

PENGUIN: A Language for Reactive Graphical User Interface Programming

Sue-Ken Yap and Michael L. Scott

Department of Computer Science, University of Rochester, Rochester, NY 14627

ken@cs.rochester.edu, scott@cs.rochester.edu

PENGUIN is a grammar-based language for programming graphical user interfaces. Code for each thread of control in a multi-threaded application is confined to its own *module*, promoting modularity and reuse of code. Networks of PENGUIN *components* (each composed of an arbitrary number of modules) can be used to construct large reactive systems with parallel execution, internal protection boundaries, and plug-compatible communication interfaces. We argue that the PENGUIN building-block approach constitutes a more appropriate framework for user interface programming than the traditional Seeheim Model. We discuss the design of PENGUIN and relate our experiences with applications.

1 Introduction

Graphical user interfaces are an essential part of current programming environments. Graphical windowing systems such as X [1] have become widely available. Unfortunately, programming tools for composing interfaces have not improved commensurately.

1.1 Event-driven programming

Windowing applications commonly use the “big loop and case statement” technique to dispatch incoming events. As a concrete example, the code for a document previewer will typically look like this:

```
display a page
loop
  get an input event
  case event of
    keystroke:
      /**/
    mouse button:
      do user command
    repaint signal:
      repaint window
  end case
end loop
```

In this program organization unrelated streams of input flow through a common dispatch point, adversely affecting modularity and the ease of revision of the interface. Suppose, for example, that the programmer decides to augment the previewer to allow the user to jump to page N. A page number is a string of digits, so digit keystrokes must be collected. The collection of keystrokes can occur within a single branch of the case statement, but then the window will be insensitive to repaint signals until the number has been entered in its entirety. Alternatively, the digits can be collected one at a time, in successive iterations of the main loop, but only at the cost of declaring global variables to retain state between digits.

Some windowing libraries use another technique: callback routines. Semantic actions need not be embedded in an explicit loop, but the program has to register callback routines with an

event dispatcher at initialization time. Like the digit-at-a-time approach above, this method has the drawback of requiring routines to maintain explicit, self-contained state between events. It also complicates the handling of unexpected or exceptional events.

Similar problems arise if the programmer wishes the previewer to be sensitive to user interrupts while painting the page. There is no easy way to integrate the reading of individual document characters from the file system into the event loop or callback routines without adversely affecting the clarity of the code. System-specific interrupt handling can be used as a work-around, but will lower the portability of the previewer.

In both cases, the crucial observation is that the polling and callback methods fail to reflect the logical structure of multi-threaded interfaces. Both methods force the programmer to deal with events in isolation, despite the fact that most interesting computations comprise a series of events. Because they are designed for ordinary sequential language, both methods must explicitly mirror the potential interleaving of unrelated events.

To overcome the limitations of the polling or callback techniques, we propose a programming language that supports event-driven programming. Rather than dictating when input is expected, we suggest that programs be *reactive*. The resulting change of perspective can lead to a clearer programs.

In PENGUIN, the sequencing of input events is expressed by grammars. PENGUIN does not have input statements or input procedures. The appearance of a terminal in a grammar indicates that the program is willing to accept that terminal as input in the context of the surrounding symbols. Grammars are contained in *modules* and the composition of modules is a module that is sensitive to the input specified by union of all the grammars, taking into account the *context* (source) of the input. This decentralized module-by-module approach to input specification makes it easy to modify input syntax. Modules are also managers of private data, distinct in each instance of a module. As a result, the semantics of modules are local, decreasing the risk of inadvertent interaction between unrelated segments of code.

Previous event-driven languages include Esterel [2] and Input Tools [3, 4]. Esterel allows event specifications to be compiled into an automaton. Piecewise construction of larger programs is not supported; the automaton generated is global to the entire program. Communication is by broadcasting. Input Tools [3, 4] allows programs to be composed hierarchically, with low-level tools accepting input, processing it, and propagating information up-

wards to higher-level tools. There is no provision for input selection based on the source of an event, so broadcasting of events is still required. Experience with an implementation of Input Tools [5] suggests that this broadcasting is a serious source of inefficiency, particularly for large systems.

PENGUIN encourages modular construction, separates the grammatical specification of input sequencing from the bulk of the program code, and does not require broadcasting of events.

1.2 Composing applications

Another deficiency of current graphical user interfaces is the difficulty of composing graphical applications from free-standing, pre-existing pieces. Experience with pipes in the Unix operating system¹ shows that it is possible to build stream-based process communication mechanisms that are extremely easy to use. It has been suggested [6] that similar support be provided for non-linear process graphs as well, but even the linear variety has yet to be heavily used in user interface design. While most Unix programs can participate in multi-process combinations, the average graphical program is a free-standing entity, not easily connected to other programs.

We propose the PENGUIN *component* model for the composition of multi-component programs. A PENGUIN component is a set of modules linked with a parser for their grammars. A component can be free-standing or can be connected to other components in a general communication graph. Easy composition of components with compatible interfaces encourages code re-use, rapid prototyping, and the construction of flexible, general-purpose tools.

Two systems that explored facilities for the composition of graphical programs are ConMan [7] and Fabrik [8]. ConMan is a high-level visual language that allows the user to build a complex application from components on the fly. Its primary goal is the manipulation of graphical images. Its components are programs that transform or display data. Fabrik is a similar system for experimenting with visual programming, but its components are interactors or computational modules. Both systems define specialized environments.

PENGUIN provides a formal, generalized model to describe intercomponent connections. Since PENGUIN components are reactive, the composition of components can achieve more than data transformations; it can also specify the interactive behaviour of a system of interconnected objects.

The remainder of this paper describes how the design of PENGUIN achieves the goal of making user interfaces easier to build and easier to understand.

2 Language Overview

This section provides a quick overview of the PENGUIN language. PENGUIN's compilation units, *modules*, are organized around augmented context-free grammars. The design decisions taken and algorithms used by PENGUIN have already been described[9]. A PENGUIN implementation consists of a compiler that translates grammars and their associated data declarations and action routines into executable program components. Data and actions are written in a host language (currently C++) of which PENGUIN is an extension. The output of the PENGUIN compiler is a program in the host language without extensions.

The most noticeable difference between programming a user interface in PENGUIN and a conventional language is that the spurious juxtaposition of unrelated threads of execution introduced

by the event loop model disappears. The programmer only needs to consider the sequencing *within* a thread of control.

2.1 Forks

The productions of a PENGUIN grammar specify valid sequences of terminals that may be received by the grammar's module. Terminals are matched by input events, following *context* matching rules, and may carry information from the outside world via *attributes*. Input events encompass more than data received by input statements in conventional languages; they also include asynchronous signals and exceptions, which are difficult to handle in a non-reactive language.

Multi-threaded execution in PENGUIN programs is achieved with *fork* productions. There are two types of fork productions: the AND fork and the OR fork. A variety of useful behaviour can be synthesized with these two variants. A fork creates one or more *subparsers*, which are disjoint, concurrent regions of parser activity. A module may be thus willing to accept terminals from multiple sources. Moreover terminals from independent sources may be accepted by a PENGUIN program in arbitrary order, without the need to accommodate their interleaving in user-written code.

```
tool &> canvas panel;
run |> work abort;
```

In the first example the AND fork (specified with a *&>* derivation symbol instead of the usual *=>*) requires that all the component windows start running in parallel and that all of them complete before the parent advances past the fork. In grammar terms, the yield of the non-terminal *tool* is some arbitrary interleaving of the yields of the non-terminals *canvas* and *panel*. In procedural terms, a subparser for *canvas* and a subparser for *panel* begin execution in parallel; when they complete, the subparser for *tool* can continue.

In the second example the OR fork (specified with *|>*) requires that only one of *work* or *abort* complete. Specifically, hitting the abort button will cancel all work in progress in the sibling window, returning the locus of control to the parent production. Completing *work* will disable the subparser for *abort*. By nesting abort productions, the programmer can allow the user to back out of multiple levels of interaction.

A PENGUIN program is thus driven from below by the arrival of events that match terminals, and from above by the creation of threads of control by fork productions. Forks are a linguistic mechanism for domesticating threads for use in user interface programs. The PENGUIN compiler handles the messy thread management which would otherwise have had to be coded by the programmer.

2.2 Context

The presence of more than one active production requires a mechanism for dispatching events. We stipulate that every input event be tagged with *context*, an attribute that uniquely identifies its source. In a typical windowing system, the source of an event would be a window, but alternative interpretations are equally valid. In a flight simulator, context attributes could be used to distinguish between mechanically similar flight controls. The PENGUIN parser contains a dispatcher that delivers incoming events to productions in which they match both in value and in context.

Here is a simple dialogue that awaits the key *k* and then creates two new sub-windows with an *&>* production. Both sub-windows

must terminate before the parser proceeds. Inheritance rules are written using a notation similar to argument passing in imperative languages: `Y(@X.c1)` means that the first attribute of `Y` is copied from the `c1` attribute of `X`. In practice, default context rules eliminate much of this verbiage.

```
terminal key(context ctx) = 'k';
nonterm S(context ctx), X(context c1, context c2);
nonterm Y(context c), Z(context c);
nonterm new(|context ctx1, context ctx2);
...
S => key(@S.ctx) new X(@new.ctx1,@new.ctx2) ...;
X &> Y(@X.c1) Z(@X.c2);
```

The non-terminal `new`, whose sub-productions are not shown, causes the creation of two new windows and returns their contexts. These contexts are passed to `Y` and `Z`, so that they can operate independently, even if their token alphabets overlap.

Attribute flow rules for context, enforced by the PENGUIN compiler, ensure that at most one production will accept an incoming event. The compiler can compute in advance the locations that the parser must examine to find a match. As in regular attribute grammars, attributes other than context carry information about the symbol to which they are attached.

2.3 Modules

The unit of compilation in PENGUIN is the module. A module contains a header, declarations, private variables, and a grammar. PENGUIN does not allow the programmer to write a main program. The locus of control is retained by the PENGUIN parser which is the dispatcher of input events. The PENGUIN parser is the heart of an *executing* interactive program. It should not be confused with the portion of PENGUIN compiler that parses PENGUIN source. The programmer provides modules to be linked with the PENGUIN parser. The parser initially predicts a non-terminal of the topmost module. The chain of predictions eventually will require one or more terminals to be consumed.

Declarations inform the compiler of the types of attributes of symbols in the grammar. Attributes transmit information between productions. Inherited attributes pass information to descendants while synthesized attributes return information to ancestors. Attributes are attached to all symbols: terminals and non-terminals.

```
terminal lparen = '(', rparen = ')';
nonterminal S(int i|int j);
```

These declarations state that the value of the terminal `lparen` is the character code for '(' and that the nonterminal `S` carries an inherited attribute `i` and a synthesized attribute `j`.

Productions may refer to non-terminals in other modules (externals). Since there may be more than one instance of an external module active, an external non-terminal is qualified with a module handle. A handle is initialized by predicting its `create` symbol (which every module must provide).

```
module canvas C;
module panel P;
...
create => C:create P:create;
tool &> C:start P:start;
```

Private data are local to each instance of a module. They provide a repository for the shared state of related events, independent of the rest of the program. The current version of the PENGUIN compiler does not check type declarations, so the data types available are those provided in the target language.

Changes to the outside world are effected via *actions*, which are symbols in productions representing executable code. The code is executed when the PENGUIN parser reaches the position of the action while recognizing its production.

```
nonterminal click(color newc);
...
click => left_button
      $( change_color(@click.newc);
        output("OK"); $);
```

In the code fragment above, an action to change the color of the button is executed after the event `left_button` has been received. As the example shows, actions may use inherited attributes, as with other items.

Ideally, actions that are triggered by events would take no noticeable time. In practice an input buffer smooths out the effect of delays in actions. As long as the input is ordered by time and buffer overrun does not occur, a PENGUIN program does not lose any input information. Communication between parallel PENGUIN *components* (discussed in the following section) provides a more general solution for programming actions that may introduce arbitrary delays.

3 Components

A set of modules linked with a PENGUIN parser is a *component*. A component may be a free-standing program or may co-operate with other components. The synergy of co-operating components makes it easy to adapt existing tools to novel applications.

PENGUIN components communicate with other components via bidirectional reliable data streams. The unit of information exchanged is the token or terminal. Each token is a complete piece of information and can be as small as a single character or as large as a picture—whatever is appropriate for the application.

Figure 1 is an example of components co-operating to implement a game-playing program. The Chess component controls the reaction of the application to user input, received via the presentation component. A second, parallel presentation component provides another display of the game in progress. Since move generation and evaluation may require substantial amounts of computation, some work is given to parallel components to preserve the interactiveness of the user interface. This is an example of a general technique that can be applied to any computation that may delay the flow of events through the grammar: do the work in a separate component and devise a protocol for sending data to, receiving results from, and querying the status of the sibling component. In response to queries the user interface might say "Thinking" if the move generator is still busy.

Two features of PENGUIN provide flexibility in creating networks of components. First, communication partners need not know the identity or implementation details of their peers. Second, the connection graph of components does not have to be fixed until run time, allowing alternative configurations to be created. Tokens are addressed to output streams, not to named components, and the connections between components can be established under program control. The only requirement for two components to be able to communicate is that each send only

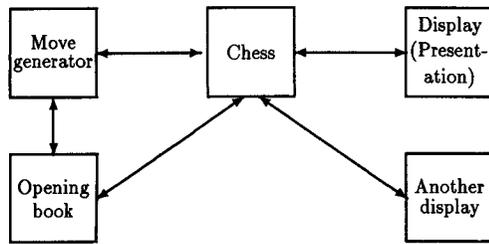


Figure 1: A multi-component Chess program.

tokens that are in the other's input alphabet. The behaviour of a component is determined by the grammar within. Alternative implementations of components can be produced to meet a functional specification. Interchangeable components are useful for selecting between different versions of components or for producing different effects. The example above, for instance, could be configured with a variety of different move generators or opening books. A logging component could also be interposed between the Chess and Display components, to record the entire session.

4 The Seeheim Model Reconsidered



Figure 2: The Seeheim model of a UIMS.

In the original Seeheim model [10], depicted in Figure 2, a program is postulated to have three layers: application, dialogue and presentation. These layers correspond to division by semantic, syntactic and lexical functions. The Seeheim model assumes that a separation between syntax and semantics provides the most natural partition of a graphical application. The fallacy of this assumption can be seen in our game-playing example. A chess program must examine the positions of other pieces on the board to discover whether it is legal to move a king. One can imagine an implementation that allows the user, syntactically, to make an illegal move, but an interface that provides immediate feedback, indicating that the move is not allowed, is far preferable to one that accepts the illegal move, displays it, and then forces it to be rescinded after performing semantic checking. Immediate feedback requires a coupling between presentation and application not accommodated by the Seeheim model.

Separating the application from the input device through a long communication path creates long feedback loops that diminish both performance and conceptual clarity. Separating lexical checking from semantic checking works well in non-interactive programs such as compilers. It can be extended to simple devices such as buttons that have no significant semantic component and require little feedback. It does not generalize well to non-trivial interactive programs. Artificial distinctions between feedback based on the state of a logical input device and feedback based on the state of the application serve only to complicate programs. Both cases can be modeled as actions that test an internal resource. The only difference lies in the sophistication of testing.

The specification of input syntax should be separated from the specification of computation semantics, but this separation is not an appropriate basis for division into components. Syntax and

semantics are too tightly bound to be placed in separate components. Components that have been segregated by lexical, syntactic or semantic functions have *logical cohesion* or, at best, *communicational cohesion*[11]. Both these types of cohesion are weaker than the *functional cohesion* exhibited by PENGUIN components.

The Seeheim model envisions a syntactic and a semantic component, both of which encompass all of the functional tasks of the application. The PENGUIN model assigns each functional task to a separate component that includes both syntax and semantics. Seen in this framework, a presentation manager is more than just the lexical front-end of the Seeheim model; it is a first-class component with internal resources and syntax (protocol). A window server such as X can be built as a PENGUIN component, and clients should be structured as components too, rather than as a completely different kind of program.

5 Experiences with PENGUIN

We have constructed a PENGUIN compiler and used it to construct reactive applications. We discuss two of these applications here.

5.1 pfig: A graphics editor

The first application is an adaptation of an existing line graphics editor called `xfig` comprising of about 15000 lines of code, of which about 2500 are concerned with the user interface. The editor interface comprises a window subdivided into a canvas area, a panel of buttons, a message window and rulers at the edges. The window hierarchy is:

```

xfig
  canvas
  panel
    buttons
  message
  rulers
  
```

The events that need to be handled by each window include mouse clicks, key clicks and exposure notifications. The parent window creates instances of each of the second level modules and predicts their `create` symbols. The panel module then creates buttons. The terminals generated by the windows have distinct contexts so the parser is able to dispatch all events to the correct thread.

Measurements indicate that about 700 lines of PENGUIN grammar and code expanded to 2600 lines of C++ code, which replace 2500 lines of old C code. We estimate that compiler generated C++ code is twice as bulky as good human generated C++ code, i.e. a human would have had to write about 1300 lines of C++ to replace the 2500 lines of C code. There are 102 productions in total of which 14 are `&>` productions. There are no `|>` productions, probably because exceptional conditions were already handled by the old C code. A similar editor created from scratch might profitably use `|>` productions in some places. All of the `&>` productions are used to create parallel threads of control, none of which expect to complete.

5.2 alarm: Alarm clock

Our second application is a simulated alarm clock with time and alarm displays, and buttons to set the display mode and the time

of the alarm. This application demonstrates the handling of signals as events. The window hierarchy is simple:

```

wrapper
  clock
  alarm
  mode.button
  set.button

```

A run time library timer module sets up a handler for Unix alarm signals. When the operating system delivers an alarm signal, the library translates it into a PENGUIN event. Since Unix signals carry no information other than the fact that they occurred, an auxiliary queue is used to store a list of pending timer events. The library obtains the context and value of the token to create from the queue when a signal occurs.

The clock time and alarm time subwindows respond to exposure events, so they will redisplay the time when the window first appears or when it is unobscured. The clock subwindow schedules a timer event once a minute to update the digits. The alarm subwindow schedules a timer event for the time at which the clock is due to beep.

Because the alarm clock was coded from scratch, there is no old implementation to compare against. Some indication of the degree of programming help provided by the compiler can be seen from these statistics: some 650 lines of PENGUIN code generated about 2800 lines of C++ code. Another 550 lines of auxiliary routines in C++ were needed. The run time library is identical to that used by `pfig`, except for the addition of the timer module. There are some 50 productions, of which 4 are forks. Our subjective impression is that the clock would have been significantly harder to write without PENGUIN. Now that the timer code has been added to the run time library, future programs requiring timer events would be even easier to write.

5.3 Evaluation

Writing modules to react only to events that concern it and letting PENGUIN compose the modules and correctly dispatch events is pleasantly natural. The distinction between module templates and module instances is important. For example, there is only one copy of the code for a graphical button, but as many instances as there are buttons in a panel, each instance customized via attributes.

PENGUIN makes possible an interesting technique for intermodule communication we call "event forwarding." Here is an example of how this works: A panel comprises buttons. Each button may trigger a different action routine when clicked upon. Since the code template for the button is common to all buttons, it is not appropriate to put the call to the action routine in the button code. There are two traditional ways to deal with this situation. The first method is to give each button a unique index and have the code use the index to find the appropriate action. This requires a case statement or a global array somewhere. The second method is to pass a pointer to an action routine to each button (callbacks). PENGUIN has a third method: take the input event, change the context to one expected by the recipient and forward the event. In this example the buttons forward the event to the parent panel. Since the button and panel are part of the same component, the forwarding amounts to putting the modified token back on the input stream, where the parser will deliver it to a different module. As far as the panel is concerned, a click event has happened at its window, even though the presentation does

not deliver any.

The alarm clock program provides another example. The button for setting the alarm time forwards button events so that they appear to come from the alarm display subwindow. The code for setting the wakeup time and the variable holding the time can be kept local to the alarm module.

It is not necessary for modules to have a parent/child relationship for forwarding to work. The pointers on the top and side rulers of `pfig` track the movement of the cursor in the canvas window. The canvas receives cursor motion events, uses them to update its display (and its internal state), and forwards them to both of the rulers, even though the canvas is a sibling of the rulers. Forwarding can also be used profitably in constructing a composite module comprising a viewport and a scrollbar. All actions on the scrollbar can be made to appear to happen in the viewport, simplifying the programming and preserving the modularity of the composite window. The PENGUIN forwarding mechanism is more powerful than the forwarding strategy of windowing systems such as X11, which can forward only to ancestor (enclosing) windows, unless extraordinary and inconvenient arrangements are made.

Finally, because all events go through the parser, it is possible in PENGUIN to isolate windowing system dependencies in the run time library. Details such as the layout of input event structures can be confined to a few routines that turn window system functions into events. Different run time libraries can interface to different window systems. The cost of porting a program to a different windowing environment will be smaller than if the program had been written for a single environment because fewer idiosyncrasies will have been introduced and because the labour of adapting the run time library can be amortized over many applications.

6 Implementation

The PENGUIN compiler was written with the help of `flex`, `bison`, and `g++`, the GNU (Free Software Foundation) C++ compiler. The PENGUIN compiler converts a module grammar into code and a set of tables.

The multithreading of forked productions can be implemented with a coroutine package or with interpretive code. In the former method, the compiler generates a conventional sequential parser (e.g. recursive descent) for each PENGUIN subparser. One subroutine is used for each production, non-terminal or start symbol. Terminals are translated to calls to a run time routine that fetches an event. Prediction points are translated to calls that peek at the next event and use it to decide which production routine to call. Fork productions are translated to calls that create new threads. A coroutine library switches between the threads of a module, giving each thread control when an event acceptable to it arrives. Some subtlety is required to re-join threads at the end of a fork production: the first OR branch or last AND branch must clean up all its siblings.

In the latter, interpretive method of parser construction, the parser runs an extension of the normal table-driven predictive parsing algorithm. It builds a branching (cactus) stack to represent parallel productions. When a production is predicted, one branch of the cactus is extended. As terminals are matched, the appropriate branch is trimmed. Forks and joins change the number of branches.

We originally planned in our implementation to use the interpretive method, but eventually switched to coroutines. Both methods are feasible but the coroutine method simplifies attribute passing. In the interpretive method, either the set of attribute types must

be restricted to those known to the PENGUIN compiler, or calls to external code must be made to copy attributes. In the coroutine method, the PENGUIN compiler can generate appropriate assignment statements and let the host language compiler implement type-specific copying. The coroutine method has an additional advantage when the amount of inline code is large in comparison to the number of symbols in the grammar. The interpretive method turns every piece of inline code into a subroutine and devotes much time to linkage overhead. The coroutine method can execute inline code in line.

Prediction tables for either the interpretive or coroutine approaches are simply initialized data. They are shared between all instances of a module. Private variables are unique to every module instance. In our current host language, C++[12], private variables have a natural expression as class members. Slightly less attractive translations could be found for other languages.

7 Conclusions

Programs have traditionally been viewed as data transformers. They read their input, compute a function, write their output, and terminate. Programs that run indefinitely (operating systems for example) have been seen as exceptional cases. Programs with graphical interfaces, however, are representative of a growing class of applications with a heavy interactive component. Real-time process control is also in this class. For these applications the sequencing of acceptable input is just as important, and just as complicated, as the operations on that input.

Mainstream programming languages have no facilities for expressing this sequencing beyond the standard control flow operators. Pseudo-parallelism may be obtained by calling system dependent operations to poll for data, but the underlying procedural execution model makes the resulting code obscure and non-portable. It can be difficult to ascertain the effects of a given input sequence. This in turn makes it difficult to ensure that all valid input sequences are covered, that invalid sequences are rejected, that appropriate actions are taken in every case, and that resources are recovered when no longer in use, a particularly important consideration for long-lived programs. PENGUIN goes to the heart of the problem by providing a notation that better matches the programming tasks confronted in graphical user interfaces. This notation frees the programmer from thinking in sequential terms and instead encourages the decomposition of an interface into self-contained modules with well-defined entry points triggered by the arrival of input events. Reasoning about the effects of input sequences and the life history of resources becomes much easier.

Our experience with PENGUIN suggest that its reactive execution model, its separation of dialogue and computation, and its automatic dispatch of tokens can make complicated reactive programs easier to write, easier to read, and easier to debug and maintain. PENGUIN's component model provides a basis for building applications incrementally as a collection of co-operating components, and encourages the reuse of existing components in new combinations.

Notes

1. Unix is a trademark of AT&T Bell Laboratories.

References

- [1] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, 6(2), April 1987.
- [2] G. Berry, P. Couronne, and G. A. Gonthier. Synchronous programming of reactive systems: an introduction to Esterel. Technical Report 647, INRIA, March 1987.
- [3] Jan van den Bos. Input tools – a new language construct for input-driven programs. In *Proceedings of the European Conference on Applied Information Technology of IFIP*, September 1979.
- [4] Jan van den Bos. Abstract interaction tools: A language for user interface management systems. *ACM Transactions on Programming Languages and Systems*, 10(2):215–247, April 1988.
- [5] J. Matthys. Recent experiences with input handling at PMA. In *User Interface Management Systems*. Springer-Verlag, 1985.
- [6] Chris McDonald and Trevor I. Dix. Support for graphs of processes in a command interpreter. *Software Practice and Experience*, 18(10):1011–1016, October 1988.
- [7] Paul E. Haeberli. ConMan: A visual programming language for interactive graphics. In *SIGGRAPH '88 Conference Proceedings*, pages 103–111, August 1988.
- [8] Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph, and Ken Doyle. Fabrik: A visual programming environment. In *OOPSLA '88 Conference Proceedings*, pages 176–190, September 1988.
- [9] Michael L. Scott and Sue-Ken Yap. A grammar-based approach to the automatic generation of user-interface dialogues. In *CHI '88 Conference Proceedings*, pages 73–78, May 1988.
- [10] Günther E. Pfaff, editor. *User Interface Management Systems*. Springer-Verlag, 1985.
- [11] Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, 1979.
- [12] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.