The Architecture of *Pan I*

Robert A. Ballance[1]
Michael L. Van De Vanter
Susan L. Graham

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California
Berkeley, California 94720

September, 1986
Revised Summer, 1987

PIPER *Working Paper 87-4*

Abstract

Pan is a prototype and testbed for language-based editors and viewers. Its design addresses the needs of experienced users who manage complex objects such as large software systems. All of *Pan*'s components are multi-lingual, incremental, description-driven, customizable, and extensible. Viewing is facilitated by semantics-based browsing and an object model which integrates text and structure. *Pan* is intended to share information with other tools, allowing integration into a larger language, program, and document development environment.

This document describes the internal design of *Pan I* the current implementation of *Pan*. It begins by reviewing goals that motivated the project and *Pan*'s particular approach to those goals. The body of the document describes the implementation from five successive points of view: implementation layers, functional components, basic objects, services provided to the user, and the thread of control in the running system.

[1]Sponsored by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871 (monitored by Space and Naval Warfare Systems Command under Contract No. N00039-84-C-0089), by IBM under IBM Research Contract No. 564516, and by the State of California MICRO program. Robert A. Ballance and Michael L. Van De Vanter were supported in part by MICRO fellowships.

Contents

1 Introduction

Pan[2][2] is a multilingual language-based editing and browsing system under development at the University of California, Berkeley.

As a front-end to an integrated development environment, *Pan* must be able to analyze programs and other documents and to share the information gathered with other tools. Algorithms and techniques appropriate for stand-alone systems are not sufficient in this context.

The design of *Pan* addresses several goals:

- To provide both text- and structure-oriented manipulations while hiding from users the internal representation of the underlying structure.
- To retain information for other tools.
- To provide as much feedback to the user as possible as soon as possible.
- To support the experienced user.
- To handle multiple production languages.
- To provide a test-bed for user-interface designs.

1.1 Design Motifs

In the final analysis, it is *how* any system supports these goals that distinguishes it from other systems. The design of *Pan* has been guided by the following principles:

Present the logical structure of documents

Users should be able to interact with a language-based editor in the terms and concepts of the language being edited. Thus, for a programming language having modules, functions, statements, and expressions, the user should be able to select and operate upon modules, functions, statements, and expressions. Text editors only provide operations on a stream of characters. Structure-oriented editors, on the other hand, maintain an internal representation of the document's structure within which internal nodes represent the language's substructures. Structure-oriented editors, however, do not interpret those nodes, requiring the user to use structure-oriented commands (e.g., Left, Right, In, Out, Delete-Node) rather than more natural language-oriented commands (e.g., Next-Function, Previous-Declaration, Delete-Statement).

Pan provides language-oriented, rather than structure-oriented, commands, while hiding internal representations from the user.

[2]Why "Pan"? In the Greek pantheon, Pan is the god of trees and forests. Also, the prefix "pan-" connotes "applying to all"—in this instance referring to the multilingual text- and structure-oriented approach adopted by this project. Finally, since an editor is one of the most frequently used tools in a programmer's tool box, the allusion to the lowly, ubiquitous kitchen utensil seems apt.

Integrate text- and structure-oriented editing

It is important that users be able to interact with a document either as text or as a structured object. First, documents, especially programs, have both text-like and structural aspects. For instance, a user may replace textually all occurrences of a particular name, both where it is used as an identifier and where it appears in comments. However, the user may also wish to make the change based on the structure of a program, for instance renaming a variable within a certain scope. In the second case, variables having the same name in nested scopes would not change.

Second, most kinds of documents contain unstructured text, for example, the sentences in a memo or the comments in a program.

Third, there are times when a user wants to transform substantially the structure of a document, but wishes to do so efficiently by altering the document's textual representation. For instance, a user may wish to substantially rewrite and reorganize a document, re-using existing pieces that cross structural boundaries.

Use incremental LR parsing to detect syntactic structures

New parsing and tree-building algorithms have been developed for *Pan*[3], based on a newly defined formal relationship between the abstract syntax of a language and its concrete (parsing) syntax. This correspondence allows a decorated abstract syntax tree to be used during incremental LR parsing.

Model contextual constraint checking on logic programming

Semantic checking in *Pan* is modeled on logic programming. Structure-independent clauses establish the axioms of the semantic definition. Constraints associated with syntactic structures are used to state and implement the context-sensitive aspects of the source language. The information gathered during checking is retained in a logic database that can be accessed by queries written in the logic-based specification language.

Creation and use of the database are intertwined; certain conditions must be true before other facts can legitimately be added. The research involved includes the application of logic programming to static semantics and the search for efficient approaches to constraint satisfaction in the presence of a changing database.

Create description-driven components

Components such as the parser, the semantic checker, and the tree-presentation modules of *Pan* are description-driven. This supports declarative specifications while enabling component sharing.

Support multiple languages

Experienced programmers typically use several formal languages in their daily activities—a design or specification language, a structured-documentation language, and one or more programming languages. It is essential that a single editing and browsing system support all of these tasks. *Pan* is intrinsically multilingual, since adding a new language consists of writing the description and then loading data tables.

Provide semantics-driven browsing and presentations

The information gathered by the semantic analyzer can be used by other components of the environment—especially the components used to display and browse a complicated structure. The semantic information is itself a structure that *Pan* will be capable of displaying and browsing.

Develop a rich model of annotation

Semantic constraints can be regarded as one form of annotation; program commentary is another. *Pan* supports a rich model of annotation using the syntactic and semantic mechanisms already described. Support for linguistic conventions (such as project-specific naming conventions), programming idioms, styles, and standards, for extensible semantics, and for tree-based pattern matching will be provided in order to further extend this model.

1.2 Implementation Status

Pan I[4] is the currently implemented version of *Pan*. It can be used to edit text and tree-structured documents, although development has focused primarily on experiments in editing programming languages. It includes incremental parsing; a prototype of the semantic description analyzer and checker is under development. *Pan I* runs under UNIX[3] on the Sun Workstation[4] using the SunView[1] windowing system.

Pan II is now being designed, based on experience with the *Pan I* prototype. Major goals include more advanced presentation and browsing facilities, a more integrated model of text and structure editing, and the ability to run as a program-driven tool. The implementation will use the X window system[7]. *Pan II* will serve as the basis for integration into a larger language, program, and document development environment.

For the remainder of this document, the term "*Pan*" refers to *Pan I*.

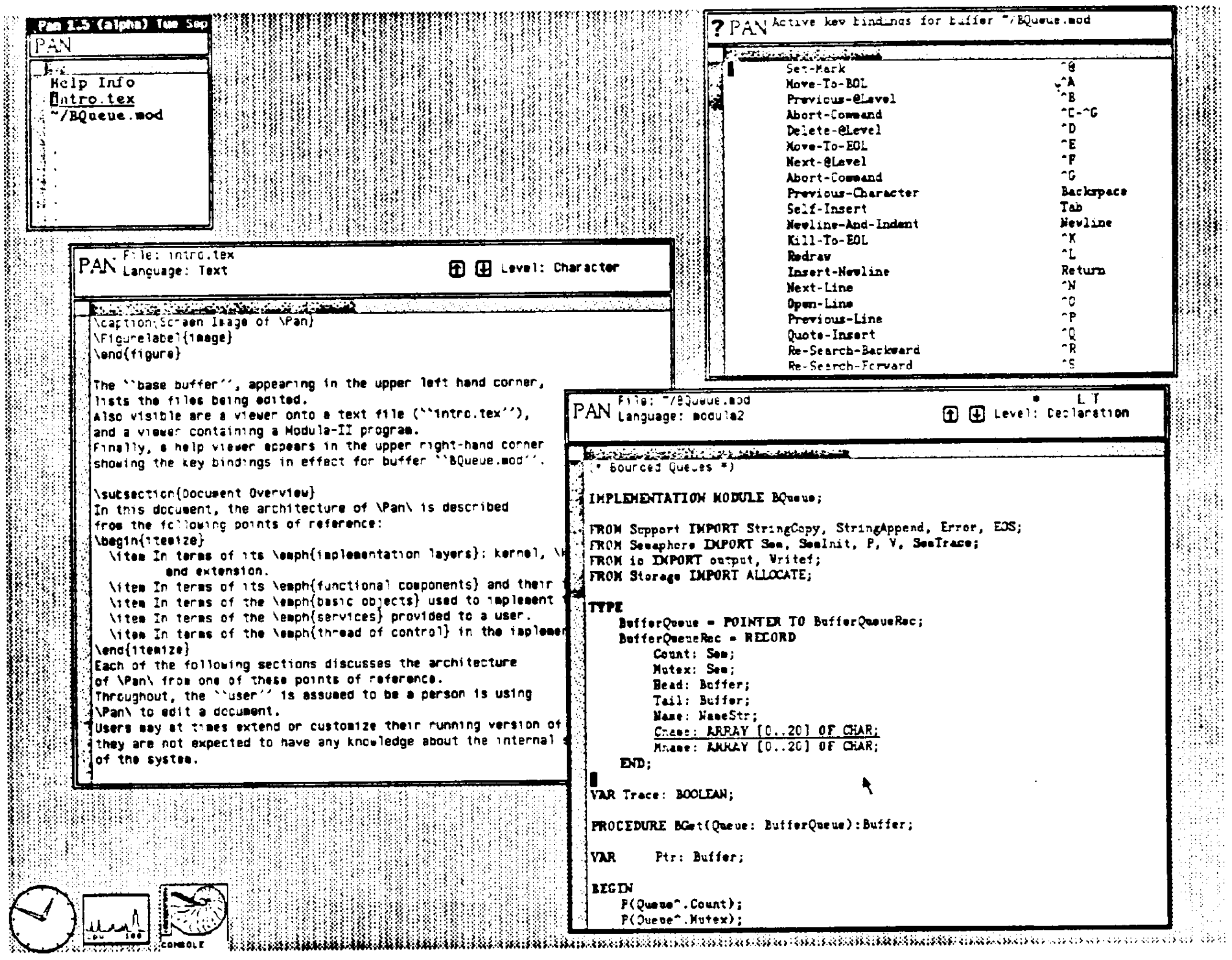
Figure 1 shows the screen of a workstation running *Pan*. The "base buffer", appearing in the upper left hand corner, lists the files being edited. Also visible are a viewer onto a text file ("intro.tex"), and a viewer containing a Modula-II program. Finally, a help viewer appears in the upper right-hand corner showing the key bindings in effect for buffer "BQueue.mod".

1.3 Document Overview

In this document, the architecture of *Pan* is described from the following points of reference:

- In terms of its *implementation layers*: kernel, language-primitive, and extension.
- In terms of its *functional components* and their interdependencies.
- In terms of the *basic objects* used to implement the system.
- In terms of the *services* provided to a user.
- In terms of the *thread of control* in the implementation.

[3]UNIX is a registered trademark of AT&T Bell Laboratories
[4]Sun Workstation, and SunView are registered trademarks of Sun Microsystems, Inc.



```
Help Info
intro.tex
~/BQueue.mod
```

```
File: intro.tex
Language: Text                                   Level: Character

\caption{Screen Image of \Pan}
\Figurelabel{image}
\end{figure}

The ``base buffer'', appearing in the upper left hand corner,
lists the files being edited.
Also visible are a viewer onto a text file (``intro.tex''),
and a viewer containing a Modula-II program.
Finally, a help viewer appears in the upper right-hand corner
showing the key bindings in effect for buffer ``BQueue.mod''.

\subsection{Document Overview}
In this document, the architecture of \Pan\ is described
from the following points of reference:
\begin{itemize}
  \item In terms of its \emph{implementation layers}: kernel, \
        and extension.
  \item In terms of its \emph{functional components} and their
  \item In terms of the \emph{basic objects} used to implement
  \item In terms of the \emph{services} provided to a user.
  \item In terms of the \emph{thread of control} in the implemen
\end{itemize}
Each of the following sections discusses the architecture
of \Pan\ from one of these points of reference.
Throughout, the ``user'' is assumed to be a person is using
\Pan\ to edit a document.
Users may at times extend or customize their running version of
they are not expected to have any knowledge about the internal
of the system.
```

```
Active key bindings for buffer ~/BQueue.mod

     Set-Mark                        ^@
     Move-To-BOL                     ^A
     Previous-@Level                 ^B
     Abort-Command                   ^C-^G
     Delete-@Level                   ^D
     Move-To-EOL                     ^E
     Next-@Level                     ^F
     Abort-Command                   ^G
     Previous-Character              Backspace
     Self-Insert                     Tab
     Newline-And-Indent              Newline
     Kill-To-EOL                     ^K
     Redraw                          ^L
     Insert-Newline                  Return
     Next-Line                       ^N
     Open-Line                       ^O
     Previous-Line                   ^P
     Quote-Insert                    ^Q
     Re-Search-Backward              ^R
     Re-Search-Forward               ^S
```

```
File: ~/BQueue.mod                                     •    L T
Language: modula2                                Level: Declaration

(* Sourced Queues *)

IMPLEMENTATION MODULE BQueue;

FROM Support IMPORT StringCopy, StringAppend, Error, EOS;
FROM Semaphore IMPORT Sem, SemInit, P, V, SemTrace;
FROM io IMPORT output, Writef;
FROM Storage IMPORT ALLOCATE;

TYPE
    BufferQueue = POINTER TO BufferQueueRec;
    BufferQueueRec = RECORD
        Count: Sem;
        Mutex: Sem;
        Head: Buffer;
        Tail: Buffer;
        Name: NameStr;
        Cname: ARRAY [0..20] OF CHAR;
        Mname: ARRAY [0..20] OF CHAR;
    END;

VAR Trace: BOOLEAN;

PROCEDURE BGet(Queue: BufferQueue):Buffer;

VAR     Ptr: Buffer;

BEGIN
    P(Queue^.Count);
    P(Queue^.Mutex);
```

Figure 1: Screen Image of *Pan*

Each of the following sections discusses the architecture of *Pan* from one of these points of reference. Throughout, the "user" is assumed to be a person is using *Pan* to edit a document. Users may at times extend or customize their running version of *Pan*, but they are not expected to have any knowledge about the internal structures of the system.

2 Implementation Levels

The implementation of *Pan* consists of three levels: *kernel*, *language-primitive*, and *extension* as illustrated in Figure 2. Each level builds on the lower level. Communication between levels is strictly hierarchical. Language-primitive commands call kernel-level functions, and extension-level commands call primitives, but there is no communication between the extension level and the kernel.

Kernel-level procedures are called *functions* in this document; language-primitive and extension

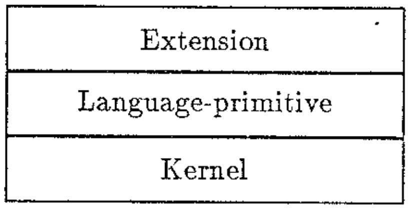


Figure 2: Levels of Abstraction in the *Pan* Architecture

procedures are called *commands*. Commands are differentiated from functions by the use of the command Define-Command rather than the *Lisp* defining function `defun` for their definition.

2.1 Kernel Level

The kernel level comprises the procedures and data structures that implement the fundamental components of *Pan*. The programming languages *Lisp* and *C* are used in the prototype. The boundary between the two languages is fluid. In general, policies and data structures are written in *Lisp* while communication with the window system, access to other system packages, and additional speed is provided by *C* functions. *Pan* uses *C* in the same way that many systems use assembly language—as a lower-level, faster language providing more access to the system and hardware than is available in the primary implementation language.

2.2 Language-primitive Level

The language-primitive level is a collection of editor commands that implements the basic editing language of *Pan*. It presents a coherent language for defining editor commands while shielding the individual writing such commands from the vagaries of the implementation. This structure allows changes in the underlying implementation to be hidden from the command level functions. For instance, the language-primitive command Announce displays a message on a viewer's annunciator line using the kernel-level function `sw-proclaim`. The details of the window system command for displaying a message are hidden by this interface.

The basic editing language of *Pan* combines *Lisp* control structures with *Pan* command definitions to provide a programming language. Most of the data structures used in programming *Pan* are supplied by *Pan*, although a programmer has recourse to basic *Lisp* control constructs as needed.

The data types in the editing language include regions of text, marks, cursors, tree-nodes, selections, rings, stacks, undo-arguments, buffers, and viewers. There is often a one-to-one correspondence between objects in the editing language and objects in the underlying implementation, but such a correspondence is not strictly necessary. For instance, in the editing language, a viewer and a viewport (see below) are treated as a single object, even though they are distinguished in the kernel.

2.3 Extension Level

The extension level is composed of editor commands written in *Pan*'s editing language. Such commands use other editor commands, together with underlying *Lisp* control structures, to implement new functionality. It is not expected that the creator of an extension-level command will need to know anything about *Pan*'s kernel.

3 Functional Components

Pan is composed of 7 functional components. These basic, separable units correspond to the modules of it's high-level design:

1. User Input/Output
2. Command Definition and Execution
3. Text Representation and Manipulation
4. Tree Representation and Manipulation
5. Syntax Definition and Checking
6. Semantics and Checking
7. Information Repository Interface

No component can operate in isolation, however; they depend on one another as shown in Figure 3.

A basic editor can be constructed from three of *Pan*'s functional components: user input/output, command definition and execution, and at least one editable representation. More interesting combinations produce more powerful editors:

User I/O + Command Execution + Text	=	Text editor
User I/O + Command Execution + Tree	=	Tree-structure editor
User I/O + Command Execution + Text + Tree	=	Structured document editor
User I/O + Command Execution + Text + Tree + Syntax	=	Language-based syntax editor
User I/O + Command Execution + Text + Tree + + Syntax + Semantics + Repository	=	Semantics-directed LBE

The last combination is the standard configuration of *Pan*.

The remainder of this section describes *Pan*'s functional components in more detail. For each component and subcomponent, the basic functionality is defined and the abstractions that the component provides are outlined.

The functional component view of *Pan*, presented in this section, is orthogonal to both the implementation-level view of Section 2 and the user services view of Section 5. However, components and the basic objects from which they are implemented are strongly related.

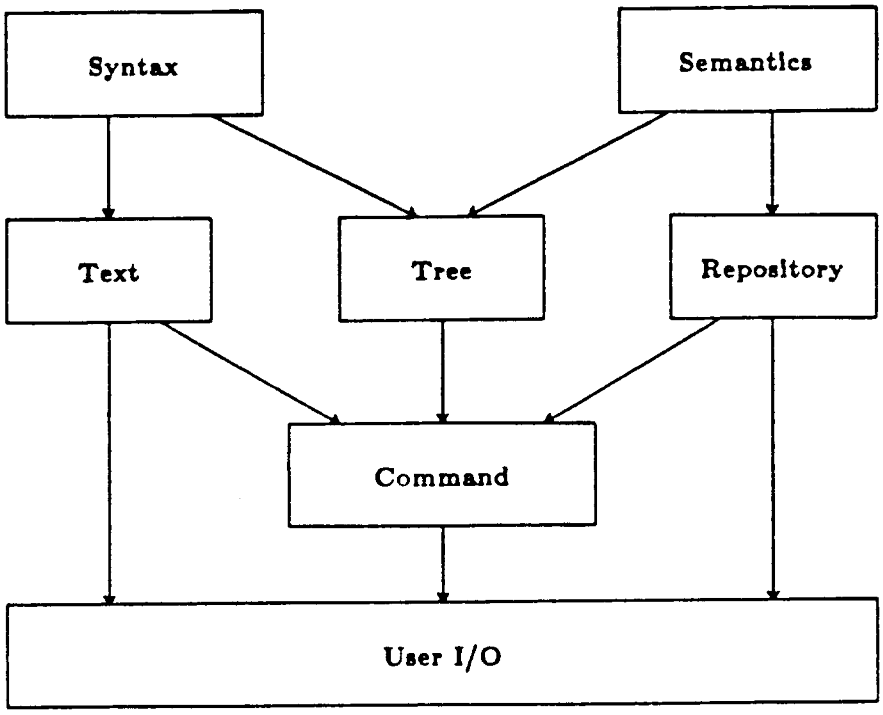


Figure 3: Component Dependencies in *Pan*

3.1 User Input/Output

The user input/output component is the base level for the entire system. It provides facilities for event handling (user input), screen management, and windowing. Most of the user input/output component is implemented in the *C* programming language. It provides the abstractions for events, timers, annunciators (user message facilities), prompters, menus, window management, imaging, and communication with other programs.

3.1.1 Event Handlers

Pan uses an event-based model of user interaction. Rather than polling for user input events, *Pan* cedes control to a manager that mediates between the window system and the user. This approach can be simulated in a polling-based windowing system such as X by creating a low-level polling loop to invoke the various event handlers.

Various classes of events are of interest to *Pan*. For each class, there is a designated function, written in *C*, that is called when an event of that class occurs. The *C*-level function is called the *low-level* event handler. In many cases, the low-level event handler passes control to a *high-level* event handler which is usually written in *Lisp*. Data may be filtered and remapped by the low-level handler before being passed to the high-level handler.

The current event classes are:

- **Keyboard events.** A keyboard event is signaled when the state of a key changes. Keys include keyboard, function, and mouse keys. Among other things, the low-level handler filters out key-up events, and maps the various key event representations into the integer key codes used internally by *Pan*. Information about the most recent keyboard event is retained by the handler in order to satisfy possible later inquiries.
- **Window events.** Window events describe changes in the state of the workstation's windowing system. For instance, the class of window events includes resize, repaint, move, open, close, quit, and get-input-focus.
- **Timer events.** Timer events are associated with the signaling of the interval timer.
- **Scroll events.** Scroll events are triggered by the user's manipulation of scroll bars. In the current implementation, horizontal scrolling is handled entirely by the underlying SunView implementation package, while vertical scrolling is handled by *Pan* itself.
- **Panel events.** A *Pan* viewer contains a "control panel" with items sensitive to user input actions. Each user-sensitive panel item has one or more events associated with it. Currently, these items relate to the language being edited, and the current operand level in effect.

3.1.2 Timers

Pan has a single interval timer. The timer is used to trigger activities in the absence of user actions. It is the policy that events triggered directly by user actions cancel pending timer events.

The high-level timer handler interprets a list of timeout actions. It first performs the action associated with the current time-out. If there are further actions on the action list, the handler then sets the timer to the interval associated with the next action.

This mechanism is used to implement timing-related actions such as window update (refresh). It can also be used to implement a policy such as "almost-per-key stroke" syntactic and semantic checking, or preemptive garbage collection.

3.1.3 Annunciators

Annunciators are facilities for displaying messages. They include the message line on viewers, the ability to flash or beep a viewer, and the manipulation of visible "flag" values on a viewer.

3.1.4 Prompters

A prompter is a facility for requesting input from the user. Currently, prompters are implemented using pop-up windows that take over all communication with the input devices on the workstation. Prompting is a therefore a synchronous activity.

3.1.5 Menus

The menu facility provides the ability to create menus, to add items to menus, to display menus, and to retrieve a menu selection. The basic abstractions are those of a menu and an item in a menu. Menus are denoted by small integer handles. Each item in a menu has an associated value that is returned to *Pan* when that item is selected. This value serves as a descriptor for the selected item. The descriptor value 0 is used to indicate that no item was selected.

3.1.6 Window Management

This component maps the *Pan* concept of a window (called a viewer) to objects in the underlying window system. In *Pan*, a window onto a buffer is called a *viewer*, while the actual object known to the workstation's window manager is called a *viewport*. Viewports are allocated to viewers as necessary. Viewers that are not visible on the display do not require a viewport. In this way, viewers (common in *Pan*) are decoupled from viewports (scarce in SunView). Distinguishing viewers from viewports appears to be useful even in an environment rich in viewports.

3.1.7 Imaging

The imaging component supplies abstractions of fonts, font maps, and graphical operations to a viewer. In particular, it provides a notion of a canvas on which graphical objects can be drawn. A canvas has a "current location" at which most operations occur. Each canvas is in one-to-one correspondence with a viewport.

3.1.8 Communication with Other Workstation Clients

The SunView system provides a facility, called the selection service, for transferring textual data among client programs. *Pan* includes an interface (not yet implemented) to the selection service so that *Pan* can participate in this exchange with other workstation clients.

3.2 Command Definition and Execution

The command definition and execution component associates user-initiated events with their resulting actions in *Pan*. It has five subcomponents: definition, binding, undo, help, and customization. Many of these components single-handedly provide services to *Pan*'s users. Section 5 provides more complete descriptions of the subcomponents as they provide services to a user.

The command execution loop reads input events, and using the current bindings, maps sequences of events to a chosen editor command. Following execution of the command, the results are passed to the undo component to be retained for possible later use. This component also provides error signaling and handling facilities for the extension language.

3.2.1 Command Definition

The command definition subcomponent provides facilities for defining new editor commands. Commands are defined using the language-primitive level command Define-Command. It is possible to execute *Lisp* functions as if they were commands, but it is not usual. Commands can be compiled by the system *Lisp* compiler. Define-Command also extends *Lisp*'s keyword argument facilities to include arguments to be obtained by prompting the user.

During command definition, several actions are taken on behalf of the help subsystems. The *documentation string* associated with a command is retained for use by the help component. The *name* of the command is broken apart at hyphens, and the different subwords are retained for use by the apropos subsystem. Other *apropos keywords* are gathered and added to the apropos database.

Both the help and the binding subsystems use the *argument list* of a command—one for documentation purposes, and the other to determine whether the command can be bound. (A command can be bound if all of it's arguments are optional or can be prompted for.)

The execution processor uses the command's *after*, and *undo* daemons when the command is executed. The *calling discipline* is used at definition time to create commands that are really *Lisp* macros.

Commands can use *Lisp* control and data structures in their implementation. If a command uses a *Pan* kernel function, it is considered a language-primitive-level command.

Each command can have up to two daemons: an undo operation that is invoked to undo the changes made by the command, and an after operation that is called following execution of the command. The after daemon is called with the arguments that were the results returned by the command.

All *Pan* commands are executed in an environment in which the active buffer (the buffer towards which the command is being executed) is accessible. From the active buffer, a command can access the buffer's language, the active viewer, and the cursor of the active viewer. The active buffer also contains the buffer-local bindings, options, and flags.

3.2.2 Command Binding

Bindings associate sequences of input events with editor commands. It is also possible to bind a *Lisp* form (rather than a command name) to a sequence of events.

Pan provides three forms of bindings, distinguished by how the event sequence originated. Keyboard bindings map a sequence of keyboard events to a command, menu bindings map the selection of a menu item to commands, and operand-level bindings map a pair (operand-level, generic-operation) to a command. The generic operations are `next`, `previous`, `select`, `mouse-select`, `mouse-extend`, `cut`, `copy`, `paste`, and `delete`. The set of generic operations is a *Pan* compile-time option. Bindings are discussed in more detail in Section 5.2.1.

3.2.3 Undo

The undo processor collects information about command executions and retains it for later use. The `Undo` command implements a specific undo strategy. The current strategy is one of single undo/undo, where only the most recent command, or sequence of similar commands can be undone. Only the only most recent undoable command is remembered. This means that a sequence of commands that are not undoable can be executed, and then the most recent undoable command can be undone.

If a command can be undone, an undo daemon must be specified with the command and the values returned by the command must be valid arguments to the undo daemon. Multiple values can be returned by the command. The maximum number of results is limited by the value of the compile-time constant **max-undo-args**.

If the command has an after daemon, the results of the command are passed to the after daemon, and the results of the after daemon are retained by the undo daemon. After daemons are typically used to collect the undo information from a sequence of similar commands (such as a sequence of character insertions) into a single undo action.

The undo daemon for a command should be another editor command; it may itself be undoable.

3.2.4 Help

Online help facilities provide assistance to the user. There are two primary mechanisms. "Apropos" allows one to get help information using keywords. "Describe" gives detailed information for a single command, option, or flag. Both use information gathered automatically from command definitions. Descriptions of other objects, such as the active bindings of a buffer, must be provided by hard-coded procedures.

Help information is displayed in a special buffer managed by the help component.

3.2.5 Customization

Customization includes the processing of the start-up configuration file, loading command libraries, setting configuration and policy parameters, defining new parameters, defining variables and flags for use in extension programming, and automatic execution of commands upon creation of a new buffer. (See Section 5.2.)

3.3 Text Representation and Manipulation

The text component provides an editable representation of text. Like any purveyor of an editable representation (see Section 4.2), it supplies an internal representation, display facilities, and primitive editing operations. The text component also supports textual searching and text file input/output.

3.3.1 Internal Representation of Text

The internal representation of text stores characters in an edit-worthy data structure. The text data structure provides for insertion, deletion, and copying of textual regions, two coordinate systems, and methods for referencing characters. The internal representation also provides pointers to the lexical stream (in a parsed-tree representation) that are used to map from characters to lexemes.

Text that is part of the textual representation of an object being edited is called *the text stream*. In the future, it is expected that the text stream will be discontinuous, consisting of isolated pieces of text linked together by other editable representations. Text deleted from the text stream is placed into *deleted text space* from which it can be retrieved.

Text in *Pan* can be conceptualized as either as a stream of characters or as a two-dimensional infinite quarter-plane of characters. Both views are implemented by distinct coordinate systems overlaid on the internal representation.

Characters are represented using 16-bit integers. Eight bits of the representation is used for the character code, four bits for a font code, and four bits are reserved for display mode information. Character codes 0–127 are the normal ASCII codes. Codes 128–255 are used for non-ASCII keyboard codes such as mouse buttons and function keys. The full range of character values (0–255) may be used for processing keyboard events. The internal representation of text truncates the non-ASCII characters to 7-bit ASCII values before placing inserting them into the text stream.

Each character object contains four mode bits. The text viewer paints characters by interpreting each bit independently. At present, the four supported modes are **underline**, **inverse**, **gray**, and **background**. A buffer option specifies which mode to use for displaying the current selection in text viewers; the choice is currently limited to **underline** (the default) and **invert**.

The four bits of font code can be interpreted as small unsigned integer index (0–15) into a *font map*. Each *Pan* viewer has a font map for interpreting font codes. The default font index is 0; empty slots in the font map are filled with the default font.

Font and mode information is uninterpreted by the text manipulation, navigation, and selection facilities, though it is interpreted by display mechanisms.

Each ASCII character is a member of a single character syntax class. Character syntax classes are used to implement text cursor motion commands such as searching for balanced brackets or skipping over white space. The character syntax classes defined by *Pan* are: "space", "word character", "punctuation character", "left bracket", "right bracket", and "other".

There are several ways to refer to characters, including character pointers, marks, cursors, and regions. Character pointers refer to a character, not a position between characters. They are implemented using sticky-pointers.

In *Pan*, where character pointers are maintain the mapping between lexemes and underlying text, updating every character pointer on every insertion or deletion would be very costly. The sticky-pointer data structure was chosen because the individual sticky-pointers do not have to be updated until they are dereferenced. The drawback to the sticky-pointer implementation is a higher storage overhead.

A mark indicates a position between two characters in the text stream. Marks are implemented using character pointers that are updated following every deletion of a character. Only the marks that point to just-deleted characters have to be updated. (Marks implemented by sticky-pointers require no update following insertions.) Each buffer contains a stack of marks that can be manipulated by either a user or by someone programming new commands.

The distinction between character pointers and marks is subtle but important. It is best illustrated by considering the deletion of a character. Suppose that the text "Pan" is in the text stream, that *CP* is a character pointer pointing to the "P", and that *M* is a mark for the position of "P" in the text stream. If the character "P" is deleted, *M* will now refer to the position to the left of "a", while *CP* will still point to "P" where "P" now resides in deleted text space. Marks always refer to positions in the text stream; a mark cannot refer to a position in deleted text space.

Cursors are the means for navigating and selecting text. Text is inserted to the right of the position designated by a cursor.

A text cursor is implemented by a mark together with cached quarter-plane coordinates. Converting a cursor to a mark or a mark to a cursor is a simple operation.

A *region* is a contiguous sequence of characters. Every command that inserts or deletes text ultimately uses one of the commands Insert-Region or Delete-Region. Commands that operate on single characters, such as the command that deletes a character at the current cursor, act upon implicitly defined regions. The notion of a region is fundamental to both editing and display in *Pan*. In contrast, the *Emacs*[8,9] notion of a region is much weaker.

Various methods are provided for storing and manipulating text objects. In particular, the text representation provides rings (circular bounded stacks) for marks, for deleted text regions (the kill ring), and for text regions shared between different objects (the clipboard).

3.3.2 Text Alteration

Alterations of text are implemented by the kernel primitives for inserting and deleting single regions into the text stream. These are the only two operations that modify the textual representation. When a textual representation is modified, all of the viewers that are clients of that representation are notified.

3.3.3 Text Display

The text display component maps a textual object onto a viewport's display area. This map is currently implemented as an infinite quarter-plane. Horizontal scrolling is supported; line-wrap is not. A textual display object is called a *viewer* in the *Pan* nomenclature. There may be several viewers sharing a given textual object. Users always interact with objects through viewers. Each viewer provides a cursor at which operations may occur.

3.3.4 Regular Expression Matching

The *Pan* textual representation supports the standard UNIX regular expression facilities (see the ED(1) manual page).

3.3.5 Text File I/O

This component provides facilities for reading, writing, and appending text files. The objects read from or written to text files are regions of text.

3.4 Tree Representation and Manipulation

The tree representation and manipulation component provides another of *Pan*'s editable representations. It provides primitives for constructing, navigating, selecting, viewing, and editing tree structures. Tree editing is distinct from the syntax definition component so that simple tree-structured editing can be provided independently from the more complex representations used to implement programming languages. "Editors" for simple tree structures such as outlines or pure trees should be simple to implement using *Pan*'s primitives. By providing a basic tree data type, the particulars of any tree can be regarded as an mapping onto the general type. The same definitions used to define the layout of a syntax tree should be used to define the internal representation of simpler trees.

3.4.1 Internal Representation of Trees

The tree representation provides trees using node, parent, and children abstractions. Navigation operations are the standard tree operations (up, down, left-sibling, right-sibling). Selection is by tree cursor—currently implemented as a pointer to a tree node. The only coordinate system provided by the tree component is that of tree node addresses. Tree nodes also provide for attached properties, and for extensions (in size) to support trees created by a parser and for use by semantic processors. Nodes in parsed trees (or semantically checked trees) can be regarded as subtypes of basic tree nodes.

3.4.2 Tree Alteration

Tree alteration is via kernel operations that replace a subtree with another subtree.

3.4.3 Tree Display

A tree-oriented viewer is under development. This viewer will pretty-print tree structures using user-customizable format specifications. Like text-oriented viewers, it will provide a cursor for editing commands.

3.4.4 Tree I/O

This component reads and stores tree structures in the file system. It is not yet implemented.

3.5 Syntax Specification and Checking

The syntax component builds on both text and trees. As a combiner, it uses facilities from both subcomponents to provide editing operations on structures having both text-oriented and tree-oriented aspects. For instance, the "cursor" actually implemented in *Pan* combines a text cursor and a tree cursor. Selecting a subtree also selects the region defined by the tree's yield. Deleting a subtree deletes the associated region. The support for editing is implemented by a collection of *Pan* commands.

Using the two editable representations, it provides incremental mappings from text to tree and from tree to text (as necessary). It also provides navigation and selection mechanisms based on the syntax of the language being represented. The syntax component can be divided into several subcomponents: *Ladle*—the language-definition processor, the lexical table generator, lexical analyzers, the parse table generator, parsers, internal (tree) representation, and support for editing.

3.5.1 Ladle

Ladle[5] is a preprocessor for language descriptions. A language description contains four parts that describe the lexical structure, the parsing grammar, the abstract syntax, and the static semantic constraints of a language. Only the abstract syntax description is required, but if text-oriented editing of syntactic structures is supported, then the information from lexical and syntactic specifications is necessary. The semantic description is required only if semantic checking is to be performed. Figure 4 illustrates the flow of information from a language description through *Ladle* and into *Pan*.

Ladle verifies that its input is well-formed by checking various relationships between the four parts. When the description is well-formed, *Ladle* generates several outputs: input to the lexical analyzer generator, input to the parser generator, and tables describing the language and its internal

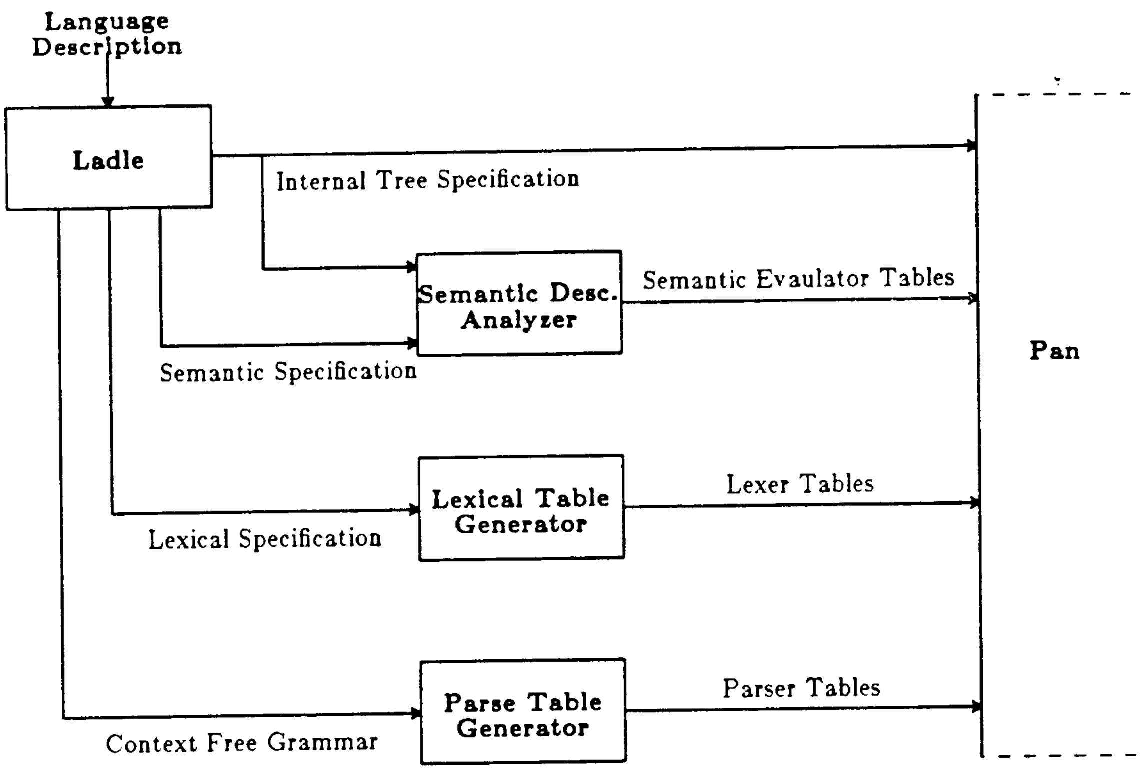


Figure 4: Flow of Information between *Ladle* and *Pan*

tree-structured representation for use by *Pan*'s editor commands and by the semantic description analyzer. After the tables have been generated, *Pan* loads them to create a language description object (Section 4.5).

3.5.2 Lexical Analyzer Table Generator

The lexical analyzer table generator creates tables for use by *Pan*'s lexical analyzers. The input to the lexical analyzer generator is a description of the lexical portion of a language. The description can include regular expressions and expressions denoting bracketed expressions. The bracketing can be either nested or non-nested. The output of the lexical analyzer generator is a collection of tables.

3.5.3 Lexical Analyzers

Pan's lexical analyzers synchronize a stream of lexemes with an underlying text stream. A full (non-incremental) lexical analyzer recomputes the entire stream of lexemes from scratch. An incremental lexical analyzer updates only the changed portions of the lexical stream. Additionally, an incremental lexical analyzer creates a summary of changes for use by an incremental parser. In the current implementation, an incremental lexical analyzer capable of creating an entire stream of lexemes is provided.

The stream of lexemes maintained by a lexical analyzer need not be an editable representation. It would be possible to provide the user with the illusion of traversing and editing the lexical stream using operations defined on the parsed tree representation.

3.5.4 Parse Table Generator

The parse table generator creates LALR(1) parse tables for use by *Pan*'s parsers. The input to the parser generator consists of a description of a context-free language together with links to the language's lexical specification and error recovery information. Output of the parser generator is a collection of tables.

3.5.5 Parsers

Pan's incremental parsers maintain the tree-structured representation of the object being edited. A full parser creates a tree from scratch; incremental parsers need only to update changed areas of the tree by synchronizing the tree with the lexical stream maintained by the lexical analyzers. Currently, only an incremental LR(1) parser is provided. It is capable of parsing a buffer from scratch just like a full parser.

Parsers must also provide information to the semantic analyzer. Even though bottom-up parsing is used, it is necessary for the parser to alter the tree as little as possible. In particular, if the modification is isolated to a single subtree, then the nodes on the path from the root to the modified tree must be preserved with their annotations intact. Such nodes would be classified as possibly changed, and their semantic attributions reviewed by the semantic checker. What must be provided is a partition of tree nodes into four categories: created by the parser, deleted by the parser, possibly changed by the parser, and definitely unchanged by the parser.

3.5.6 Internal Parsed Tree Representation

The tree component of *Pan* supplies the underlying editable representation, while *Ladle* generates tables that describe the actual tree-node descriptions and relationships.

The abstract syntax specified in a *Ladle* description includes information describing the contents of tree nodes. For instance, besides the primary structural information, information is provided to reserve slots in tree nodes for use by lexical analyzers, parsers, the semantic evaluator, and other tree-processors. A symbolic name for the node type is also provided in the description of the abstract syntax.

The trees constructed by a parser are approximations to parse trees. Structurally, they resemble abstract syntax trees decorated with sufficient additional information to enable an incremental parser to use the tree as a base for parsing. Other slots may be provided for semantic information, or for attributes maintained by other tools.

Syntactic errors in the textual representation of an object are represented in the tree by special "error" nodes. An error node has a variable number of children. Subtrees (and lexemes) that are discarded by the parser during error recovery become the children of the error node that represents the recovery.

It should be possible to determine, from an arbitrary position in a tree, the language description used to create that tree node. In the current implementation, only a single language per buffer is supported, so this requirement is trivially satisfied.

3.6 Semantics Specification and Checking

The semantics specification and checking component of *Pan* provides facilities for checking the context-sensitive semantic constraints of a language. It has two subcomponents: a description analyzer and an evaluator. Semantics requires the presence of both the tree component and the database component in order to function. Usually, a semantics component will be used in conjunction with the syntax component.

3.6.1 Description Analyzer

The description analyzer provides a language for defining contextual constraints between subcomponents of an object. Usually, the subcomponents will be nodes in an abstract syntax tree that represents the object. From a description of the constraints, the analyzer generates tables required by the evaluator to ensure efficient, incremental semantic checking.

3.6.2 Evaluator

The semantic evaluator checks the contextual constraints associated with nodes in the internal structured representation of an object. Information about the nodes is retained in the information repository, or database. Constraints are re-checked following each incremental parse, if incremental parsing is being used. Otherwise, checking can be performed after each structure-editing operation.

When using the incremental parser, the evaluator uses information gathered by the parser about how the abstract syntax tree has changed. The parser is able to detect four classes of tree nodes: newly inserted, newly deleted, potentially changed, and nodes definitely unchanged by the parser. The classification provided by the parser is used as a first step in reevaluation. The semantic information of any node in the tree might be altered, of course.

3.7 Information Repository Interface

Pan is planned to serve as the front-end for an integrated programming environment. The programs developed, together with the information gathered during the development process, will be stored in a persistent, shared, information repository. This component of *Pan* provides access to the information repository and requires certain functionality of the repository. It is composed of three interfaces to the repository: structure access, semantic information storage, and user queries.

3.7.1 Structure Access

The structure access interface provides for creating, storing, and accessing structures (trees, graphs, text) in the information repository. When a structured object is being manipulated, an in-memory representation may be necessary. The information repository must therefore supply grouping mechanisms for fast reading and writing of large, structured objects.

3.7.2 Semantic Information Storage

The semantic evaluation process of *Pan* is modeled on logic programming. Information about objects is stored in a graph-structured semantic database where the structure of the database reflects the scoping structure of the object being edited. The semantic database is embedded in the information repository. The fundamental operations in the incremental semantic evaluator are the addition and removal of facts—and discovering which constraints require reevaluation following the addition or removal of a fact.

The information repository must provide storage for facts about objects (to the granularity of tree or graph nodes), and for dependency information on facts. The evaluator itself is responsible for maintaining the facts and the dependency information. Since the implementation of the semantic evaluator is based unification, the information repository (or its in-memory version) must support efficient unification.

3.7.3 User Queries

The user query interface allows a user to access the data in the information repository. This includes help in formulating queries, a language for expressing queries, and methods for displaying results. The syntax of the query language need not be the syntax of the semantic description language. User queries can be supported by providing a surface syntax that translates into the underlying semantic specification language.

4 Basic Objects

This section describes the basic objects in *Pan*'s architecture. The implementation of the objects is not described. Each object may provide certain services, and may own other objects.

4.1 Buffer

A buffer is a locus of editing attention for an object or collection of objects. A single buffer coordinates access, modification, and display of one object among several viewers.

Each buffer consists of:

One or more language descriptions
A collection of editable objects
One or more editable representations
One or more viewers
Local option values
Local key bindings
Local menus and menu bindings
Flag values
Undo information
A kill ring
A mark ring
A current selection

Locally cached data

A language description provides the tables and other necessary definitions needed for language-based editing. Except for the built-in language "Text", most of the language description will be constructed by *Ladle*.

The multiple editable representations are different representations of the same underlying object(s). Each viewer uses one or more of the representations to construct a presentation of the object. The viewers are not constrained to be a single type—more than one type of viewer can be open onto a single buffer. Local values of options and bindings have already been discussed. Flag values are always local to a buffer.

The kill and mark rings are data structures used for programming commands. "Killed" text is retained in the kill ring. Marks (positions in the text stream) are stored on the mark ring. The current selection is a region of text (or a tree node) that serves as the operand for many editing commands. Locally cached data is used for added efficiency.

4.2 Editable Representation

An editable representation is a low-level representation of an object. Editable representations must provide the following minimal set of facilities:

Fundamental abstractions
Construction
Coordinate system(s)
Reference
Selection
Navigation
Editing operations

It may optionally supply mappings to other editable representations of the same underlying object, and file system interactions. In general, an editable representation may be designed in such a way as to facilitate the implementation of certain operations.

For instance, the text representation supplies the following:

Fundamental abstraction:	Character
Coordinate system:	Stream offset and quarter-plane coordinates
Construction:	Read-Region-From-File
Reference:	Character pointer, marks
Selection:	Region abstraction
Navigation:	Cursor
Edit Operations:	Delete-Region, Insert-Region
Facilities:	Character display modes and fonts Text to lexeme mapping

Similarly the parse-tree representation supplies the following:

Fundamental abstractions:	Subtree, Tree node
Construction:	Incremental-Parse
Access:	Subtree pointer
Alteration:	Replace-Tree, Incremental-Parse
Coordinate system:	Subtree address
Navigation:	Tree cursor abstraction
Selection:	Tree abstraction, tree yield
Mappings:	Tree node to lexeme
Facilities:	Tree node properties, attribute slots

(Section 5.1 discusses many of these facilities in detail.)

4.3 Font Map

A font map associates fonts with font descriptors.

4.4 Key Map

A key map associates key sequences with command names or other *Lisp* forms.

4.5 Language Description Object

A language description object contains the information necessary for language-based editing, including coordination between multiple editable representations of a single object. Language descriptions do not include such local information as operand-level definitions and language-specific bindings. The bulk of a language description are tables for tree node creation, scanning, parsing, tree construction, and semantic analysis.

4.6 Viewer

A viewer is *Pan*'s notion of a window. Users edit objects through viewers. Every viewer is associated with a buffer; buffers may have more than one viewer. When visible on the screen, a viewer is displayed in a viewport. Several different viewers, having different artists, can coexist within *Pan*. Not every viewer will be able to intelligently display every editable representations used in *Pan*. For instance, a purely text-oriented viewer may not be useful for displaying pure trees, and vice versa.

Viewers present an object on a display surface, and provide scrolling and navigation. Since there may be several viewers onto a single object, each in a separate window on the workstation's screen, a viewer has its own local state. The local state of a viewer is retained even when the viewer is not being displayed

A viewer consists of:

An editable representation (shared with a buffer)
Size and scale
A position relative to editable representation
A cursor

An operand level
An input state (sequence of events directed at that viewer)
A font map
A viewport

Viewers are registered with a single editable representation. When the representation is altered, all of its registered viewers are notified of the change.

Since the presentation of an object may not fit within the confines of a viewer, the viewer is responsible for providing some sort of display condensation. For a text viewer, this consists of scrolling. The position relative to the editable representation places the viewer in the object. For a scrolling viewer, this position might be the offset in lines from the first line of the object.

The edit cursor, operand level, and input state are local to a viewer because each viewer can be an active editing window. Any viewer can set its buffer's current selection.

The viewer's input state contains the current event sequence—it is cleared after a command is executed from the top level. A keystroke sequence can be started in one viewer (e.g. `Esc-`), the input focus shifted to another viewer, commands executed elsewhere, and the input focus returned to the original viewer where the sequence can be completed.

The font map local to a viewer defines the interpretation of the font codes on characters in an underlying textual representation. Every character object contains a four bit font identifier. The text viewer paints characters by reference to a sixteen-slot font map, local to each viewer instance. A buffer option specifies the font map to be used by all associated viewers. Changes to the font map propagate dynamically. Since fonts have different characteristics, the font map is required in order to map from a presentation to the internal representation.

A viewer implements the following abstractions:

A local coordinate system
An artist (or renderer)
A mapping from an editable representation to a presentation
A presentation (ephemeral)
A mapping from local coordinates to presentation coordinates
A mapping from presentation coordinates to local coordinates

The local coordinate system of a viewer specializes one of the coordinate systems supplied by the underlying editable representation to the size and position of the viewer. For instance, the text viewer has a local coordinate system in which line 0 is the topmost line visible on the viewer. The position of the viewer in the text file is used to convert to absolute text coordinates.

An artist is an interpreter which creates a presentation from an object (represented by one or more editable representations) and semantic rules (e.g. pretty printing rules or character display rules) An artist may be as simple as a routine which expands tabs and formats control character, however this definition permits an artist to be a debugger, a programming language interpreter, a WYSIWYG system, or a text formatter such as TEX. The function of an artist is embedded in a viewer.

The artist also implements the mapping from representation to presentation. The mapping must be well defined so that actions relative to the representation can be accurately displayed relative to the presentation, and vice versa.

A presentation is the result generated by an artist. In our system, it will generally be a visual representation of the underlying object. It need not be objectified. Since presentation coordinates are not necessarily representation coordinates, the viewer must supply transformations from one to the other.

4.7 Viewport

A viewport mediates between a presentation as created by a viewer and a physical output device. It may be the case that relatively few viewports are available. In this case, they will be multiplexed between perspectives and buffers as necessary. Each viewport owns a presentation surface and a location (in presentation coordinates) at which the next display operation will occur.

A viewport provides a coordinate system for the presentation surface (normally pixels), and a collection of presentation operations for use by the artist. Viewports are allocated to viewers when the viewer is made visible. A viewer has at most one viewport.

4.8 Global State

Pan's global state is a collection of shared objects. The following objects are in the global state:

- **Base Buffer** The *base buffer* is a distinguished text buffer is created at start-up time. It contains a list of the other buffers being edited. The base buffer's viewer is the only viewer that can be made iconic; when it is made iconic, all of the viewers visible on the screen disappear. When the *Pan* icon is opened, the viewers that disappeared when *Pan* was made iconic reappear.

- **Help Buffer** The *help buffer* is a special buffer used by the help subsystems. Output from help commands is displayed there.

- **Buffer List** The buffer list is an internal *Lisp* variable that contains the list of all of the active buffers except for the base buffer.

- **Active Buffer** The active buffer is normally the buffer to which input actions are directed. The active buffer has an active viewer (also called the *edit window*) which is viewer having the focus of editing attention. Commands may temporarily act on buffers that are not the focus of keyboard input, as when the help system generates text into the help buffer.

- **Global Key Map** The global key map contains the default key bindings. When a keystroke sequence is processed, the global key map is consulted whenever the key map local to the active buffer does not contain a binding.

- **Root Menu** The root menu is copied into each buffer when the buffer is created, so that local menu bindings will inherit global bindings as desired.

- **Menu List and Menus** The global menu list and menus contain the menus defined in *Pan*.

- **Variables** Global variables are *Lisp* and *Pan* variables used by the implementation.

- **Option Table** The global *option table* contains the global option values inherited by all buffers. Option tables local to buffers are used to store local values for options.
- **Clipboard** The *clipboard* is a ring for storing text or trees. It is shared between buffers so that text or trees can be cut and pasted between buffers.
- **Help Data** Help data includes a hash table that maps apropos keywords to command names and descriptors for each defined *Pan* symbol. The descriptors appear as properties on the symbol's property list.

5 Services Provided to the User

A user of *Pan* sees a number of *services* that support the user's activities, services that are supported by various primitives of the editing language. From this point of view, *Pan* offers the following services:

1. Editing
2. Customization
3. Extension
4. User Interaction
5. File System Interaction

Each of these categories is discussed in more detail below.

Providing a service may require cooperation among several of the system's functional components. For instance, one service is the ability to select an object. As a service, selection requires functionality from both the user input/output component and the editable representation of the object being edited. Provision of a service also requires coordination between functions at the kernel level and commands of the language-primitive implementation level.

5.1 The Editing Service

The editing service provides basic editing capabilities. In *Pan*, this includes construction, viewing, navigation, selection, alteration, and undo. (This framework is based on the model of Meyrowitz and Van Dam[6]). Editing services combine functionality supplied by an editable representation with the display functionality supplied by a viewer.

Construction allows the user to create new, possibly structured objects. For text, this is corresponds to the creation of a new, empty, textual object. For syntax trees, construction is provided by parsing. Reading objects from an external repository (a database or a file system) and creating an editable representation is also a form of construction.

Viewing provides the ability to display portions of an object on a screen. When a user views an object, a *presentation* of the object is created.

Navigation allows the user to move the focus of editing attention within an object by using the object's internal structure. For instance, the typical pure tree movements Up, Down, Left-Sibling, Right-Sibling are examples of navigation commands. So are Next-Character, Next-Line, Previous-Character, and Previous-Line. The operand-level mechanism provides an alternative form of navigation in syntax trees.

Selection provides the ability to choose an operand on which to perform an operation. Many character-oriented text operations operate on the character implicitly selected by the text cursor. Others operation on a distinguished region of text called the *current selection* that is independent of the text cursor. Tree-oriented commands operate on the subtree selected by the tree cursor.

Alteration provides the ability to change an object. Most alteration commands act upon operands chosen by the selection facilities such as regions or subtrees.

Undo provides the ability to undo the effects of a previous operation. The undo facility in *Pan* is designed to support experimentation with undo policies. Beyond the interface to command execution defined by the commands Mark-Undo and Undo, many policies can be implemented.

5.2 Customization

Customization allows users to tailor their editing environment to their own needs. *Pan* is intended to be a system in which almost all policies and values can be easily customized. The basic forms of customization include altering bindings, setting different values of options to control configuration, and the control of policies.

Pan supports two levels of definitions for many values. Key bindings, menu bindings, and option definitions can occur either globally (available to all buffers) or locally, within a single buffer. Normally, a local definition hides a global definition. This means that a redefinition of a global value will not be reflected in a buffer if that buffer has an overriding local definition for the same value.

5.2.1 Bindings

A *binding* is an association between a sequence of user input actions (typically key strokes) and an editor command. *Pan* bindings exist in three forms: key bindings, menu bindings, and operand-level bindings.

Key Bindings

A key binding associates a keystroke sequence with an editor command or a *Lisp* form. Every buffer has a set of key bindings together with the bindings inherited from the global key map. Bindings local to a buffer hide bindings of the global environment.

Key bindings consist of one- or two-keystroke sequences. The keys `Control-X`, `Control-Z`, `Control-C`, and `Esc` are reserved to be prefix keys. The shift and control keys act as modifiers upon a keystroke. For instance, `Esc-d`, `Esc-D`, and `Esc-Control-D` form three different keystroke sequences. In the ASCII encoding, `Control-D` and `Control-d` are indistinguishable.

Mouse buttons, and the function keys on a keyboard, act as bindable keys that can be used in conjunction with prefix keys. In the current implementation, the shift and control modifiers are not used with the mouse buttons, but can be used with function keys.

Key binding facilities provide an implementation of key maps, ways to print keystroke sequences, methods for setting and clearing key bindings, and code for printing key bindings which is used by the help component.

Menus and Menu Bindings

Menu bindings associate menu items with editor commands. Each menu used by *Pan* is declared in a global name space of menus, and is usually accessed by name. For instance, the menu named "Pan" is the root menu for all *Pan* menus.

Menus can contain other menus, menu items, and labels. A menu item has a title and a descriptor associated with it. The title appears in the menu display. The descriptor is an internal value returned by the menu-display component when that item is selected. Internally, *Pan* maintains a mapping between descriptors and editor commands. The value 0 is returned by the menu-displaying function when no item was selected. Labels are "non-selectable" titles that can be used to install text in a menu.

Like key bindings, a buffer has both local and inherited (global) menu bindings. This allows a buffer to add local menus and bindings to the basic *Pan* menu. (Internally, each buffer has a single root menu that is a copy of the global root menu with local bindings added.) Menus are copied as necessary. Menu bindings are accessed by the command Execute-From-Menu.

The menu service provides an implementation of menus, ways to describe menus, methods for setting menu bindings, and code for describing menu bindings which is used by the help component. It is not possible to remove menu bindings in the current implementation, although bindings can be redefined.

Operand Types and the Operand Level

Operand-level bindings are the third form of *Pan* bindings. The operand-level generalizes the notion of operand modes by supplying an explicit relationship between operand types and operations.

In a text editor, commands like Delete-Word combine the operation `delete` with the operand type "word" into a single operation. This combining is reasonable where the set of operand types is both small and well defined.

In a language-based system, the set of operand-types is considerably more rich. Along with the basic textual types of character, word, and line, a programming language might supply operand types like lexeme, error, declaration, statement, expression, or imperative. A *Ladle* language description provides a way to group node-types in the internal tree representation into operand classes.

A persistent operand-type selector, called the *operand-level* has been included in *Pan* for experimentation. Each buffer contains an operand type selector that is shared by all viewers open onto that buffer. The current operand-level is persistent: it is changed only by the user. The operand level is most useful in concert with navigation and selection.

Along with the operand-level selector are a set of generic editing operations such as `next` or `paste`. Operand-level bindings map a pair ⟨`level`,"generic editing operation"⟩ to an editor command. The basic method for doing this is to create a generic operation command such as Next-@Level which executes the "next" operation using the current operand level. New operand-levels can be added by a user, and current operand level bindings can be changed by a user.

Facilities for customizing the operand level bindings include methods for defining new levels, for binding commands at new levels, for changing existing bindings, and for describing the operand level bindings.

5.2.2 Options

Options are used to customize the behavior of *Pan*. An *option* is a strongly-typed variable declared in the global name space. When an option is defined, its name and global value are visible to all buffers. An option can be given a local value which hides the global value. Unless otherwise specified, reference to an option will retrieve the option's local value if it is defined, and otherwise its global value.

When an option is defined, its type and default value must be supplied. It is also possible to register a notifier procedure with the option; when the option's value is set, the notifier procedure will be called with relevant data. This implements a form of active values.

The option mechanism is used by those *Pan* commands that implement policy decisions. For instance, the command that loads a file of *Lisp* and *Pan* commands into the executing *Pan* searches a directory path specified by the value of the option :pan-load-search-path. In another example, the boolean option :proportional-scroll controls whether proportional vertical scrolling is used to control the vertical scroll bars. Constants, such as the default size of windows, and variables, such as the font map used by buffer, are defined using options.

5.2.3 Character Syntax Classes

The character syntax class of any character can be changed by the Set-Syntax-Class interface of the character syntax class implementation mechanism.

5.2.4 Configuration

The primary way for a user to configure *Pan* is to place a file of configuration commands named `".panrc"` somewhere in their working, or else their home, directory. This file is loaded when *Pan* starts up. The `".panrc"` file normally sets various option values, and can directly load other files or libraries of *Pan* code.

The auto-load mechanism provides a way to ensure that certain libraries are loaded when necessary. This mechanism associates a (list of) file names with a UNIX file name expression. For instance, one might associate the file `my-c-commands` with *C*-language files specified by the expression `"*.[ch]"`. Auto-load ensures that `my-c-commands` has been loaded into *Pan* whenever a file matching the associated expression is visited.

Auto-execution ensures that a specified command is executed upon creation of a buffer having a specified name. Like the auto-load mechanism, a list of command names is associated with a UNIX file name expression. When a buffer whose file name matches the expression is created, the associated commands are executed. This provides a way to implement minor modes—collections of language-specific local bindings.

A user's preferred bindings can be established from the start-up file, from automatically loaded files, or from automatically executed commands. Files are only automatically loaded once, so they should be used to define values in the global space. Automatically executed commands are executed when a new buffer is created. They can set values that are to be local to a buffer.

5.3 Extension

The extension services of *Pan* provide ways for a user to add new operations to the repertoire. New operations range in complexity from the addition of a new navigation commands to the creation of a whole new undo strategy. The primary extension services are command, option, and flag definition, and command compilation. The service that loads libraries of commands is considered to be part of customization.

Commands are the way that new operations are added to *Pan*. Options, flags, and variables are mechanisms for defining and manipulating internal values to control commands. It is possible to view options, flags, and variables as a single conceptual entity. In the current implementation, however, they have slightly different implementations and semantics tailored to different roles.

5.3.1 Commands

New commands are added to *Pan* using Define-Command. Commands are always added at the language-primitive-level, or the extension level. Command definitions contain various pieces of information used by *Pan*, including help information, specification of an undo inverse and an after daemon, and definition of the internal binding discipline of the command.

5.3.2 Options

Options are declared, strongly-typed variables that can be used by the author of a command to control the policies implemented in a command. Options are declared using the command Define-Option.

Help and apropos information for options is gathered as it is with command definitions. Each option has a type, specified by either a type predicate or as a list of legal values, and an initial value.

An option can also have a notifier function. When the value of an option is changed, the notifier is invoked (after the value is changed) with four arguments: the name of the option, the buffer in which the change occurs, the former value of the option, and whether the change is local (to the buffer) or global (to all buffers). When an option's value is changed globally, the notifier is invoked on each active buffer.

5.3.3 Flags

A flag is a boolean value defined to *Pan*. Definition of a flag results in a documentation string and apropos keywords being stored in the appropriate structures. Flag values are always local to a buffer.

Unlike options, flags have no notifier function. Instead, they can have a visible manifestation on the panel of a viewer. This visible marker must be a single character. A flag's marker has one of two behaviors: either blank when the flag is clear, or greyed-out when the flag is clear. In either case, the flag marker is in a solid font when the flag is set and the marker is visible. The set of visible flags is controlled by the option :visible-flags.

5.3.4 Variables

A *Pan* variable is a global *Lisp* variable that is known to *Pan*'s help system. It carries no other attributes or semantics. As a result, all variables are global. Help and apropos information for variables is handled just like the same information for options and flags.

5.3.5 Defining New Languages

Defining a new language is a form of extension. A language is defined to *Pan* in two phases. First, a language description is written and presented to *Ladle*. Second, the tables generated by *Ladle* (including the tables generated by the lexical analyzer generator, the parser generator, and the semantic description analyzer) can be loaded into *Pan*. Language tables can be dynamically loaded into *Pan* using the same extension facilities as for commands. New languages generally have specialized bindings and operand level definitions.

5.3.6 Compilation and Loading

Commands can be compiled using the system *Lisp* compiler provided that the necessary primitives (such as Define-Command) are available to the compiler. The command Load-Command operates on either compiled or interpreted files; if both exist, then the most recently modified version is loaded.

5.4 User Interaction

User interaction services allow the author of a command to control interaction with the user. This interaction comes in four forms: help, error handling, messages, and prompting.

5.4.1 Help

When a command, option, flag, or variable is defined, a documentation string is retained for the describe mechanism of the help component, and the name of the object is analyzed and the subwords retained for the apropos mechanism of the help component. Thus the help component's database is kept up-to-date with the currently defined set of objects.

5.4.2 Error Handling

Pan's execution component provides standard mechanisms for signaling errors during the execution of a command. Once an error is signaled, execution returns to an enclosing handler, usually the top level of the command processor. The basic command for signaling an error, Editor-Error, can format and print a message on the annunciator line if so instructed. New error handlers can be introduced with the form Execute-Protect.

5.4.3 Messages and Mouse Icon Manipulation

Each viewer provides an annunciator line on which messages can be displayed. This line is used by commands and by the error signaling command.

The mouse cursor icon can be temporarily changed to a new icon during the execution of a command. Three icons are currently defined to the system: the normal arrow, a rake to signify garbage collection, and a light bulb to signify a time-consuming operation. Adding a new cursor image can be done only at *Pan* creation time since it involves the allocation of static structures in the *C*-level user input/output component.

5.4.4 Prompting and Command Arguments

Input to commands can be gathered in several ways. First, arguments to commands can be designated as "promptable". If a command is invoked, and a promptable argument is not supplied in the invocation, then a pop-up prompter will appear on the workstation's display.

Second, a command can explicitly invoke a user prompt. There are three prompters—one for yes-or-no boolean values, one for *Lisp* symbols, and one for *Lisp* strings.

Third, a command can use the current selection of the buffer relative to which the command is executing. Many commands check first for a selection before resorting to a pop-up prompter.

Fourth, *Pan* has an *Emacs*-like numeric prefix argument that can be accessed by a command. The numeric prefix argument is typed before the command is invoked, and is consumed by a command invocation. Like *Emacs*, each command uses the numeric prefix argument idiosyncratically. The value and presence of the numeric prefix argument persists through an entire command execution, and is cleared at when the outer-level command completes.

5.5 File System Interaction

File system interaction is required because *Pan* has not been integrated with a broad-spectrum persistent database. Initially, *Pan* editable objects are buffers represented by text files in the UNIX file system.

File Input/Output

Pan provides methods to read buffers from and write buffers to files, to append buffers to files, to insert files into buffers, and to write or append regions to files. File system permissions are checked and observed by *Pan*.

Backup and Checkpointing

Along with the mechanisms for file input/output, there are mechanisms for creating backup files when a file is first visited, and for automatically checkpointing a buffer during editing.

Working Directory

Finally, *Pan* provides a rudimentary notion of a working directory which is a global variable. Future improvements have been proposed in order to make the working directory a buffer-local notion.

Directory Listing

There is no directory editor interface implemented for *Pan* yet. In its absence, a method is provided for listing specified directories, and for visiting files named in that listing. The listing itself appears in the help buffer's viewer.

6 Control Flow Architecture

This section describes the thread of control through *Pan* during normal processing of events.

6.1 Events

Events are triggered by changes in a viewer's environment (such as resize or repaint events) or by an input directed to the viewport having the current input focus. Events are initially handled by low-level event handlers. Each event is tagged with the small-integer handle of the viewport that has the current input focus. The viewport that received the event is called the *active viewport*.

6.2 Low-Level Event Handlers

The low-level handler examines the event and its related data. It decides whether to handle the event, to ignore the event, or to pass control to a high-level event handler having access to most of *Pan*'s internal data structures. Before the high-level handler is called, the data describing the event can be mapped or processed. The low-level handler also updates some local state information in order to respond to queries about the event.

6.3 High-Level Event Handlers

The high-level handler, if called, responds by invoking one of *Pan*'s kernel level functions associated with the event. Before calling the kernel function, several pieces of state are updated. In particular, the buffer associated with the active viewport is made the *active buffer*, and the viewer associated with the active viewport is made the *active viewer*. If the active buffer has an associated language description, that language description is also made active. Almost all of *Pan*'s commands implicitly use the active buffer and viewer; language-oriented commands use the active language description. The current implementation does not support multiple language descriptions per buffer.

The high-level event handlers associated with an event are statically defined due to implementation restrictions. In the current implementation, the kernel function for handling an event is also statically determined. Associating event-handling functions with buffers has been considered. This would allow different types of buffers to handle events differently.

Once the kernel function completes, control returns to the high-level handler, the low-level handler, and the window manager in order.

Almost all events are handled following this general scheme. However, events associated with horizontal scrolling are currently handled entirely within the SunView implementation.

6.4 An Example

Consider the following specific example. The user hits the key "D". SunView's event dispatcher calls *Pan*'s low-level keyboard input event handler. This function recognizes a keyboard hit, updates its internal data structures, and calls the high-level keyboard event handler.

The high-level handler receives the viewport handle of the viewport having the input focus along with the key code for "D". The handler cancels pending timer events and sets the active buffer to be the buffer having a viewer having the viewport to which the input was directed. The viewer attached to the active viewport is made the active viewer of the active buffer. This action makes the cursor associated with the edit viewer the active edit cursor. Finally, the high-level handler calls the kernel function `dispatch`.

The `dispatch` function consults the input state of the active viewer, and determines that the input ("`D`") is bound to the command Self-Insert that inserts the most recently typed character into the text at the position marked by the current text edit cursor. Self-Insert is invoked.

The Self-Insert command creates a single-character region and then calls Insert-Region to perform the actual modification in the active buffer's underlying textual representation.

Insert-Region performs the modification, notifies all of the visible viewers registered with the active buffer that the insertion has occurred. This notification allows the viewers to update their internal data structures, and possibly their external presentations. The results of Insert-Region are then returned to Self-Insert which returns them to `dispatch`.

If Self-Insert has an after daemon, and control returns normally, the results of the execution are passed to the after daemon. Provided that control has returned normally, the results of the after daemon (or of the command, if no after daemon has been defined) are then handed to the undo service by the command Mark-Undo.

Finally, `dispatch` clears the input sequence of the active viewer and returns to the high-level event handler which can initiate a timer event before returning to the low-level handler.

7 Acknowledgments

Many have helped with the creation of *Pan*. Thanks especially to Jacob Butcher and Christina Black. Jacob implemented the language-description processor *Ladle* and the tree data structures. Christina is developing a pretty-printing viewer and helped with the preparation of these reports. Eduardo Pelegrí-Llopart and Phillip Garrison have also lent ideas and time to the *Pan* project.

References

[1] *Windows and Window Based Tools: Beginner's Guide.* Sun Microsystems, Inc., 1986.

[2] Robert A. Ballance. Design of the Pan language-based editor. February 1986. Working paper.

[3] Robert A. Ballance, Jacob Butcher, and Susan L. Graham. Grammatical abstraction and incremental syntax analysis in a language-based editor. In *Proc. of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, 1988.

[4] Robert A. Ballance and Michael L. Van De Vanter. *Pan I: An Introduction for Users.* Technical Report 88/410, Computer Science Division, UC Berkeley, March 1988.

[5] Jacob Butcher. Ladle. In preparation.

[6] Norman Meyrowitz and Andries van Dam. Interactive editing systems: parts I and II. *ACM Computing Surveys*, 14(3):321–415, September 1982.

[7] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Trans. on Graphics*, 5(2):79–109, April 1986.

[8] R. M. Stallman. EMACS, the extensible, customizable, self-documenting display editor. In *Proc. of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, pages 147–156, 1981.

[9] Richard Stallman. *GNU Emacs Manual.* October 1986.