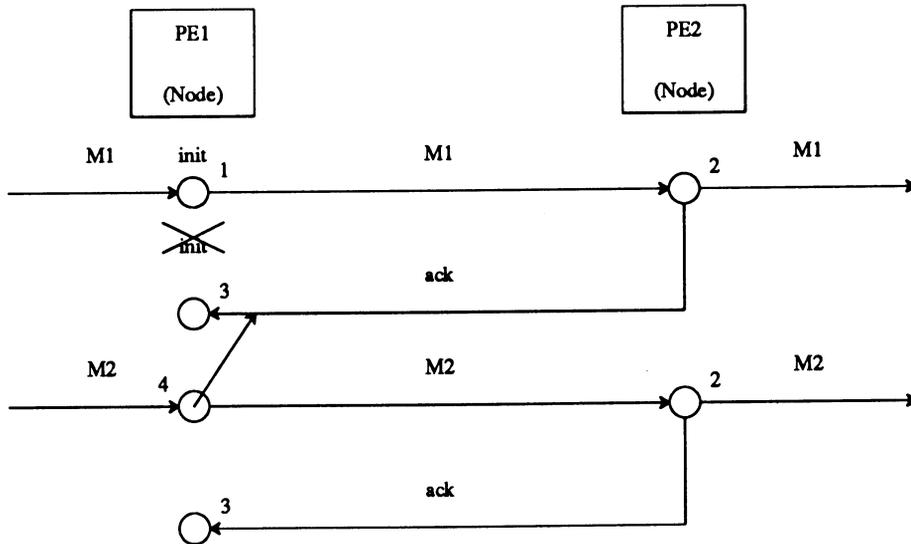
The protocol behavior can now be read off the diagram. If node PE1 is in the `init` state, and it receives a message from its user, it sends the message to PE2 and leaves the `init` state. If node PE2 receives a message from PE1, it passes it on to its user and sends an `ack` (acknowledgement) message 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. This corresponds to the rule that a message cannot be sent until an acknowledgement for the previous message has been received. If PE1 receives a



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 from its user and it has just received an ack, it passes the message to PE2. Note that an explicit init state is necessary, since there can be no message to be acknowledged before the first message is sent. This behavior is described in the rules, shown at the bottom of the figure, that are generated by the MFD system.

The original design of MFD was closely based on the message-flow diagrams displayed by the Cara programming environment,²⁹ 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,³⁰ from textual descriptions. The alternating bit protocol is a protocol that attempts to remedy certain deficiencies of the stop-and-wait protocol through the use of a single sequencing bit attached to the message. Sliding window protocols are a family of protocols designed to allow communication between transmitters and receivers of differing speeds. It also allows messages to be received out of order. Associated with each message packet is a sequence number. The receiver accepts messages whose sequence numbers fall within a certain range (called the 'window'). When a contiguous block of messages at the bottom end of the window have been received, the receiver 'slides' the window a corresponding number of positions and a new range may be accepted. Similarly, the transmitter can send or resend any message within its own window of sequence numbers; when a block of messages at the bottom of the window is acknowledged, the transmitter's window advances. For a more detailed discussion of these and other protocols, the reader should consult Reference 30.

Four of us, none of whom were experienced protocol designers, read the MFD documentation and jointly attempted to specify the protocols. The designer (Citrin) did not provide the MFD solutions. The language designer was present, but did not intervene unless requested by the others. The walkthrough session took about three hours.

There were five 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. Although 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 revealed three 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. The other two ambiguities dealt

with meanings of a kind of annotation not shown in the example. The original design of MFD allowed the programmer to specify running values for PEs' state variables on the diagram. It was not clear to the testers whether the explicit running values or the values computed as the result of executing textual actions associated with events took precedence when the two were in conflict. This was an oversight on the part of the language designer. The ambiguity was resolved by eliminating the explicit running values as a feature of the language. Instead, display of running values was made an optional feature of the MFD environment. New values of state variables cannot be entered through this method, as was previously allowed, but may only be specified through textual event actions. Without early walkthroughs, it is possible that this ambiguity might not have been discovered until much later.

The second ambiguity concerned the meaning of state facts (generalized system state that can be checked and altered by PEs). State facts may be checked, asserted, or retracted, and may contain variables (which must be instantiated when the fact is asserted). The original MFD definition did not permit the programmer to distinguish between state facts that contained (instantiated) variables and state facts that contained constants. The revised MFD definition included additional syntax to make the distinction.

Finally, the walkthrough revealed the opportunity to simplify the way in which complex messages are interpreted by events. Messages are given in the message flow diagram with all variables instantiated to the values they would possess at that point in the scenario. The programmer then specifies the methods by which these instantiated values (called exemplars) are created (in the case of the transmitting event) or dissected and unified with other variables (in the case of the receiving event). These specifications are called variable associations. The original MFD design allowed associations with single variables, more complex message substructures, or entire messages. This proved to be confusing to the testers, and the revised MFD definition permitted only associations involving entire messages, through the use of message templates. This simplified the mechanism at the cost of some flexibility. Although 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. The solution incorporated into the revised design proved much more understandable to the testers.

A language for graphical simulations: ChemTrains

The walkthrough evaluations of CMPL and MFD occurred early in the design process 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 design effort spanned several months, including many days of walkthroughs. 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 non-programmers with an interactive visual programming environment for modeling qualitatively defined systems, such as a Turing machine or document flow in an organization.^{31,32} 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

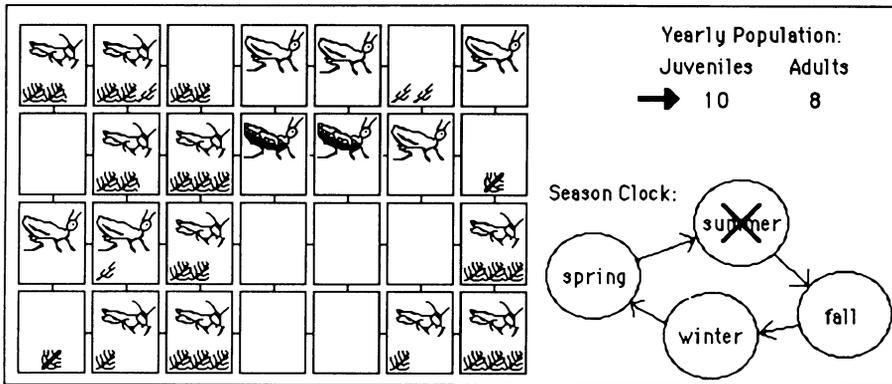


Figure 2. A ChemTrains simulation display of grasshoppers in a field

paths. Behavior of these entities is controlled by ‘if-then’ production rules, also described graphically by the user. The language similar to BitPict,³³ another rule-based visual language, except that patterns in ChemTrains contain objects and relationships among them rather than pixel configurations as in BitPict.

Figures 2 and 3 show part of a ChemTrains program for simulating grasshopper population dynamics. Figure 2 shows a snapshot of the simulation, and Figure 3 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 focused on six specific ‘target problems’ that the language should support,

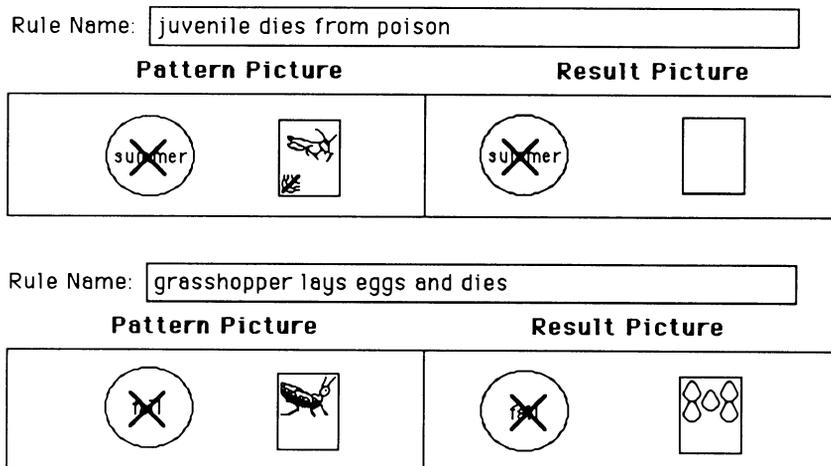


Figure 3. Two of the twenty-five rules governing the grasshopper simulation

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 that provided by power, influenced by the OPS family of rule-based systems;³⁴ and 'ShowTrains' aimed for concreteness, allowing the user to specify rules by demonstration.

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 designers' differences of opinion as to which was the best design. Overall, the power-oriented OPSTrains design was found to be easiest to write programs in, as described in the next paragraph. ShowTrains placed second, and ZeroTrains placed last, with each solution requiring work directly attributable to getting around ZeroTrains' 'simplifying' assumptions.

The walkthroughs demonstrated several things that general arguments had left unresolved. Most strikingly, the walkthroughs contradicted the designers' 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 usable. The additional features in OPSTrains directly matched the problem-solving needs of the programmer, whereas 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 expected variables 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.³⁵ The walkthroughs indicated that a little bit of knowledge could make an apparently difficult language feature easy to use.

Validating the walkthroughs

Although the walkthroughs and the design process had focused 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 synchronization 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. Secondly, the cursory description of the required knowledge was sometimes inadequate to overcome users' expectations about how they should proceed. Thirdly, 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 Reference 36; the user testing is covered in Reference 37.

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 prototyping³⁸ 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 directly 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 analyzing only the problems most clearly affected by a design change, and by focusing individual walkthroughs on just parts of solutions that were affected. This narrow focus had the side benefit of permitting the designer to detect subtle differences in ease of programming between alternatives being examined on small parts of a larger problem.

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 Rosing, Schnabel and Weaver.³⁹ 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++ prototype, 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 remote communication of a variable from local reads and writes.

The walkthroughs were performed by two of the authors.⁴⁰ 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. Although 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 that guiding knowledge was needed 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 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.

DISCUSSION

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 much later in the design process. The information produced can include

- (a) aspects of problems that require many steps
- (b) choices for which adequate guidance is not available
- (c) choices that require extensive or esoteric knowledge
- (d) language features whose definitions are ambiguous
- (e) a clarified or modified view of how the language might be used.

Designers can apply this information by

- (a) redesigning the language to provide simpler solutions
- (b) redesigning the language to avoid problematic choices
- (c) documenting the knowledge needed to use the language
- (d) clarifying the definitions of language features
- (e) redesigning the language to support a different approach to using it, or
- (f) choosing between design alternatives by comparing walkthrough results.

A key point supported by the case studies is that programming walkthroughs go beyond a designer's unaided judgements in evaluating language designs. The CMPL, MFD, and ChemTrains walkthroughs all gave results contradicting the designers' predictions about their languages. This is a primary advantage of the programming walkthrough over existing approaches to the design for ease of programming. 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 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 Parker⁴¹ 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 in that the designer may not keep an accounting of the knowledge needed to arrive at a solution, a key determinant of ease of programming. Suppose a designer arranges for test users to solve sampled problems, collecting thinking-aloud protocols (as in Reference 36) 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, the method would essentially be a walkthrough. Incidentally, we think the walkthrough is good preparation for user testing. The walkthrough can help clear away problems before testing begins, and it can suggest where testing effort should be focused.

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 earlier, for example, the very simple problems used in the initial ChemTrains evaluation caused us to miss key issues that would arise in real use. The walkthrough must be seen as a supplement to other methods, including those based directly on designers' judgements, that can assess the importance of language characteristics not exercised in any particular sample problem.

The walkthrough is not a substitute for judgement in any case, since judgement 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 judgement but that it guides the application of judgement in a productive way.

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 using relatively small (though not necessarily simple) problems, and when done early in design so changes can easily be made. This point about timing is obvious when pointed out but nevertheless easy to overlook: the biggest payoffs from the walkthrough come not from superficial refinements, which might be made as afterthoughts to the basic design, but from better decisions on basic design alternatives.

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 more than 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 results, and follow-up discussions. Yet in each case the walkthrough yielded clear evidence of problems that had escaped the designer's notice, but that could be fairly easily corrected. Importantly, some of the results led to substantive, not superficial, design revisions in each case.

acknowledgements

This work was partly supported by the following grants sponsored by the U.S. National Science Foundation: CCR-9208486, IRI-8722792 and IRI-9116640. We would also like to acknowledge support from US West Advanced Technologies. Finally, we thank the anonymous reviewers of the paper for their insightful comments.

REFERENCES

1. N. Wirth, 'An assessment of the programming language Pascal', *IEEE Trans. Software Engineering*, **SE-1**, 192–198 (1975).
2. Stardent Corporation, Concord, MA, *AVS Tutorial Guide*, 1990.
3. J. Backus, 'The history of FORTRAN I, II, and III', in R. L. Wexelblat, (ed.), *History of Programming Languages*, Academic Press, New York 1981.
4. W. Citrin, 'Visualization-based visual programming', *Tech. Rep. CU-CS-535-91*, Department of Computer Science, University of Colorado, Boulder, CO 80309, July 1991.
5. W. Citrin, 'Design considerations for a visual language for communications architecture specifications', in *Proceedings 1991 IEEE Workshop on Visual Languages*, 1991.
6. N. Wirth, 'Program development by successive refinement', *Communications ACM*, **14**, 221–227 (1971).
7. N. Wirth, 'From programming language design to computer construction', *Communications ACM*, **28**, 160–164 (1985).
8. N. Wirth and J. Gutknecht, 'The Oberon system', *Software—Practice and Experience*, **19**, 857–894 (1989).
9. A. R. Feuer and J. H. Gehani, 'A methodology for comparing programming languages', in A. R. Feuer and J. H. Gehani, (eds), *Comparing and Assessing Programming Languages: Ada, C, Pascal*, Prentice-Hall, Englewood Cliffs, NJ, 1984.
10. C. A. R. Hoare, 'The emperor's old clothes', *Communications ACM*, **24**, 75–83 (1981).
11. T. Pratt, *Programming Languages: Design and Implementation*, Prentice-Hall, Englewood Cliffs, NJ, second edn, 1984.
12. A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck and C. H. A. Koster, 'Report on the algorithmic language ALGOL 68', *Numer. Math.*, **14**, 79–218 (1969).
13. R. Sebesta, *Concepts of Programming Languages*, Benjamin/Cummings Publishing, Redwood City, CA, 1989.
14. B. MacLennan, *Principles of Programming Languages*. Holt, Rinehart, and Winston, New York, NY, 1987.
15. T. E. Kurtz, 'BASIC', in R. L. Wexelblat (ed.), *History of Programming Languages*, Academic Press, New York, NY, 1981.
16. E. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.
17. D. Gries, *The Science of Programming*, Springer-Verlag, New York, NY, 1981.
18. N. Wirth, 'The design of a Pascal compiler', *Software—Practice and Experience*, **1**, 309–333 (1971).
19. P. Bayman and R. Mayer, 'A diagnosis of beginning programmers' misconceptions of BASIC programming statements', *Communications ACM*, **26**, 677–679 (1983).
20. R. E. Mayer, 'The psychology of how novices learn computer programming', *Computing Surveys*, **13**, 121–141 (1981).
21. E. Soloway and K. Ehrlich, 'Empirical studies of programming knowledge', *IEEE Transactions on Software Engineering*, **SE-10**, 595–609 (1984).

22. E. Soloway, 'Learning to program = learning to construct mechanisms and explanations', *Communications ACM*, **29**, 850–858 (1986).
23. R. S. Rist, 'Plans in programming: definition, demonstration, and development', in E. Soloway and S. Iyengar (eds), *Empirical Studies of Programmers*, Ablex Publishing, Norwood, NJ, 1986, pp. 28–47.
24. J. C. Spohrer, E. Soloway and E. Pope, 'A goal/plan analysis of buggy Pascal programs', *Human-Computer Interaction*, **1**, 163–207 (1985).
25. J. R. Anderson, R. Farrell and R. Sauers, 'Learning to program in LISP', *Cognitive Science*, **8**, 87–129 (1984).
26. J. R. Anderson and R. Jeffries, 'Novice LISP errors: undetected losses of information from working memory', *Human-Computer Interaction*, **1**, 107–131 (1985).
27. J. R. Anderson and E. Skwarecki, 'The automated tutoring of introductory computer programming', *Communications ACM*, **29**, 842–849 (1986).
28. S. Maurich and B. Zorn, 'The core macro processing language (CMPL) design: working document', work in progress.
29. A. A. R. Cockburn, W. Citrin, R. F. Hauser and J. von Kaenel, 'An environment for interactive design of communications architectures', *Proc. 10th International Symposium on Protocol Specification, Testing, and Verification*, 1990.
30. A. S. Tanenbaum, *Computer Networks*, Prentice-Hall, Englewood Cliffs, NJ, 1981.
31. C. Lewis, J. Rieman and B. Bell, 'Problem-centered design for expressiveness and facility in a graphical programming system', *Human-Computer Interaction*, **6**, 319–355 (1991).
32. B. Bell and C. Lewis, 'ChemTrains: a language for creating behaving pictures', *Proc. 1993 IEEE Conference on Visual Languages*, 1993.
33. G. W. Furness, 'New graphical reasoning models for understanding graphical interfaces', *Proc. ACM CHI'91 Conference on Human Factors in Computer Systems*, 1991, pp. 71–78.
34. C. L. Forgy, 'OPS5 user's manual', *Tech. Rep. CMU-CS-81-135*, Computer Science Department, Carnegie-Mellon University, July 1984.
35. M. Zloof, 'Query by example', in *AFIPS Conference Proceedings*, **44**, 431–432 (1975).
36. J. Rieman, B. Bell and C. Lewis, 'ChemTrains design study supplement', *Tech. Rep. CU-CS-480-90*, Department of Computer Science, University of Colorado, Boulder, CO 80309, June 1990.
37. B. Bell, J. Rieman and C. Lewis, 'Usability testing of a graphical programming system: things we missed in a programming walkthrough', *Proc. ACM CHI'91 Conference on Human Factors in Computer Systems*, 7–12 (1991).
38. W. Buxton and R. Shneiderman, 'Iteration and the design of the human-computer interface', *Proc. 13th Annual Meeting of the Human Factors Association of Canada*, 1980, pp. 72–81.
39. M. Rosing, R. Schnabel and R. P. Weaver, 'The DINO parallel programming language', *Journal of Parallel and Distributed Computing*, **13**, 30–42 (1991).
40. R. P. Weaver and C. Lewis, '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, CO 80309, October 1990.
41. T. R. G. Green, R. K. E. Bellamy and J. M. Parker, 'Parsing and gnisrap: a model of device use', in G. Olson, S. Sheppard and E. Soloway (eds), *Empirical Studies of Programmers: Second Workshop*, Ablex Publishing, Norwood, NJ, 1987, pp. 132–146.