1988

Integrating Editing in a Monolingual Environment

Prasun Dewan

Report Number:
88-763

INTEGRATING EDITING IN A MONOLINGUAL ENVIRONMENT

Prasun Dewan

CSD TR-763
April 1988

Integrating Editing in a Monolingual Environment

Prasun Dewan
Department of Computer Science
Purdue University
W. Lafayette, IN 47907

ABSTRACT

A monolingual environment offers a common language for the various activities supported by the system. Previous work shows how a monolingual environment may be created for carrying out the programming, command invocation, and debugging tasks. In this paper extensions to such an environment to also support editing are discussed. The role of editing is compared with the role of interactive programming in a monolingual environment. An approach is described for integrating editing and programming languages into a powerful combined language that supports incremental replacement, the notion of composing a value before committing it, undo and redo capabilities, user-programmed changes, and selection and naming of data structures in all contexts. The approach is compared with related work, and its application to multilingual environments is discussed.

1. Introduction

Traditional interactive environments are multilingual, that is, they offer different languages in different contexts. In particular, they offer different languages for command invocation, debugging, and programming. Typically, each one of these languages is developed independently without considering the constructs offered by other languages. This situation leads to at least two problems:

First, the environment is *non-uniform*, that is similar facilities are offered by different constructs in different languages. For instance, a Unix[1] user using the *shell* command language and the *C* programming language

[1]Unix is a registered trademark of AT&T Bell Laboratories

"assigns" a file to another file by executing a command of the form

cp <source file> <dest file>

while he assigns a program variable to another variable by executing a statement of the form

<dest variable> = <source variable>

Second, each individual language is *weak*, that is, it does not offer several useful facilities offered by other languages. For instance, the *shell* command language does not support type-checking, which would be useful to ensure that command procedures do not manipulate arbitrary files. Similarly, the C programming language does not offer interactive programming, which could save the user from the overhead of the edit-compile-execute process when evaluating simple expressions.

Heering and Klint describe these problems in detail in [9] and present an approach for extending a conventional programming language to also support command invocation and debugging. Their approach is based on the following three principles:

(1) Each extension should add substantial power in all three contexts: programming, command, and debugging.

(2) The semantics of each extension must be context-independent.

(3) The resulting language must provide the user with adequate facilities (compared with existing integrated and non-integrated languages) in all three contexts.

One limitation of their approach is its lack of support for editing. Heering and Klint realized this limitation and left integration of editing as future work. In this paper we discuss extensions to their approach to overcome this limitation. The rest of the paper is organized as follows: Section 2 summarizes the basic components of a monolingual environment. Section 3 defines an editing model and associated terminology. Section 4 motivates the need for editing in a monolingual environment. Section 5 is the major part of the paper. It presents an approach for integrating editing primitives with the constructs offered by the common programming-command-debugging language. Section 6 compares our work with related research, and discusses its applications to multilingual environments.

2. Properties of a Monolingual Environment

In this section, we summarize the basic properties of a monolingual environment of the kind proposed by Heering and Klint, which has most of the features of Smalltalk [7], Interlisp [15], and other environments with similar goals. In the remainder of this paper, we shall use the term *monolingual environment* to refer to such an environment. We shall also refer to the common programming, debugging, and command language supported by a monolingual environment as the *programming language* of the environment. Such a language is an extension of some conventional programming language, which we shall call the *base language* of the system.

In a monolingual environment, there is no distinction between files and local variables, and programs and procedures. The programming language provides nested scopes rooted by a top-level *system procedure*. Declarations made in this procedure are persistent, and replace files of multilingual environments. Declarations made in scopes local to the system procedure are temporary, and last only as long as the scope exists.

An **interact** statement is provided by the system to allow a procedure to transfer control to the user, who can then do interactive programming. All names entered by the user are resolved in the scope of the procedure that executes **interact**. A user can end an interaction session to return control to the procedure.

The command-level in this environment corresponds to interactive programming in the top-level scope. The **interact** statement is implicitly executed by the system procedure at login time. The user may now declare and call new procedures in this scope, which in turn can invoke **interact** to provide interactive programming at deeper levels.

As an example of interactive programming in such an environment, consider modification of a telephone directory. Such a directory can be created as an associative table that maps names to telephone numbers. Then to change an entry in the directory, the user simply executes a statement of the form:

```
telephonenumbers[person] := newnumber
```

To accomplish the same task in a traditional environment, a user would have to enter a special-purpose telephone directory editor (or a general-purpose text editor if the telephone directory is stored in a text file), change the entry, and return to the command level.

One problem with defining traditional programs and files as top-level procedures and data structures respectively, is that most conventional languages do not allow dynamic changes to procedure and type declarations. This feature would be unacceptable in a realistic environment in which programs and the structure of permanent data are expected to evolve. Therefore, extensions are made to the base language to make procedures and types first-class variables whose values can change.

Another important extension made to the base language is an *event association* mechanism of the kind proposed by Hanson [8]. An event association has the form

<event association> := [<label>] when <event> do <action> od

and associates an *event* with an *action*, which is executed whenever the event is *triggered* by some condition. An example of an event association is

```
when x is modified
        do
                display x
        od
```

which causes x to be displayed whenever it is modified. Event associations are useful as a debugging tool. Heering and Klint also show they are useful in the command context to simulate automatic Unix-like makes of related objects, and in the programming context to simulate exception handlers.

All constructs offered by the programming language are available and useful in all three contexts: command, programming, and debugging. As a result the environment is truly monolingual, that is, no context-specific constructs are offered by the system. The notion of a context is only in the mind of the user, who determines the purpose for which a construct is used. The command and programming contexts can indeed be differentiated on the basis of the level of interactive programming. However, the debugging context cannot be differentiated from the other two contexts. For instance, a user executing a statement in the top-level scope may be debugging his system or invoking a command. Similarly, a user executing a statement in a local scope may be debugging or programming.

The monolingual environment has several limitations including its lack of support for multiprocessing, protection, and editing. Lack of protection seems an inherent property of an environment that does not distinguish between debugging and other contexts. Heering and Klint did not consider multiprocessing and editing to simply

limit the scope of their work. In this paper, we shall discuss extensions to their approach to support editing.

3. A Model of Editing

We define below an editing model and associated terminology that we shall use in the remainder of this paper.

Editing essentially allows the user to modify data structures by modifying their visual representations or *presentations*. It offers the user commands to *compose* changes to presentations. These changes can later be *committed* by executing the *write* command, which changes the data structures according to their new presentations. A user may end the editing session by executing the *quit* command, which removes the presentations of the data structures from the display.

Commands are entered by specifying their operands and operations. Operands are specified by *selecting* them from the screen using the facilities of a pointing device and/or cursor keys. Operations are specified via name entry, menu-selection, or special keys. A wide variety of commands are offered to modify presentations. These include commands to *undo* and *redo* other commands.

An editor is a *text editor* if it can edit only text files, otherwise it is a *structure editor*. The main difference between a text and structure editor is that the latter needs to convert between edited data structures and their presentations. In addition, it may provide the user with facilities to:

- format the presentations of structures.
- incrementally check the changes composed by the user for syntax and semantic errors.
- perform *semantic actions* when changes are committed by the user. For instance, a spreadsheet editor propagates changes to related data structures, and a *print queue* editor needs to stop printing the current entry if it is deleted from a presentation of the queue.
- support *structure-editing commands*, that is, commands that understand the structure of data being edited. Examples of these commands are the *elide* and *expand* commands, which allow data structures to be viewed at multiple levels of detail (these commands are illustrated in § 5.1).

4. Role of Editing in a Monolingual Environment

As we considered integration of editing with the programming, command, and debugging contexts, we realized the editing is not a "context" in the same sense as the other three are. Instead, it is a *means* for changing data structures in a context. In this respect, it is similar to interactive programming, which is also a means for changing data structures.

This point is best understood by considering the nature of the editing task. As described in the previous section, editing allows a user to change the value of a data structure by changing its visual representation or presentation. For instance, to change an entry in a telephone directory displayed as

Name: Joe Doe
Number: 317-222-2222

Name: Jim Smith
Number: 608-123-1234

...

the user can change its presentation to, say

Name: Joe Doe
Number: 317-222-2221

Name: Jim Smith
Number: 608-123-1234

...

Interactive programming environment offers an alternative method for modifying data structures based on statement execution. Thus to cause the above change, a user can also execute the statement:

telephonenumbers["Joe Doe"] := "317-222-2221"

Therefore, a natural question at this point is: Why introduce another method for changing data structures?

We justify the inclusion of editing by arguing that it should be part of the "adequate facilities" that a monolingual environment needs to offer. Our argument is based on two useful properties of editing that are lacking in interactive programming:

First, editing allows the user to specify the data structure to be modified by *selecting* its presentation from the screen using a pointing device and/or cursor keys. Interactive programming, on the other hand, requires that the user specify the data structure by *naming* it. Thus in the above example, in the editing mode, the user specified the (part of the) telephone number to be changed by selecting its presentation from the screen, while in the interactive programming mode, he specified the number by entering the name "telephonenumbers["Joe Doe"]"

.

Specification by selection saves the user from knowing and entering the name of the data structure to be modified. On the other hand, it may require effort on the user's part to change the display to contain a presentation of the data structure. However, this effort is amortized over all the selections a user makes before changing the display. Thus selection is useful when a user can modify a large number of objects in the current display before changing it.

Second, editing provides *incremental replacement*, that is, it lets the user change only a portion of the presentation of a data structure to enter a new value. Returning to the telephone directory example, in the editing mode a user needs to replace a single character to change the telephone number of "Joe Doe", while in the interactive programming mode, he needs to enter the complete value. The usefulness of this feature is perhaps better illustrated by considering the change of a single statement in a large procedure. Editing allows the user to enter only the changed statement while interactive programming requires reentry of the complete procedure!

Editors also offer other useful facilities not supported by interactive programming. These include the notion of composing changes before committing them, semantic actions, and commands to undo and redo other commands. We believe these primitives also belong in the set of facilities offered by an environment. However, these primitives are not tied to the editing mode, and can also, as we shall show later, be provided in the interactive programming mode.

Inclusion of an editing language in a monolingual environment may seem contrary to the goal of supporting a single language in the system. However, we define a multilingual environment as an environment that provides several context-dependent languages. But editing is a context-independent language that is useful for modifying data structures in the command, programming, and debugging contexts. Traditionally, it has been used solely at the command level for modifying only programs and data files. Recent work on generalized editing [3, 4, 12, 13]

has shown the usefulness of editing other command-level data structures such as directories, process lists, data structures on a disk, and line-printer queues. Moreover, editing can just as usefully be used for modifying internal data structures of a program, for debugging or normal programming purposes. For instance consider an internal variable *v*, defined as

```
v: record
        f1: integer;
        f2: real;
end
```

It would be useful if the user could modify the value of the variable by editing a presentation of the form

```
v:
        f1: 5
        f2: 3.0
```

Thus the editing language, like the programming language, is useful in all contexts, and together, these two languages can form a powerful context-independent language of a monolingual environment. We discuss below an approach for integrating these two languages. Part of our approach is based on our previous work on Dost [2, 3], which illustrated an approach for supporting editing of general data structures.

5. Integration of Editing

Our approach "integrates" editing with incremental programming in two ways: First, it allows a user to mix editing and incremental programming commands to modify common data structures. Such integration is an example of a general kind of integration defined by Ambriola and Notkin in [1]. We shall refer to this integration as *mode integration*. Second, it enhances the editing language with features offered only by the programming language, and vice versa, thus decreasing the necessity to switch modes. We shall refer to this integration as *language integration*. We discuss below both kinds of integration.

5.1. Mode Integration

Interact and Edit Windows

To simultaneously support both the editing and interactive programming modes, we first introduce in the monolingual environment the notion of a *window*. The system may display several windows simultaneously. One

of these is the *interact window*, which is used for interactive programming, and simulates the display of the original window-less monolingual environment. Other windows are *edit windows* used for editing data structures.

The constructs **create_window** and **destroy_window** are added to the programming language to allow edit windows to be created and destroyed dynamically. The former returns a *window number*, which is used to identify the window in other window operations, including **destroy_window**. Like other constructs in the language, these statements may be executed by the user or a procedure.

Different windows are devoted for editing and interactive programming for two main reasons: First, special key bindings may be used in an edit window. Second, edit and interact windows show different aspects of the interaction. The interact window shows the history of interactive programming, while an edit window continuously displays some data structure that may be edited. Several edit windows may be used to display independent data structures separately. For instance, one edit window could display a line-printer queue, while another could display the values of local variables of some procedure being debugged.

The Edit Construct

The **edit** construct may be used to populate an edit window with data structures to be edited. This construct is of the form

edit w, x1, ..., xn

where *w* is a window, and *x1*, ..., *xn* are variables to be edited in the window. It appends presentations of the variables to the contents of the window.

Both global and local variables may be displayed in an edit window. A local variable is automatically removed from the display when the scope defining it is destroyed. A variable may be explicitly removed from the display by execution of the *erase* editor command, which erases its operand(s) from the display. All variables displayed in a window may be erased by executing the *quit* command, which ends the editing session in that window.

As in Descartes [14], the presentation of a displayed variable always reflects its current value. This value may be changed by editing or interactive programming commands. Thus a telephone directory displayed in a window may be changed by editing its presentation or assigning a new value to it. A problem with letting the user

mix editing and interactive programming commands is that the presentation of a variable may change as a user is editing it. Therefore, we serialize updates to variables. The editing and interact windows are associated with locks that determine if the user can execute commands in these windows. Windows get locked and unlocked according to the following rules:

(1) While a command is executing in an interact window, all other windows are locked. These windows are unlocked when the command either returns to the user or executes interact.

(2) While changes are being composed in an edit window, all other windows are locked. These locks are released when the user commits changes in the edit window.

(3) At login time, all windows are unlocked.

Synchronous vs Asynchronous Edit

The **edit** construct proposed by us is similar to a version of it supported in EZ [5, 6] for editing Snobol4 tables and strings. The EZ version of edit is *synchronous*, that is, the user or procedure is blocked from executing other statements until the end of the editing session, when the edited variables are erased from the display. Our version, on the other hand, is *asynchronous*, that is, it is non-blocking and updates the variables asynchronously.

The synchronous version requires less overhead compared to the asynchronous version. In a system that supports the former, the scope of a variable cannot be destroyed while it is available for editing. Therefore, the implementation of a procedure return does not involve erasing any local variables from the display. Moreover, a variable displayed in an edit window cannot be assigned a new value in the interactive programming mode. Therefore, the implementation of an assignment statement does not involve updating the display of the target variable. Finally, the system does not need to lock and unlock windows.

We have chosen an asynchronous edit because of the following advantages it offers over its synchronous counterpart:

- It reduces the the cost of switching between editing and interactive programming modes. The synchronous version requires that the user quit the editing session when he wants to switch from the editing to the interactive programming mode, and reexecute **edit** when he wants to switch back to the editing mode. As a result, the cost of switching between these two modes is high. The asynchronous version, on the other

hand, allows the mode to be changed during an editing session. For instance, a user who has displayed a print queue in an edit window can intermix interactive programming and editing of the queue without having to either start or terminate the editing session on every mode change.

- It lets the variables displayed in edit windows be added incrementally to these windows by several edit statements executed in possibly different scopes. As a result a user can simultaneously be editing local variables of all procedures involved in a procedure-call chain. The user of a synchronous edit, on the other hand, can simultaneously edit only those variables that are visible in the scope in which the statement was executed.
- It lets a variable displayed in an edit window be updated not only by editing commands but also by statements executed in the interactive programming mode. Such updates are not supported by traditional editing systems and can lead to powerful modes of interaction. For instance, the value of a variable (such as a print queue) may be monitored continuously by the user. Moreover, a user can evaluate arbitrary expressions in the programming language to update the display. For instance, a user wishing to cube the value of integer *i* displayed in an edit window can simply execute the statement:

```
i := i*i*i
```

Support for Structure Editing

Variables displayed by the execution of the **edit** construct may be integers, records, procedures, and other modifiable data structures in the programming language. The environment needs to provide a structure editor, which we shall call the *base editor* of the environment, to edit these variables. Our approach is independent of the exact set of commands supported by this editor, which may include commands of existing editors.

Like a Dost dialogue manager, the base editor can use information of the types of the variables to provide structure-editing commands. For instance, it may provide an *elide* command to change a presentation of the form

```
v:
        f1: 2
        f2: 3.0
```

to

<v...>

and an *expand* command to perform the reverse process. The type of a variable may also be used to provide incremental detection of syntax errors.

However, not all facilities offered by a structure editor can be provided solely on the basis of types of structures. These include facilities to (a) format the presentation of data, (b) impose semantic constraints on the value of user-defined data structures, and (c) define semantic actions. We discuss below how these features may be added to a monolingual environment.

Attributes

Our previous work on Dost suggested the use of *attributes* to format the display of a variable. An example of these attributes is the *initialized* attribute, which determines if the presentation of a variable displays a placeholder

<integer>

or its actual value

5

Another example is the *value* attribute, which keeps the current value *composed* by the user, which may be different from the last value committed by him. Attributes are associated in Dost with both variables and types, and an attribute inheritance mechanism is defined for inheriting default attributes of a variable from its type.

Such attributes can also be used in a monolingual environment to format the display of data structures. Unlike Dost attributes, which are kept by a separate editor purely for editing purposes, attributes in the monolingual environment can be supported in the programming language. As a result, they can be modified and examined in the interactive programming mode. (We shall see the use of this feature later.) Moreover, they can be part of a general set of attributes that also keep program information of the kind stored by Ada attributes. In the remainder of this paper, we shall use an Ada-like syntax for referring to these attributes. Thus attribute `a`, of the program entity `x` will be referred to as *x'a*.

Incremental Semantic Checking

We use the mechanism of event associations to support incremental detection of semantic errors. We introduce a new event, *composed*, which is triggered whenever a new value of the variable is composed by the user (which may later be committed by invoking the *write* command). The action part of this event can access the *value* attribute of the variable (which holds the composed value) to check it for semantic consistency. The use of the *composed* event is illustrated by the following event association:

```
when x is composed
        do
                if not odd (x'value) then
                        error "value not odd"
        od
```

The construct error reports errors in a standard error area used by the system.

Often the same semantic constraints need to be imposed on all variables of a certain type. Therefore, we allow the *composed* event to be also associated with all variables of a particular type. This event is triggered whenever any of these variable is composed by the user. In the scope of the action part, the Smalltalk-like pseudo variable self names the specific variable composed. Thus the user may create the following event association:

```
when variable of type T is composed
        do
                if not odd (self'value) then
                        error "------"
        od
```

Semantic Actions

The *modified* event proposed by Hoering and Klint may also be used to associate semantic actions with *commitment* of updates to variables. Thus a user may make the event association

```
when i is modified
        do
                display A[i]
        od
```

It would be useful to associate the *modified* event also with all variables of a certain type. Thus a user may create the event association

```
when variable of type T is modified
        do
                display A[self]
        od
```

5.2. Language Integration

The editing and programming languages support different paradigms of interaction. In contrast to the programming language, the editing language supports selection (§ 4), incremental replacement (§ 4), *undo* and *redo* commands, and multiple windows for interaction. Moreover, it lets the user compose a change before committing it. This facility is useful for avoiding spurious semantic actions when values of structured data are changed. As an example, consider the following *person* data structure

```
person: record
        name, telNum, adr: string;
end;
```

associated with the semantic action

```
when person is modified
        do
                print(person)
        od
```

Assume that the user needs to change both the address and telephone number of *person*. Then he can compose new values for these fields before committing them in order to avoid printing an inconsistent value.

Conversely, in comparison to the editing language, the programming language supports naming of data structures (§ 4). It also supports control constructs for changing values. Furthermore, it lets the user store a sequence of steps in a user-defined procedure which may later be invoked to change values. The programming language is also necessary for declaring the type of a variable, which is used in both the editing and interactive programming mode to ensure that type-correct values are assigned to a variable.

It would be useful if each of the two languages could be enriched with some of facilities found in the other language to reduce the need for changing modes. Indeed some programming and command languages offer facilities offered traditionally only by editing languages, and vice versa. For instance, Argus [10] supports the notion of "composing" a transaction before committing it. The Unix *shell* command language supports commands for

redoing previous commands. Conversely, extendible text editors such as Emacs offer the capability of invoking user-defined functions on text being edited. We propose below similar extensions to the editing and programming languages for bridging the gap between these languages.

The programming language can let the user assign to *value* attributes of variables to compose changes to them. A special **write** construct with the same semantics as the *write* editor command can be provided to commit these changes. Thus a user can change the *person* data structure by executing the following statements

```
person.telNum'value := newTelNum;
person.adr'value := newAdr;
write;
```

Moreover, special **undo** and **redo** constructs may be provided by the language with semantics similar to the ones provided by the base editor.

The programming language may also be extended to support incremental programming in multiple windows. A variation of **interact**, **interact in new window**, may be supported to start incremental programming in a new window, which is destroyed when the user ends that interaction session. Thus each "open" interaction session can be associated with its own window.

The editing language can be enhanced to support the invocation of procedures defined in the programming language. One simple extension is to allow any one-parameter procedure to be invoked as an editor command on a variable whose type matches the parameter of the procedure. The procedure is called with the value attribute of the variable whenever the corresponding editor command is executed. Thus the procedure

```
procedure cube (var x: integer)
begin
        x := x*x*x;
end;
```

may be associated with an editor command which may be invoked on integer variables to cube their values.

5.3. Implementation Aspects

Many of the extensions proposed here have been individually implemented in other environments. For instance, the close coupling between a variable and its displayed value is supported in Descartes. Attributes and structure editing of typed-variables are implemented in Dost. In Dost, procedures and types cannot be edited since

they are not modifyable data structures in the programming language. Moreover, the design of Dost precludes editing of local variables of a procedure. However, the implementations of these and other new extension proposed by us is straightforward and simply follows their semantics described above.

6. Discussion

Related Work

Our work essentially combines the work done by Heering and Klint and the work done in generalized editing [2, 12, 13]. The former proposes a common language and implementation for supporting an interaction style based on incremental programming. The latter proposes a common language and implementation for an interaction style based on editing data structures. Our approach combines these two languages and implementations into an environment that lets the user mix editing and incremental programming in all contexts. Moreover, it enhances the two modes to reduce the need for switching between them.

Our approach is perhaps the closest to the one used in EZ, which also supports both editing and incremental programming. The biggest difference between the two is that our approach supports an asynchronous edit while EZ supports a synchronous edit (§ 5.1). Moreover, EZ does not provide support formatting of data, incremental semantic checking, or semantic actions.

Application to Multilingual Environments

Our approach for combining the interactive programming and editing styles of interaction can also be applied to individual applications of a multilingual environment. For instance, a Lisp or ML [11] interpreter can be augmented with extensions to also support editing of data structures. Similarly, a conventional debugger can be extended to also allow editing of debugged data structures. Conversely, editors of various objects such as text, forms, spreadsheets, and programming languages can be extended to also support incremental programming.

Summary

In this paper, we have explored the notion of integrating editing in a monolingual environment. We have compared editing with interactive programming, and presented an approach for integrating the two modes of interaction. Our approach allows mixing of editing and interactive programming commands by supporting interact

and edit windows, an asynchronous **edit** construct, attributes, and the *composed* event. It also bridges the gap between the two modes by proposing in the interactive mode the notion of composing a value before committing it, and commands to undo and redo changes, and in the editing mode the notion of invoking user-defined functions declared in the programming language. Our approach can also be used to integrate editing and interactive programming in individual applications of a multilingual environment.

REFERENCES

[1] Vincenzo Ambriola and David Notkin, "Reasoning about Interactive Systems," *IEEE Transactions on Software Engineering* 14:2 (February 1988).

[2] Prasun Dewan, "Automatic Generation of User Interfaces," Ph.D. Thesis and Computer Sciences Technical Report #666, University of Wisconsin-Madison, September 1986.

[3] Prasun Dewan and Marvin Solomon, "Dost: An Environment to Support Automatic Generation of User Interfaces," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices* 22:1 (January 1987), pp. 150-159.

[4] Christopher W. Fraser, "A Generalized Text Editor," *CACM* 23:3 (March 1980), pp. 154-158.

[5] C.W. Fraser and D.R. Hanson, "A High-Level Programming and Command Language," *Sigplan Notices : Proc. of the Sigplan '83 Symp. on Prog. Lang. Issues in Software Systems* 18:6 (June 1983), pp. 212-219.

[6] C.W. Fraser and D.R. Hanson, "High-Level Language Facilities for Low-Level Services," *Conference Record of POPL*, 1985, pp. 217-224.

[7] Adele Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, Mass., 1984.

[8] D.R. Hanson, "Event Associations in SNOBOL4 for Program Debugging," *Soft. Pract. Exper.* 8 (1978), pp. 115-129.

[9] Jan Heering and Paul Klint, "Towards Monolingual Programming Environments," *TOPLAS* 7:2 (April 1985).

[10] Barbara Liskov and Robert Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *Proceedings of the 9th POPL*, 1982, pp. 7-19.

[11] Robin Milner, "The Standard ML Core Language," *Polymorphism* 2:2 (October 1985).

[12] David Notkin, "Interactive Structure-Oriented Computing," PhD Thesis and Technical Report, CMU-CS-84-103, Department of Computer Science, Carnegie-Mellon University, February 1984.

[13] Jeffrey Scofield, "Editing as a Paradigm for User Interaction," Ph.D. Thesis and Technical Report No. 85-08-10, University of Washington, Department of Computer Science, August 1985.

[14] M. Shaw, E. Borison, M. Horowitz, T. Lane, D. Nichols, and R. Pausch, "Descartes: A Programming-Language Approach to Interactive Display Interfaces," *Sigplan Notices : Proc. of the Sigplan '83 Symp. on Prog. Lang. Issues in Software Systems* 18:6 (June 1983), pp. 100-111.

[15] Warren Teiltelman and Larry Masinter, "The Interlisp Programming Environment," *Computer*, April 81, pp. 25-32.