Coherent User Interfaces for Language-Based Editing Systems

Michael L. Van De Vanter
Robert A. Ballance
Susan L. Graham

Computer Science Division (EECS)
University of California, Berkeley
Berkeley, California 94720

July 1990

Report No. UCB/CSD 90/591

Research sponsored in part by the Defense Advanced Research Projects Agency (DoD), monitored by Space and Naval Warfare Systems Command under Contracts N00039-84-C-0089 and N00039-88-C-0292, by IBM under IBM Research Contract No. 564516, by a gift from Apple Computer, Inc., and by the State of California MICRO Fellowship Program.

Abstract

Many kinds of complex documents, including programs, are based on underlying formal languages. Language-based editing systems exploit knowledge of these languages to provide services beyond the scope of traditional text editors. To be effective, these services must use the power of language-based information to broaden the options available to the user, but without revealing complex linguistic and implementation models. Users understand complex documents in terms of many overlapping structures, only some of which are related to linguistic structure. Communications with the user concerning document structures must be based on models of document structure that are natural, convenient, and coherent to the user.

Pan is a language-based editing and browsing system designed to support development and maintenance of complex software documents. *Pan* uses a variety of mechanisms to help users understand and manipulate complex documents effectively, in terms of underlying language when necessary, but always in the framework of a coherent, user-oriented interface. A fully functional prototype is in regular use. It serves as a testbed for ongoing research on a wide variety of tools to aid document comprehension, as well as on language-based analysis methods.

Contents

List of Figures

1 Introduction

Interactive language-based editing and browsing systems will be a central and crucial component in software development environments of the future, serving as the primary interface among expert practitioners, expert systems, and the complex software being managed. They will provide access to a wealth of information created both by human developers and by increasingly powerful and sophisticated tools. A successful system of this kind must exploit the power of language-based technology in ways that enhance its users' abilities to carry out their tasks, but do not hinder their flexibility of action. In particular,the multiplicity of structures inherent in a document (or collection of documents) must be made comprehensible to the user. Since the various structures in a document overlap meaningfully, it is essential that the system provide a *coherent* user interface—one that clarifies and augments understanding without hindering the user.

Pan[1] [8] is an interactive editing and browsing system that addresses this challenge. *Pan* operates on *documents*—traditional natural language texts as well as objects with formalized structure, such as formal designs and program components. Such documents have both textual and structural aspects. Every document is encoded in one or more languages, simple text being the universal base language.

The project

Pan grew out of our research into how language-based technology can be made useful in interactive computing environments. This investigation led us to the design and construction of a prototype editing/browsing system to serve as a platform for that research. The fundamental characteristics of the prototype system demanded a number of advances in language-based technology, reported elsewhere, and some new approaches to designing the user interface. The current prototype is fully functional and is in use for our ongoing research. We are designing a successor based on our experience.

The user-centered view

We view the design of *Pan* as a user interface design problem in a deeper sense than is usually construed for this kind of system. Although it is important for the system itself to have a well-designed user interface that makes its behavior easily understood and controlled, we feel it is just as important to think of the entire system as an interface between people and documents, an interface whose purpose is to help make documents more easily understood and modified.

Thus, the design begins with consideration of intended users: their tasks, expectations, knowledge, expertise, and working environments. We developed a vision of the role to be played by language-based systems in future software development environments. The

[1]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 the software professional's tool box, the allusion to the lowly, ubiquitous kitchen utensil is apt.

primary users are seen to be software professionals fluent with their languages and tools who manage documents written in many different languages and who spend more time reading and trying to understand documents than they do authoring. The general design for *Pan* describes the kind of support we feel can be provided using language-based technology. This design involves rethinking many of the functional aspects usually associated with browsers and editors.

The prototype

Pan I [9, 10] is a fully functional prototype now in use for ongoing research in language description and processing techniques, user-interface design, advanced document viewing methods, and related areas. The prototype is a multi-window, multiple-font, mouse-based editing system that is fully customizable and extensible. It incorporates two new language description and analysis techniques: *grammatical abstraction* [7, 14] and *logical constraint grammars* [6].

Pan I addresses the combination of four fundamental characteristics: unrestricted text editing, multiple languages per editing session, an accessible language description mechanism, and a description-driven incremental analyzer that maintains information derived from each document, even in the presence of editing changes. We believe that these characteristics provide considerable leverage for developing more advanced editing and viewing capabilities that can be used to enhance a user's understanding. The system was constructed both to be usable and to provide a flexible, extensible platform for ongoing research.

With respect to our long range vision, this prototype has some limitations: current description techniques are aimed at a particular class of formal languages; the implementation supports only one language per document; a single analysis may span multiple documents, but only within one language; the system provides only part of the desired flexibility in generating visual presentations. Related issues, including support for novice programmers and learning environments, support for program execution, graphical display and editing, and the actual design and implementation of a persistent software development database, were deferred. The *Ensemble* project, which will create a successor to *Pan I*, is addressing some of them.

Within this framework, aside from addressing the technical challenges of making such a system work at all, we have been developing a number of services for users based on the kind of information the prototype can derive and maintain. These begin with our own version of services traditionally provided by language-based editors, but without many of the restrictions they impose. These services have been produced using a number of general mechanisms, whose potential for extension to new services we are now exploring.

Major accomplishments

The prototype has enabled us to realize many of the features important to achieving our goals. Those characteristics are summarized here and discussed in more detail in subsequent sections.

Pan I provides the user with unrestricted text editing, requiring none of the limitations or compromises traditionally imposed by systems that perform language-based analysis. The user need not switch editors when moving from formally specified documents to simple text, nor follow a system-prescribed discipline for authoring or modifying structured documents such as program components.

When structural information exists the system provides structure-oriented operations, but without interfering with text editing. For example, the user can smoothly mix structural browsing with textual editing. The user can also move from one language to another during a session.

Description-driven document analyzers maintain incrementally for each document a repository of information that is used to provide language-oriented editing operations and other language-based services. Information derived from other analysis tools can be incorporated as well.

Document analysis includes, but is not limited to, checking for well-formedness with respect to the underlying language. For example, one might extend a language description to check for violations of site-specific naming or stylistic conventions.

The system can be extended by the addition of language descriptions. These can have varying degrees of completeness, permitting quick prototyping and gradual enhancement.

In *Pan I* what have traditionally been called "language errors" are simply another kind of analysis-derived information, available to the user upon request. We attempt to place no more restrictions on the user in their presence than a standard text editor does in the presence of spelling errors. Analysis never "fails," and documents are never "illegal" (although they may be incomplete or ill-formed in some way).

A guiding principle has been to design and present services based on a conceptual model of document structure that is appropriate to the user. Our intent is to hide the internal complexity that supports those services. The user should be unaware of the underlying representations, such as the form of a syntax tree, and even of some of the implementation-oriented distinctions between "syntax" and "static semantics".

The author of a *Pan I* language description can use a variety of mechanisms to specify and tune aspects of the user interface for the language, such as appearance attributes and conceptual components. There may be more than one description for a given language, each suited for different tasks or classes of users.

Retrospective

One of the major challenges in designing and implementing *Pan* has been the integration and delivery of the envisioned features in a practical, usable system. We now understand how some of our early and important implementation decisions can be improved. We have made some implementation changes over time and others will be made in the successor system.

We are pleased with our success so far in presenting users with a document model that isn't visibly rooted in underlying language technology and is not subject to the same technical constraints. More and more we've been able to hide language technology, providing "services" instead. We believe this will continue to be a workable approach.

We have been struck by the importance of adjusting the user interface for each language. Careful choices of appearance make a difference, as does consideration of the major conceptual notions appropriate to each language.

A major problem with the design and implementation of systems like *Pan* is the degree to which different aspects interact. For instance, the goal of presenting conceptual models of documents to users while hiding internal complexity has tangibly affected our language description techniques, document analysis algorithms, and the design of internal data structures. One theme of this paper concerns these interactions and their affect on interface design.

Overview

This document describes our approach to the design of *Pan* and reports on our experience with the current prototype. Section 2 reviews the analysis that drove the design and shows how it moves beyond previous systems. Section 3 introduces the prototype, its organization and fundamental mechanisms. The remaining sections introduce services for users that we've developed, in each case discussing historical precedent, special design problems, and the actual implementation adopted in the prototype.

2 Goals for the *Pan* System

The *Pan* project was motivated by a particular vision of the role to be played by language-based browsing and editing systems. This section describes that vision and shows how it defines the fundamental requirements for *Pan*'s design.

The term **language-based** indicates that one or more of the facilities provided by the system makes use of language-specific information derived from the documents known to the system. In the context of this paper, the term **system** (or **editing/browsing system**) encompasses the entire collection of services that are used to browse, manipulate, and modify one or more documents interactively. The term **editing interface** refers to the fact that those services are provided to the user through a generalization of the services of a traditional interactive editor. Fraser has pointed out that the user interface to many interactive services can be modeled as a form of editing activity [26]. His examples include editors for simple tables, file system directories, and general data structures. In that sense editing is a fundamental part of an integrated software development environment.

2.1 The Working Environment

Pan is intended to support experienced professionals who manage large collections of inter-related documents. Many common assumptions about traditional language-based editors do not hold in this domain. In particular, the kind of support most helpful to experienced users is different from the support needed for beginners. In addition, the primary task for experienced professionals is comprehension and modification of existing documents, rather than authoring of new ones.

Neal [47] distinguishes three dimensions of expertise in users of program editors: computer, language, and programming expertise, in addition to expertise with a particular tool. When designing *Pan*, we assumed a high level of expertise with computers (and eventually with *Pan*) and a relatively high (but not uniform) level of expertise with languages and with programming.

Yet even experienced users are confronted with unfamiliar languages or novel situations when they need the support normally provided for novices. We believe that the kinds of analysis and support necessary to support experienced users can be adapted to support novices. The converse—supporting the experienced practitioner using tools tuned to the novice—is far more difficult.

Understanding is the primary activity

Editors tuned for authoring fail to address today's problems. Software systems have become so large and complex that developers spend far more time trying to read, understand, modify, and adapt documents than they do creating them in the first place [29, 66]. A successful interactive development environment must support understanding by recognizing, exploiting, and making visible complex relationships within and among related documents [63]. At the same time, the system should provide related, but different support tuned for authoring and modifying documents.

There are many languages

Software developers typically use several formal languages daily: design languages, specification languages, structured-documentation languages, programming languages, and small languages for scripts, schemas, mail messages, and the like. Even a program written in a single programming language may contain embedded "little languages" that impose their own conventions. For example, many subroutine packages such as libraries for developing window-based applications effectively define mini-languages that determine how subroutine calls, and in particular long sequences of arguments, must be used. A language-based system must support all these different languages as smoothly and uniformly as possible.

Further, it must be convenient to add support for new languages using natural, declarative, language-description mechanisms. New languages, or extensions or modifications to existing languages, arise frequently. Declarative descriptive mechanisms allow a description writer to focus on what is being described rather than on how document analysis proceeds.

Finally, since the services that the system can provide are based on the data it derives from a document, a language *description* may include elements beyond the scope of the basic *definition* of a language. Thus, someone adding a new service to the system may need to extend some existing language descriptions to derive new data or maintain new annotations.

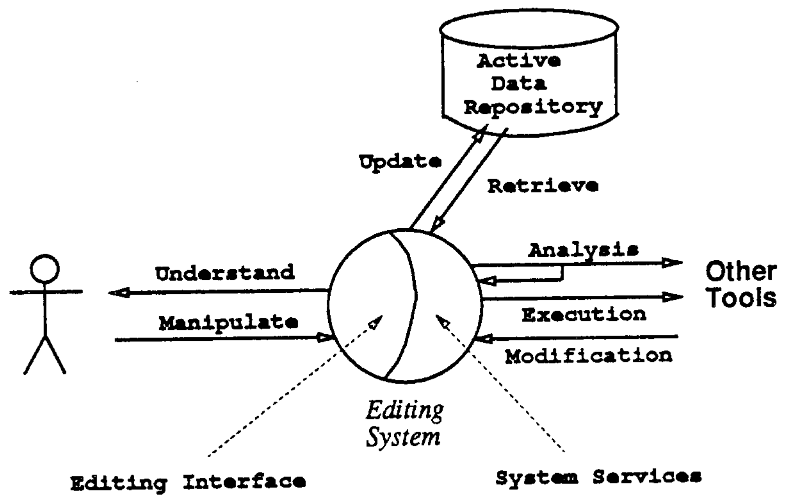


Figure 1: Editing interface and system services in relation to the environment

Users are fluent

Users know their primary languages and tools well. An editing interface must augment productivity, without any sacrifice of flexibility and power in the name of safety or learnability. For example, experienced developers will not trade away the flexibility of unlimited text editing for the safety of enforced syntactic correctness. Finer grained support should be available in situations where the user is dealing with an unfamiliar language.

2.2 The Role of the System

A language-based editing/browsing system provides the primary interface between people and integrated environments containing the documents they manage. Positioned this way, between user and documents, the system is uniquely situated to share information *about* documents that may be provided by both user and tools, as shown in Figure 1. In this model, users interact with documents through the editing interface; tools interact with documents through the system services; they communicate with one another via an active data repository. The editing interface and the system services provide alternate projections (views) of the document as well as analysis for the user.

Gathering and presenting information

Users opportunistically exploit many forms of information to help them understand and modify complex documents [42]. "Information gathering" is the primary task associated with important activities like program maintenance [34]. The system can support these activities by gathering and presenting many kinds of information.

For example, many language-based systems check that a document is well-formed. The same analysis can enable a user both to edit the document in terms of its underlying language and to locate document components that violate restrictions in the formal language definition. Other kinds of interaction may require more elaborate analysis. For example, language-based formatting (sometimes called prettyprinting), traditionally based only on surface syntax, should exploit information about scopes, types, local usage, or even distinctions such as "main-line" vs. "error-handling" code. These kinds of analysis move far beyond simple error-checking: they involve knowledge of particular organizations, techniques, and systems, not just languages. Although this kind of information must be broad in subject domain, it need not be deep (in the sense of program *plans* [43, 57] or *clichés* [53]) to be useful.

Maintaining and sharing information

Complex, expensive analyses to support an editing interface make sense only in an environment in which many tools share the information maintained by the system. The same type checking that is used to tell the user that a document is type-correct can provide type information to a compiler, an interface consistency service, or an auditing tool. In some cases information produced by other tools should be made available to the editing interface. For example, the results of performance analysis or information derived from version control can be used to produce helpful views of programs or prototypes.

2.3 THE STRUCTURE OF DOCUMENTS

Documents represent richly connected, overlapping webs of information having many structural aspects. Each aspect is more relevant for some kinds of users than for others and for some tasks more than for others. An editing interface in the role we envision must support many kinds of users, many tasks, many structural aspects.

The multiplicity of structural aspects has the potential to confuse the user, but it need not if the system supports structures already understood by users. People routinely think about complex objects from different perspectives and are remarkably adept at shifting perspective. An effective editing interface need only support these shifts without imposing any extra overhead on the user.

This discussion reviews the document aspects most important to users who are the experienced professionals in *Pan*'s intended audience.

Textual display

Text remains the display medium of choice for documents in most languages, since most languages are by definition textual. Graphical presentations of such documents have some potential value for overviews and summaries, but it is very unusual for a graphical display to carry *all* the information present in a textual rendering and it is usually not a comfortable visual field in which users can specify editing operations[2].

[2]This remark does not apply, of course, to those languages which are by definition graphical. For those languages, graphical display is the medium of choice.

An effective editing interface should exploit the full power of the textual medium. The value of high-quality typography for natural language documents is well established. Recent studies suggest the same potential benefits for programs [3, 50]; these studies involved printed versions of programs typeset from source code by language-specific formatters.

Text-oriented editing

An effective editing interface must support text-oriented editing with as few restrictions as possible.

The history of language-based editors is marked with controversy over text-oriented editing. In practice, people will not do without it. It fits naturally with the textual display medium, and people are accustomed to it. Furthermore, most kinds of documents contain textual chunks that have no structural properties beyond the textual. Examples include sentences or paragraphs in most natural language documents, labels in spreadsheets, and comments in programs.

Despite arguments to the contrary [65, 67] many language-based editors imposed varying degrees of restrictions. Some of these projects have since concluded that text-oriented editing should not be limited [5, 15].

Language-Specific Structures

Language-specific structures represent relationships among the parts of a document, or even among documents, that are defined by the underlying formal languages in which the documents are encoded. For example, consider the relationships in a natural language document defined by the connection between a figure and references to it, the relationships between declarations (definitions) and uses of variables in a computer program, the structures represented by a call graph in a program, or the relationships among grammatical units as specified by a formal syntax for the language being edited. Although the syntax of a language has often been taken to be the primary (and most interesting) decomposition, other language-oriented structures are at least as important to the user.

One useful aspect of syntactic structure is that it defines a hierarchical decomposition of each document into structural components (sections and paragraphs, for example, or blocks and statements). The formal definition of the syntax of a language is an artifact of language-description techniques and may not always produce structural components that are consistent with the way people think about document structure. In particular, there are aspects of structure that the user may wish to ignore. An effective language-based system must present and allow manipulations of syntactic components when appropriate, but must do so in terms of a conceptual model of documents that is natural and convenient for users.

For the implementor of a language-based system, the distinction between the formally-defined syntax and other derived structures is quite convenient. For users, it is useful to blur that distinction since their understanding of "the syntax" of a language may be far removed from the formal definition used by the system for internal processing.

Other Structures

A variety of other decompositions can induce structures within or among documents. For instance, editors like ED3 [59] and the Cedar editor (Tioga) [62] allow users to specify explicitly a hierarchical decomposition for each document, where the structure may or may not be correlated with structures in the underlying formal language. Outline processors are editors for precisely this kind of structure where the underlying language is simple text.

Another interesting structure arises when a single document is encoded in more than one language. In this case, the relationships across languages and their appearances within the document become interesting. The programming environment Mentor [22, 24] supports such nesting of languages.

Finally, relationships within and among documents such as those generated by hyperlinks provide a third example of document structures that are neither textual nor language-induced.

2.4 PRAGMATICS

Other issues must be addressed for an editing interface to be practical and usable in *Pan*'s intended role. Some of these issues have been complicated in *Pan* by the necessity for an early implementation to serve as a research platform.

Open Architecture

An effective system, especially one used for research, must be built on a flexible framework designed to accommodate many kinds of variation and evolution [30, 40].

Customization and Extension

A usable editing interface must be customizable and extensible to accommodate the enormous variations among individual users, among projects (group behavior), and among sites [40, 58].

Performance

Experience has shown repeatedly that an editing interface must be "acceptably" fast. Users are seldom willing to compromise, even in the name of additional or improved functionality.

Integration

In the modern workplace potential users often view a new tool from the perspective of a rich, well-established working environment. An editing/browsing system whose user interface departs dramatically from those to which users are accustomed suffers the handicap of perceived "isolation". An effective editing interface must provide an easy transition path for those who have used other systems and for those who will continue doing so.

An automatic way to import data into the system is required so that existing documents can be incorporated. Ways to export data from the system are also necessary to make use of facilities not provided by the system.

2.5 Precursors

Development of *Pan* began in 1984. At that time, a number of language-based editing systems had already been implemented or designed. The systems that influenced *Pan* can be roughly classified into three categories based on underlying models of editing: display-oriented text editors, syntax-directed structure editors, and syntax-recognizing editors.

As important as the underlying model is the choice of how to delineate and maintain structural information. It can be done *explicitly*, by interpreting the sequence of user interactions as a document is constructed, or the structures can be *derived* by a component of the editing system that examines some representation of the document after it has been constructed. Both approaches can coexist in the same editing system. For instance, Mentor [22, 24] can derive structure initially from a textually represented document, but structure may only be edited using explicit structure-oriented operations.

The two choices, explicit maintenance of structure versus derivation from some other representation, are matters of implementation but they affect the design of the user interface. For example, *syntax-directed editors* rely on the user to explicitly construct the structures that they are then allowed to edit while *syntax-recognizing editors* attempt to derive a structural representation for the document being edited. A purely *text-oriented* editor makes no real use of linguistic structures at all; its language-based operations are based entirely on examining the surface representation of a document, perhaps inferring a very simple syntactic decomposition.

Text-Oriented Editors

A **text-oriented editor** operates on a document modeled both as a stream of characters and as a two-dimensional plane of characters. (The coordinates in this plane are commonly called "lines" and "characters within a line".) Users are free to operate on any character at any time.

While groups of characters such as words or lines might be accorded special status and operations, no special structural constraints are imposed on the document and few well-formedness constraints are enforced. Bravo [39], EMACS [58], MacWrite [2], and Z [67], are examples of text-oriented editors. Both EMACS and Z provide some language-specific editing operations.

Language-based text editors provide commands that operate on easily recognizable elements of a language, but they do not provide syntax-oriented commands or deeper analysis of linguistic structures, thus limiting the kinds of services provided.

Structure and Syntax-Directed Editors

Structure editors present a document as having a definite internal structure, with editing operations modeled as operations upon that structure. Most often, the document is tree-structured, with operations defined on subtrees. Outline processors are structure editors for tree structures; spreadsheet editors are structure editors for tables. Structure editors may or may not interpret the structures that they edit. When they do attribute language-specific meanings to the structures, they are often called syntax-directed editors.

A **syntax-directed editor**[3] is a structure editor that requires that the document be syntactically correct at all times: editing operations must "follow" the syntax of the language. As in a pure structure editor, the user can add new material to a document only at those points where it can be successfully grafted onto the existing structure. ALOE [45] and the Gandalf editors [49], Centaur [12], the Cornell Program Synthesizer [60] and the Synthesizer Generator [52], Emily [31, 32], Mentor [23] and PSG [4] are all syntax-directed editors.

The syntax-directed approach greatly simplifies editor design by restricting ("narrowing") the changes users can make. However, some syntax-directed editors provide limited forms of text-editing beyond that strictly required for program entry. For instance, in the Gandalf editors and the Cornell Program Synthesizer, expressions may be entered as text and then parsed for correctness [37]. While the syntax-directed development style is useful in some cases, (e.g., in environments for novice programmers or in environments for constructing mathematical proofs [19]) the practicing programmer has more general needs.

Syntax-Recognizing Editors

A **syntax-recognizing editor** [13] derives structural information from text, in order to check for correctness without necessarily demanding the consistency constraints of a syntax-directed editor. Syntax-recognizing editors can provide structural operations as well as text-oriented editing. Babel [35], the Saga editor [38], SRE [13], and Syned [27] are all syntax-recognizing editors, as is *Pan*.

The syntax-recognizing approach[4] permits text-oriented editing by users at any time in any context, but at the cost of considerable complication in document representation, incremental analysis algorithms, and user-interface design. A common problem in early syntax-recognizing editors was failure to hide this complexity from the user.

3 The *Pan* System

From our conception of the role to be played by *Pan* we created a vision of what *Pan* should be. To its users it should appear to be a fast, convenient, fully-functioned text editor that happens to be extremely knowledgeable about the local working environment: the many languages in use, local conventions, and perhaps the user's own personal working habits. One might use *Pan* for editing text all day, without ever giving a thought to its other capabilities. But at any time one might choose to broaden the dialogue with *Pan* to draw

[3]The term "structure-oriented" [48] is sometimes used in place of "syntax-directed".

[4]The syntax-recognizing approach does not preclude a user-interface that simulates syntax-directed editing. In fact, syntax-directed editing can be provided easily in a syntax-recognizing editor.

on many kinds of information (maintained by *Pan*) *about* the document. *Pan* could then be directed to use this information to guide editing actions, to configure and selectively highlight the textual display, to present answers to queries, and more.

Technical challenges

Implementing this vision required solving three significant technical problems in combination:

- Supporting full featured text- and structure-oriented editing;
- Performing incremental analysis to maintain a database of information about each document;
- Supporting multiple languages, preferably using declarative language descriptions.

To preserve the appearance of being a "smart text editor" while providing the benefits of language-based editing, *Pan I* is a *syntax-recognizing* editing system. All language-oriented structures, including the primary syntax used to drive other analyses, are derived from analysis (parsing) of a textual representation. Structure-oriented (and even syntax-directed) editing are implemented on this base.

Adoption of syntax-recognition did not solve all of the problems. One major problem in syntax-recognizing editors was their need to keep large and unwieldy internal representations of the syntax in order to provide incremental (local) reanalysis. No such problem occurs in purely syntax-directed editors. The theory of *grammatical abstraction* [7, 14] was developed to resolve this problem in *Pan*.

Maintaining derived information in the presence of change was a second technical hurdle. Structure-oriented and syntax-directed editors localize the extent and impact of changes by localizing the user's editing actions, an approach that is antithetical to *Pan*'s goals. To solve this problem, the notion of *logical constraint grammars* [6] was developed.

In a logical constraint grammar, goals expressed in a logic programming language are associated with structures in the formal abstract syntax. Goals express contextual constraints on the language or design of the document being edited, and can be used to establish linkages between non-local components in a document or between documents. Constraints express the well-formedness requirements of the language definition, but may also describe extra-lingual constraints, such as site or project-specific naming conventions.

During editing, the satisfaction of all goals associated with *instances* of structures appearing in the document is attempted. Instances of structures are represented by subtrees of the internal tree. The document is considered "semantically" well-formed whenever all of its associated goals are satisfiable. Information gathered during constraint enforcement is retained in a logic database. A consistency maintenance mechanism revises the database and reattempts goals as documents are altered. The information retained in the database is used by other language-based services in *Pan*, such as the user-centered viewing tools.

User-Interface Goals

As important as the implementation challenge, however, was the design and delivery of user-oriented services of the kind suggested by the discussion in Section 2. This presented a double-edged problem in user-interface design.

First, the design of an effective user interface for any interactive system is challenging, especially for one with conflicting goals. For example, *Pan*'s design called for a model of text editing that is familiar to a wide variety of potential users, but it also demanded the integration of rich new functionality at all levels of the system[5].

Second, we view the the entire system as an interface between user and document, subject to some of the same considerations but at a different level. *Pan*'s ultimate goals depend on the quality of its user interface at both levels.

As *Pan*'s design proceeded, common themes emerged that shaped the user-interface.

- Provide full-featured text editing based on familiar models.
- Don't try to "do too much" [47]. Derive information from document text, but present it only upon request.
- Use *Pan*'s rich repository of derived information to build specific user-oriented services, but don't expect users to know much about the repository or analysis methods.
- Communicate with the user in terms of a useful conceptual model of documents rather than *Pan*'s representation model, particularly when communicating about structures other than visible text.
- Provide generic services and a uniform interface for all languages, but make it possible to tune the interface for each particular language and for each class of users.
- In the spirit of *user-centered system design*, treat ill-formed documents not in terms of user "errors" but in terms of successive approximations of what the user intends [44]. Maintain full service in the presence of "errors," including but not limited to the presentation of diagnostics.
- Allow *Pan*'s users to customize the user interface and control important policies. Provide for experimentation with new services defined upon existing infrastructure.

Many of these embody diSessa's principle of *continuous incremental advantage*: do not confront the user with steep learning thresholds, but provide a steady learning progression in which each quantum of learning offers immediate benefits [21].

System Organization

Figure 2 shows how the *Pan* system is organized from the perspective of its users. This

[5]For example, complex low-level data representations that could efficiently support incremental document analysis had to be implemented and refined. Support for document analysis was one major reason that we were unable to build directly on top of an existing editing system.

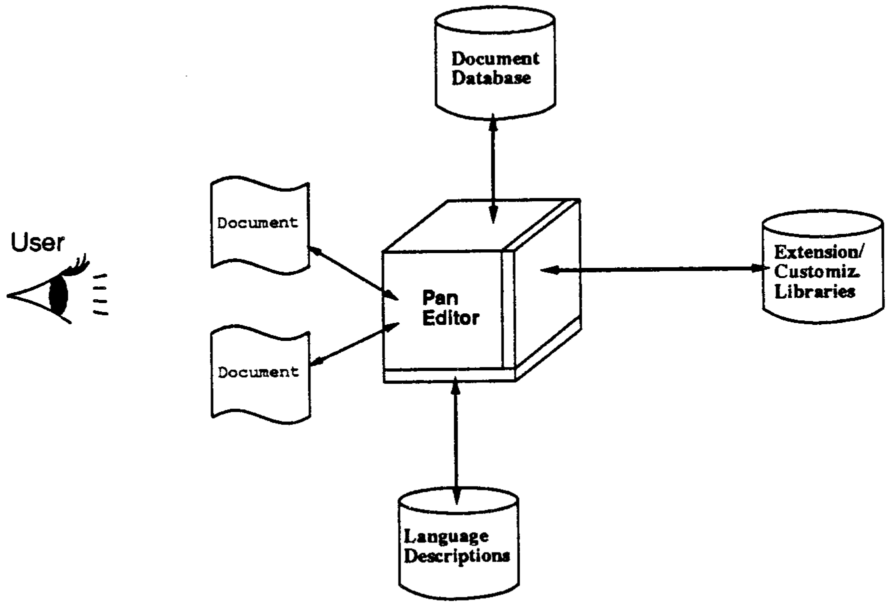


Figure 2: The *Pan* system

organization supports three different kinds of users: clients, customizers, and language-description authors.

Clients use *Pan* to edit documents: formal designs, programs, and manuscripts. For clients *Pan* is the interface to a document database, as shown at the top of Figure 2. Clients may use libraries and language descriptions but need to know little about them. All but the most naive clients, however, customize their environments in some way, blurring the boundary with the next category.

Customizers provide new services by adding to the extension and customization libraries, shown at the right of Figure 2. The expertise required ranges from the shallow (adding new groups of key bindings, for example) to the deep (adding a new kind of directory editor, for example).

Language description authors add languages to *Pan*'s collection, shown at the bottom of Figure 2. They need some expertise in language processing and with the specific techniques used by *Pan*'s document analyzer.

The next two subsections briefly describe *Pan*'s facilities for customization and for language description.

Customization and Extension

Although language definition is *Pan*'s most technically intricate form of extension, other facilities support user customization, simple extension, and rapid prototyping of new editing environments.

- *Bindings* for keystrokes, mouse buttons, and menus[6] as well as generalized *option variables* permit extensive personal customization using simple declarations. In fact, the standard behavior of the editor is determined entirely by a configuration file using the same declarative mechanisms.

- *Pan*'s configuration variables and bindings are *scoped*. Values may be assigned at one of three nested levels (buffer instance, buffer class, and global), and are dereferenced dynamically to support runtime reconfiguration.

- Simplified *command definition* facilities in *Pan*'s extension language (a superset of the underlying COMMON LISP language) make it possible to add new layers of functionality in straightforward ways. These facilities automatically integrate extension programs with a generalized undo facility, with generic exception detection and recovery mechanisms [64], and with *Pan*'s elaborate help system.

- A *run-time library* permits special editor environments and extensions to be loaded dynamically.

These facilities, combined with *Pan*'s rich infrastructure, comprise a toolkit for editor construction and for experimentation with user-interface designs.

Language Description

Adding new languages is the most important extension mechanism in *Pan*, supported throughout by description-driven system components. All aspects of language description are declarative for readability and convenient modification.

A *Pan* **language description** supplies information for several aspects of document analysis and user-interface configuration: lexical and grammatical information describes the syntax of the language as used by *Pan*, contextual constraints describe extra-grammatical well-formedness constraints (commonly called the *static-semantics* of the language) and other language-oriented structures, and user-interface definitions are used to tune the user interface.

Of particular interest is the distinction between syntax and contextual constraints[7] inherent in *Pan*'s language description techniques. The description of syntax underlies both the definition of contextual constraints and some of the user-interface definitions. Services built upon derived information may rely upon data derived during satisfaction of contextual constraints.

One portion of these descriptions configures the user interface for the language. This information can include a categorization of the structures appearing in the abstract syntax (corresponding roughly to the phyla of operator-phylum trees [36], but potentially overlapping) as well as templates for structural elaboration.

[6]Keyboard bindings are crucial, since expert users seldom use the inherently slower menu systems [40].

[7]Almost all language description techniques similarly distinguish syntax and semantics. Section 10 discusses some of the shortcomings resulting from this distinction.

The language description facilities developed for *Pan* are presented in detail elsewhere [6, 7, 14]. For this discussion, the following features are noteworthy:

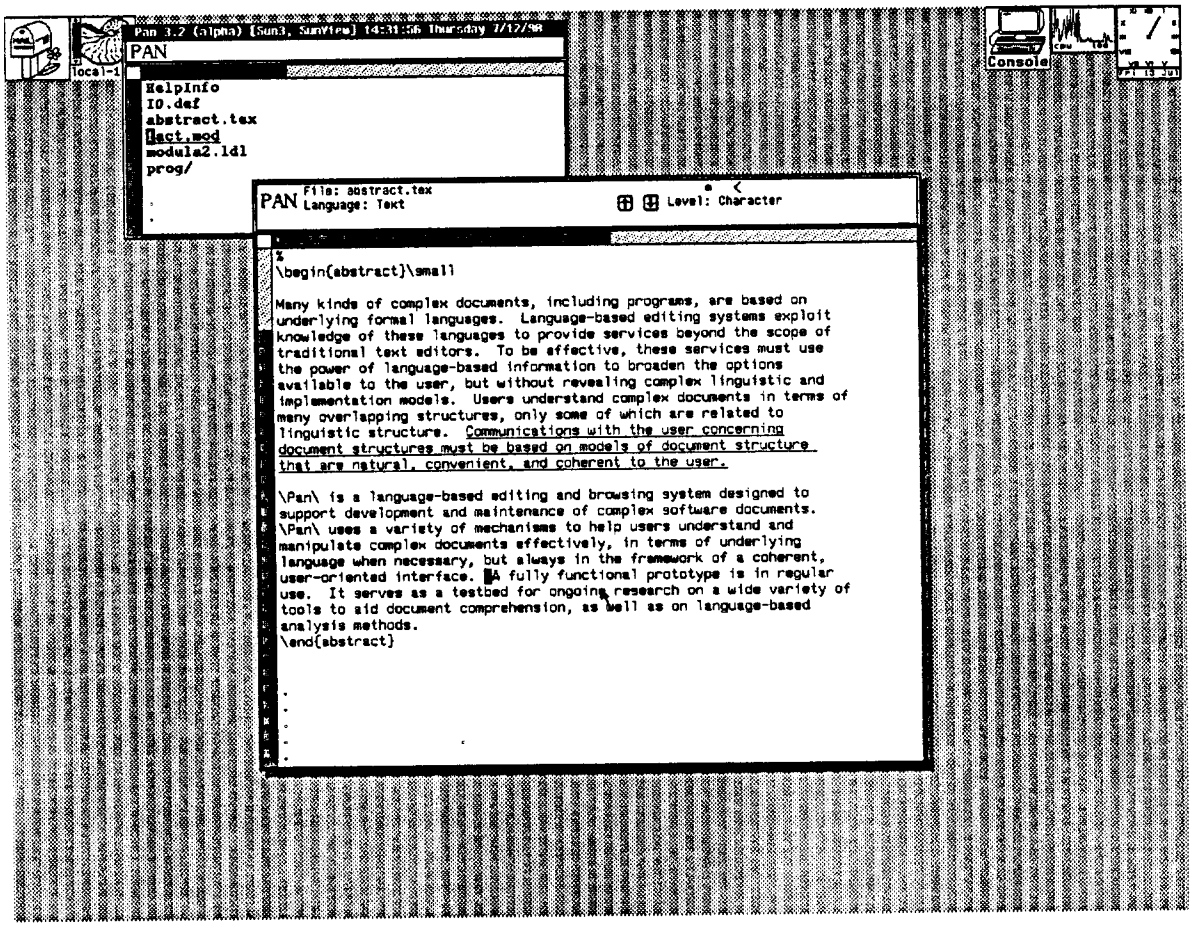
- Different formalisms, each suited for different aspects of language descriptions, are used.
- Different language descriptions for the same base language can coexist. When the same base syntax is used, this can correspond to a "layering" of language descriptions. When a different base syntax or different user-interface description is used, this corresponds to a slightly different language as seen by a user.
- The language description techniques and associated document analysis algorithms support the ability to edit documents composed from multiple languages (but the current implementation, *Pan I*, does not).
- Default definition and handling of "errors" in ill-formed documents is automatic. However, the error-handling mechanisms can be tuned for each specific language description.
- Language descriptions can be loaded dynamically during an editor session, although standard language descriptions are often preloaded.
- Language descriptions are written in two formal languages (Ladle and Colander) for which *Pan* also offers language-based support. *Pan*'s document analyzer is used as the preprocessor for one of these, and it may be run as a batch program.

To date, syntax descriptions have been written for Modula-2, ASPLE, "C" (except preprocessor macros), FIDIL [33], and for *Pan*'s language description languages. A complete language description for Modula-2 (including the language's contextual constraints) is available; other complete descriptions are being developed. Finally, like other aspects of the prototype, improvements to our language description mechanisms are underway. These are discussed in Section 10.

4 Text Editing

One of the first commitments in *Pan*'s design was to provide full-functioned text editing services that people would be willing to use. A closely associated commitment was to prevent *Pan*'s rich language-based features from ever interfering.

Figure 3 shows how *Pan I* appears during ordinary text editing. The same text-oriented services are available in every document **buffer**, whether or not language-based analysis is taking place. As a matter of policy derived information is never used to hinder text-oriented services, so it is possible to consider the *Pan* text editing interface independently of any language-based features.



```
Pan 3.2 (alpha) [Sun3, SunView] 14:31:56 Thursday 7/17/90
```

```
HelpInfo
IO.def
abstract.tex
fact.mod
modula2.ldl
prog/
```

```
PAN File: abstract.tex                                  Level: Character
    Language: Text
```

```
%
\begin{abstract}\small

Many kinds of complex documents, including programs, are based on
underlying formal languages.  Language-based editing systems exploit
knowledge of these languages to provide services beyond the scope of
traditional text editors.  To be effective, these services must use
the power of language-based information to broaden the options
available to the user, but without revealing complex linguistic and
implementation models.  Users understand complex documents in terms of
many overlapping structures, only some of which are related to
linguistic structure.  Communications with the user concerning
document structures must be based on models of document structure
that are natural, convenient, and coherent to the user.

\Pan\ is a language-based editing and browsing system designed to
support development and maintenance of complex software documents.
\Pan\ uses a variety of mechanisms to help users understand and
manipulate complex documents effectively, in terms of underlying
language when necessary, but always in the framework of a coherent,
user-oriented interface. A fully functional prototype is in regular
use.  It serves as a testbed for ongoing research on a wide variety of
tools to aid document comprehension, as well as on language-based
analysis methods.
\end{abstract}
```

Figure 3: Editing Text using *Pan I*

Models of text editing

Designing a usable text editor is by itself a significant undertaking. Among the many challenges is the need to exhibit a coherent conceptual model of what is being edited, a model reflected by both visual presentation and editing operations. *Pan* must also present an effective model of language-oriented editing, subject to the requirement that it not clash with the textual model (see Section 6). Models clash when they suggest conflicting interpretations of how the editor is behaving or should be expected to behave.

Although seldom explicit, the design of effective conceptual models for text editors has always been a problem. For example, some of the earliest interactive editors operated on a virtual deck of punch cards; users could think of these editors as card punches with an erasing backspace key. Two important historical improvements added support for lines of varying length and eventually replaced the line-oriented model with a stream model. Meyrowitz and van Dam argue that the long term significance of these changes was that

> displayed text was no longer considered to be a one-to-one mapping of the internal representation, but rather a tailored, more abstract view of the editable elements [46].

The conceptual distinction between a one-dimensional text stream (the "editable elements") and a virtual two-dimensional page (the "displayed text") usually makes itself evident in the mysterious and idiosyncratic behavior associated with "whitespace" characters (spaces, newlines, and especially tabs). For example, many text editors require that space characters be inserted explicitly at the end of a line before the cursor can be positioned in the right margin. This kind of behavior contributes greatly to the difficulty novices encounter when learning text editors, especially when misled by the "typewriter" and "blank page" metaphors [25, 55].

This bit of historical reflection offers two lessons for the design of *Pan*. The first is that a well designed user-interface can compensate for an apparently inconsistent underlying model. People become quite proficient with text editors, presumably building something like diSessa's *distributed models* [20] of the editing domain. The effect is so powerful that experienced users of text editors are often tempted to instruct novices with obviously inaccurate metaphors like "blank page". The second lesson is that learning to use any text editor is a difficult task, one that should be expected of users as seldom as possible.

The text editing interface

The *Pan* text editing interface appears to the user as a bit-mapped, multiple-font, mouse-based text editor with multiple windows, in the spirit of Bravo [39] and its many successors. A *Pan* user may open any number of **windows** onto each buffer's virtual two-dimensional text display. All windows on a buffer share a single, visible **selection** that appears as underscored text in any window in which it happens to be visible. Each window has its own scroll position and text **cursor**, both of which persist when the window is made invisible. Some text-oriented commands operate exclusively on the selection; others, including ordinary character insertion, operate at the cursor. All editing commands are undoable.

The text model

Pan's text editing model is a hybrid based on two familiar models: the Macintosh [56] and EMACS [58]. Like both of those and many other editors, *Pan* treats text as both a stream and a two-dimensional page. Like Macintosh editors, *Pan* distinguishes the insertion point from the selection[8], a global **clipboard**, and supports the menu-driven commands Cut, Copy, and Paste. Like EMACS, *Pan* offers a rich set of text-oriented editing commands and EMACS-compatible key-bindings [9].

The hybrid text model was developed to ease the cognitive burden on users, the intention being that *Pan* would behave like a familiar text editor until the user began to request additional language-based services. The combination has succeeded in some ways, but has revealed model clashes in others. For example, the model of Undo supported by *Pan I* and by the Macintosh editors differs from the EMACS Undo. *Pan* doesn't support *pending delete* selection, but could at the cost of confusion to EMACS users. Both *Pan* and EMACS have text cursors but their behaviors differ in minor ways, as do the semantics of the *kill ring* and the correspondence between buffers and windows. A user can ameliorate somewhat the effects of these clashes through customization, in particular by hiding features that cause confusion.

Displaying other information

Each *Pan* window may display optional **panel flags**, modeled on physical control panels. In Figure 3 the flag "*"[9] is on, meaning that the document has been changed since last saved. The flag "<" is also on, meaning that automatic text-filling (line-wrapping) is in effect.

Pan's text display, in addition to its conventional role for text-oriented editing, plays an important role as a medium for displaying derived information about documents. *Pan* associates with each character a font code, mapped indirectly to fonts via a user-configurable **font map**. Fonts of differing heights and widths can be mixed freely. Like other options in the *Pan* editor, font maps are scoped by buffer, by buffer class, or globally. Additional facilities are available for superimposing more information upon the textual display: underlining, stipple patterns, colored inks, and color background shading (in the manner of "highlighter" pens). Background shading may be selected orthogonally to ink color.

5 Structure vs. Presentation

A central problem in the design of any interactive editor is that the structure of an object being edited seldom maps nicely to the object's **presentation**, the visual display of the object created by the system. The previous section described how this discrepancy makes ordinary text editors difficult to learn.

[8]This conflicts with the EMACS model where the insertion point *defines* one boundary of the selection.

[9]The character "*" is actually the default visual *appearance* of the flag whose internal name is Object-Modified?. Like much of *Pan*'s user interface, both the presence and appearance of flags is configurable.

For documents with the kind of rich structure we described in Section 2.3 the problem is potentially much more troublesome. For example, syntax-directed editors present a model in which the user manipulates a tree; every operation carries the additional cognitive overhead of understanding the complex relationship between the tree and its two-dimensional textual presentation. This creates a serious clash between the editing model (what the user can do) and the presentation model (what the user sees).

Document presentation

One of *Pan*'s goals for document presentation (as distinct from editing, which will be described in more detail in Sections 7 and 9) is to guarantee the presence of an editing model (possibly in addition to other language-oriented editing models) that is structurally similar to the textual presentation. In other words, if a document appears textually, than it can be treated that way independent of any other language-oriented information derived from it. Even as we explore more elaborate methods for transforming language-based structure into text (for example program *unparsing* [28]), we would like the user to be able to think of the display as just the text it appears to be.

In *Pan I* that consistency has been a natural side effect of our first implementation strategy and our deferral of more interesting presentation services. The textual presentation in the current prototype is based on a text stream in the usual way, and the only whitespace is that which the user inserts explicitly. *Pan*'s recently added prettyprinting mode complicates this a bit by rearranging user-supplied whitespace, but it does so in the context of a familiar and predictable service similar to indenting modes in language-based text editors like EMACS.

Pan's use of fonts, point size, and color does not undercut this consistency because these visual attributes are independent of the text-editing interface and are not controlled directly by the user. New text created by the user typically has default attributes (standard font, no enhancements); the system changes the attributes to display information as requested by the user.

Displaying language-based information

So complete is the reliance on the textual presentation, that it might not be apparent at all when language-based information is present. Rather than hide the fact completely, *Pan*'s default configuration adds the panel flag "T" (shown in Figure 5) when structural information is being maintained.

Although a user might edit a document textually without thinking of language-based information at all, a number of useful services are available that display language-oriented information about documents. This information may be communicated to the user using a number of typographical **presentation enhancements**, which usually do not change the contents of the text.

For example, *Pan* supports two familiar enhancements for program documents:

- Following established custom [3], **font shifts** reveal the lexical category of text: language keywords, identifiers, and comments. It has been our experience that font shifts contribute significantly to program readability, but that font maps must be tuned for individual languages.

- When **prettyprinting** is in effect, *Pan* rearranges document whitespace after each analysis (Section 7 describes *Pan*'s analysis strategies), using indentation in familiar ways to reveal syntactic nesting levels. More advanced forms of prettyprinting for program documents, including semantically-driven elision, are under development [11].

A general form of **structural highlighting** supports other enhancements. Arbitrary groups of structural components may have their associated text rendered with one of several (generally independent) special effects: background color, stipple patterns, and ink color. For example, the user might see the answer to a particular query presented as a shift to blue ink for some text, independent of yellow background shading that might be continuously highlighting components of interest for other reasons (for example, those described in Section 8).

Pan's use of color in document presentations is consistent with the general recommendations for program typesetting suggested by Baecker and Marcus [3]. However we have found one of the options they describe, the use of color to reveal lexical status, to be inappropriate. We believe that documents in ordinary states should be rendered entirely in black and white, with color reserved for exceptional situations, under user control as always.

Open issues

As we extend *Pan*'s presentation model to include more elaborate display mechanisms, the natural consistency between presentation and editing model enjoyed by the prototype becomes more difficult to support. For example, as blank areas in the presentation become less coupled to whitespace characters inserted explicitly by the user, it becomes less clear how to support smooth text-oriented editing in those blank areas. We expect simple heuristics to suffice for this situation.

A related but slightly more interesting problem arises when certain text has special properties, such as the "placeholders" displayed at unexpanded structural sites by syntax-directed editors. Those editors typically do not permit the user to edit text that represents placeholders. Preliminary experiments with an emulator for this style of editing suggest that the restriction is not necessary in *Pan*. Textual placeholders can be added implicitly to a language description, following some simple spelling convention such as "@statement" for an unexpanded component of class "statement." The system may add these when inserting "templates," but the the user is also free to type them literally. A mistyped or misplaced placeholder becomes just another variance relative to the language description (see Section 8).

Elision is more problematic yet, since some characters, the conventional '...' for example, can represent hidden information. Here it is harder to define a natural model for text-oriented editing, since the characters are artifacts of the presentation only. In this case an attempt to edit the elision symbol might trigger expansion of the elided portion, perhaps preceded by a query to the user.

A related situation arises when the keywords in a language may be treated as just another aspect of language presentation, for example if keywords are presented visually in the user's choice of natural language (perhaps French instead of English). In this case the editing/presentation correspondence can be preserved, but the implementation becomes more complex.

Structure-oriented editors encounter many of these problems, but they typically lack *Pan*'s commitment to smooth integration with the textual presentation model. Our follow-on project is exploring these issues, both with implementation strategies and with experimentation to discover what presentation models "feel natural."

6 User Models of Document Structure

User and editor must be able to communicate about what is being edited. When presentation and structure are similar enough, as they are for familiar text-oriented editing, simple kinds of navigation like pointing are obvious and effective. In contrast, language-based document structure is both less familiar to users and less evident from textual presentations.

Making language-based structure intelligible requires that the system present it in terms of coherent conceptual models. The design of such models is a user interface problem, applied here to the structure of formal languages. This section describes *Pan*'s models for the simplest kind of language-based structure, decomposition into syntactic components. Examples of more complex structure arise later, for example in the discussion of ill-formed documents in Section 8.

Ignoring for the moment problems that arise from the mixture of textual and structural models (see Sections 7 and 9), there are two important aspects of this user model: how documents decompose into components and how those components are named.

Internal representation

Most language-based editors present a document model that is based on the way they represent documents internally, typically as a tree. For example, many early editors that provided unrestricted text editing maintained a complete parse tree [35, 38], whereas syntax-directed editors usually maintain a reduced or **abstract** syntax tree (an example of each appears in Figure 4). Arguments for the latter representation include the observation that it is a more "natural" model for user interaction, since the extra information in the parse tree is only an artifact of parsing technology. *Pan*'s internal representation is based on an abstract tree[10], and *Pan*'s language description mechanism permits considerable flexibility in the design of this representation for each language.

Experience with language descriptions for *Pan* suggests that criticism of the parse tree representation as a user model applies to the abstract tree as well. The design of tree representations is typically driven by many issues concerning language description and analysis.

[10]Combining incremental parsing with an abstract tree representation demanded novel specification and implementation techniques [7].

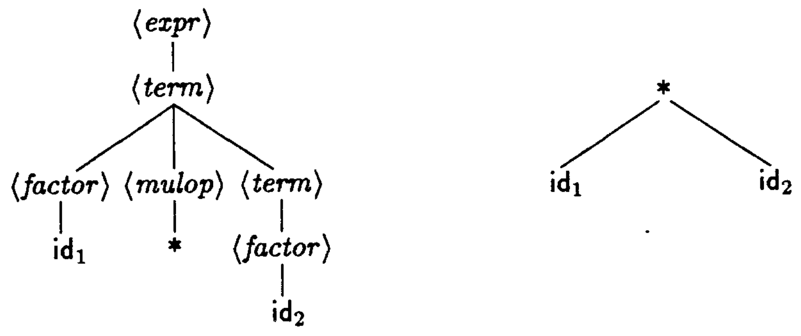


Figure 4: Parse vs. Abstract Tree Representations

Few of these issues address the need for a coherent conceptual model for documents in the language.

Names for components

Related to structural decomposition is the choice of terminology by which document components can be named. Simple structure editors (especially when designed to support multiple languages) may not name components at all, requiring the user to think in terms of representation-oriented commands (for example **Left**, **Right**, **In**, **Out**, and **Delete-Subtree**). A single-language editor can provide more natural language-oriented commands (e.g., **Next-Function**, **Previous-Declaration**, or **Delete-Statement**), but this approach doesn't generalize across languages. In some cases appropriate names overlap and depend on context; for example a structural component internally called "variable" in a programming language representation might be conceptually both a "variable" and an "expression" in some contexts but only a "variable" in others.

A separate model for users

Pan decouples conceptual models of document structure from internal representations. The language description mechanism provides a loose framework in which the author of each language description is expected to design a model. This framework is based on two assumptions about how people understand syntactic structure.

The first assumption is that people generally think not in terms of trees but in terms of familiar structural components, the ones that might be described in an informal language description: procedures, declarations, statements, and the like. People understand nesting but only as a secondary concept; for example, even though natural language permits arbitrary nesting, people find sentences with even three nested levels difficult to understand.

The second assumption is that people think of structural components in the specific terms and concepts of the language being edited, "statements" for example, and not in generic structural terms like "subtree." This is an instance of the observation that users' perceptions of systems are much more sensitive to surface representations than to underlying structure[11].

[11]For example, the subjects in an evaluation of command languages for text editors failed to detect the

Operand classes

Pan's document model for each language is driven by a separate layer in the language description. Structural aspects of each language that will be revealed to the user are specified in terms of **operand classes**: arbitrary, possibly overlapping[12], collections of structural components in the abstract representation. Each operand class has a name by which it is known to the user. Our standard Modula-2 description includes classes named "Expression", "Statement", "Declaration", and "Procedure"; some correspond to more than one kind of internal structural component. The operand class "Syntax Error" (Section 8) overlaps any of those classes; for example, a single structure represent both a "Statement" and a "Syntax Error".

The operand level

Each *Pan* window has a current **operand level**, which the user selects from a menu of classes defined by the underlying language description. The operand level is a very weak input mode that modulates the operation of five generic commands: Next, Previous, Select, Extend Selection, and Delete. The operand level affects no other commands, and a user may choose not to use the level-sensitive versions at all. In particular, the operand level neither inhibits nor modulates text-oriented editing at any time.

When the operand level is "Statement", for example, the user may press the left mouse button anywhere and the "nearest" (heuristically speaking) structural component that meets the definition of operand class "Statement" will be selected (see Figure 5). In terms of implementation, the Next and Previous commands perform a tree walk, stopping only at subtrees of the appropriate operand class.

The menu of operand levels available while editing a Modula-2 document using the standard description include "Character", "Word", "Line", "Lexeme", "Expression", "Statement", "Declaration", "Procedure", "Syntax Error", "Unsatisfied Constraint". The first three levels are built-in, corresponding to textual concepts; we find that these aren't much used, since most users prefer familiar key bindings that are specific to words and lines. The last two are also predefined and are described in Section 8.

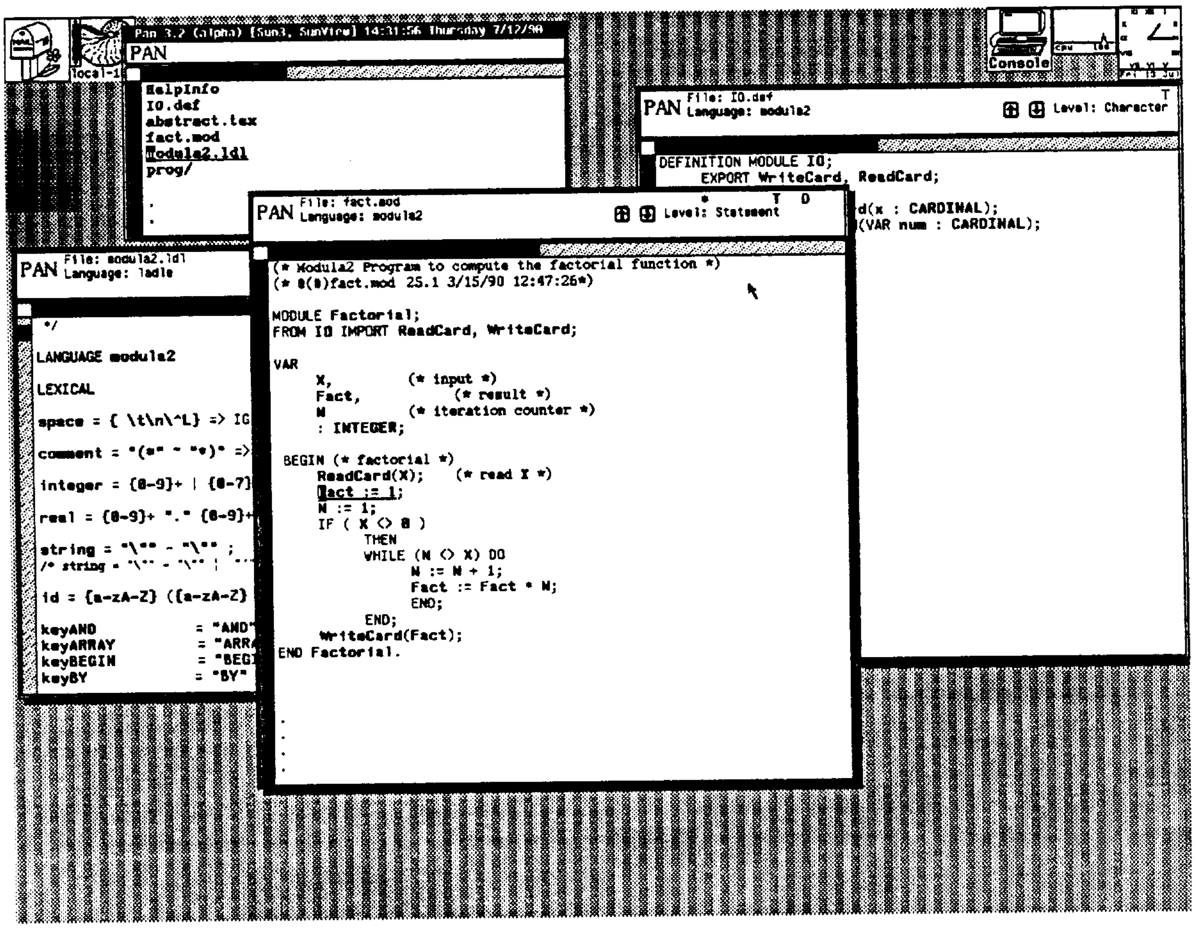
We have found the operand level a very useful mechanism, but we have yet to explore its full potential. We are now extending the implementation to permit more general definitions in terms of arbitrary predicates on structural components, drawing on language-oriented data in the database.

7 Inconsistency

Any situation where one kind of information is derived from another invites **inconsistency** between the two. The syntax-recognizing approach, where language-based information may

equivalence of two editors that differed only in details of their command languages [41].

[12]The possibility of overlapping separates *Pan*'s classes from operator-phyla [36]. This becomes important with more complex kinds of operand classes (see Section 8 for example). Furthermore, operand classes need not be defined for all possible internal structures.



```
Pan 3.2 (alpha) [Sun3, SunView] 14:31:56 Thursday 7/12/90
PAN
HelpInfo
IO.def
abstract.tex
fact.mod
Modula2.ldl
prog/
```

```
PAN File: modula2.ldl
    Language: ladle

 */

LANGUAGE modula2

LEXICAL

space = { \t\n\^L} => IG

comment = "(*" ~ "*)" =>

integer = {0-9}+ | {0-7}

real = {0-9}+ "." {0-9}+

string = "\"" - "\"" ;
/* string = "\"" - "\"" | "

id = {a-zA-Z} ({a-zA-Z}

keyAND              = "AND"
keyARRAY            = "ARRA
keyBEGIN            = "BEGI
keyBY               = "BY"
```

```
PAN File: fact.mod
    Language: modula2

(* Modula2 Program to compute the factorial function *)
(* @(#)fact.mod 25.1 3/15/90 12:47:26*)

MODULE Factorial;
FROM IO IMPORT ReadCard, WriteCard;

VAR
     X,         (* input *)
     Fact,            (* result *)
     N          (* iteration counter *)
     : INTEGER;

 BEGIN (* factorial *)
     ReadCard(X);    (* read X *)
     Fact := 1;
     N := 1;
     IF ( X <> 0 )
          THEN
          WHILE (N <> X) DO
                N := N + 1;
                Fact := Fact * N;
                END;
          END;
     WriteCard(Fact);
 END Factorial.
```

```
PAN File: IO.def
    Language: modula2

DEFINITION MODULE IO;
     EXPORT WriteCard, ReadCard;

d(x : CARDINAL);
(VAR num : CARDINAL);
```

Figure 5: Editing a Program using *Pan I*

be derived from text, is no exception. During text-oriented editing derived information maintained by the system will sometimes disagree with what the user sees, for example font shifts that reveal lexical categories may be out of date after a statement is transformed textually into a comment. Resolving inconsistency requires reanalysis, a potentially time consuming task. Yet inconsistency has the potential to detract from the goals of the system by confusing or misleading the user, by degrading performance, or by distracting the user with unpredictable behavior.

Inconsistency between text and derived information should not be confused with a related but different issue: the well-formedness of a document with respect to its underlying language. Unfortunately, many language-based editors that permit text-oriented editing are able to resolve inconsistency only for well-formed documents. In practice, a user who chooses to modify part of a document textually may do no other kind of editing until that part of the document is well-formed again. *Pan* removes this restriction, using mechanisms for document analysis that *never fail*. It is possible for the text of a document in *Pan* to be well-formed but inconsistent with respect to derived information; it is also possible for it to be ill-formed but consistent, in which case the derived data includes diagnoses. Section 8 treats well-formedness in more detail, and it will not be discussed further here.

Inaccurate information

Of the potential problems with inconsistency, the most serious would be to mislead the user with obsolete derived data.

Pan's first defense against this prospect is the policy that structural interaction may take place *only* when text and structure are consistent. This policy is implemented by **automatic reanalysis**: every command that requires derived information first ensures consistency, triggering reanalysis if needed, before proceeding. Since analysis always succeeds, this policy cannot restrict the user, although it may cause delay.

This policy is compromised when presentation enhancements (font shifts, highlighting, and the like) depend on derived information but persist visually during periods of inconsistency. A conservative policy would prohibit all such enhancements in the presence of inconsistency, since any or all of them could be incorrect. In practice such a policy would lead to unpleasant visual effects after the first text-oriented change following an analysis, as the system removed all visual enhancements. Even more important, it would result in the loss of information that most of the time is almost correct.

Pan's compromise rests on the assumption that the user can judge the implications of inconsistency[13] as long as it is visible, hence the policy to make visible both the presence and the extent of inconsistency. The presence of inconsistency is revealed by the panel flag "T"; this flag, otherwise displayed in boldface, appears gray during periods of inconsistency. The extent of any current inconsistency is revealed partially with a special font distinct from those used for lexical categories in the underlying language; this special font is used only for newly entered text that has no structural properties. To reveal completely the extent of inconsistency would require that recently deleted text be somehow visible, but the potential

[13]After all, the user is responsible for the changes that introduced it.

benefit of that extreme approach does not seem to justify the cost in implementation and display confusion.

Analysis strategies

How often and under what circumstances to attempt reanalysis is the most challenging aspect of inconsistency. A good policy involves interactions among user behavior and expectations, analysis strategies and implementations, and the speed with which analysis can be performed. Two of *Pan*'s pervasive implementation strategies help minimize analysis time.

The first implementation strategy is complete reliance on **incremental analysis**. *Pan* only recomputes information that might change, based on a record of what text has changed since prior analysis. For many kinds of textual changes, especially simple localized changes, analysis time is roughly proportional to the extent of the changes. Unfortunately, the non-local nature of document analysis means that any change creates the potential for extensive reanalysis. We expect, and have so far confirmed, that *Pan* users will develop an intuition about the kinds of textual changes that have extensive implications and will not perceive variations in analysis time as caprice on the part of the system.

The second implementation strategy includes several techniques and heuristics to help "reuse" language-based information during analysis. This strategy helps prevent information loss (and analysis time when the information is computable at all) caused by the transient appearance of minor language violations [8].

Analysis policies

Even with careful implementation, analysis time can be noticeable with present-day workstations. A careful approach to setting policy for reanalysis is important for an optimal user-interface.

An overly ambitious policy, attempting reanalysis after every character insertion or deletion, encounters two serious problems. First, analysis at this granularity would find documents nearly always ill-formed. There is little to be gained by insisting that the system perform useful analysis on documents that are in intentionally meaningless states. Second, this drain of resources degrades performance in ways beyond the user's control. Third, over-eager update of the display is visually distracting

There are two intermediate policies. The system might *guess* when the document is reasonably well-formed. This strategy has some of the same drawbacks as the ambitious strategy, for example even more potential for unpredictable and uncontrollable behavior. A second intermediate approach would restrict text editing to a bounded context (often called a "focus"), performing analysis when the user attempts to leave the context in some way. This approach is incompatible with *Pan*'s general insistence on unrestricted text editing with no overhead on shifts of perspective.

Pan's current policy relies upon the assumption that the user understands the general state of the document and can judge the tradeoffs involved. Incremental analysis is only performed when requested by the user, either *implicitly* by invoking an operation that triggers automatic reanalysis (described earlier in this section) or *explicitly* by invoking the command Analyze-Changes. Nothing prevents a *Pan* user from typing an entire document without once invoking analysis. The *Pan* approach is to encourage frequent analysis by making it cost-effective to the user: both efficient and useful.

8 Ill-Formed Documents

Inherent in the syntax-recognizing approach is a certainty that documents being modified are more often than not **ill-formed**: at variance with an underlying language definition[14].

Many language-based editors that permit text-oriented editing inherit this bias. Unable to analyze ill-formed documents, these editors insist that the user correct any newly introduced "errors" before proceeding. Often justified as a service, because it limits the extent and duration of "errors," this treatment has unpleasant side effects.

1. It "narrows" options available to the user, who may prefer to delay trivial repairs while dealing with more important issues. An "error" may often be part of an elaborate textual transformation.

2. It implies that derived information is only available and accurate when documents are well-formed, again constraining the user.

3. It implies that the user has done something wrong, when in fact the system is simply unable to understand what the user is doing [44].

Pan's approach to the treatment of ill-formed documents pervades the system. It is an entirely normal state in *Pan* for documents to be ill-formed, and every attempt is made to provide all services in the presence of any such **variances** with respect to the constraints imposed by the language description. In fact, information about variances is an important and useful kind of derived information, available when the user wants it.

Pan's approach decouples ill-formedness from inconsistency between the textual and language-based aspects of a document, discussed in Section 7. An inconsistent document may or may not be well-formed (the system can't know in this case), and a consistent document likewise may or may not be well-formed.

Variance

A central part of each *Pan* language description defines what it means for a document in the language to be well-formed. *Pan* uses that information to diagnose any variances between document and language definition. To the user, such variances are "errors," this being the familiar term, but there is nothing exceptional about them internally.

Pan's design includes two special mechanisms for handling variances, reflecting the two aspects in which the author of a language description defines well-formedness: a context-free

[14]Ill-formed documents are usually said to contain "errors," a pejorative term reflecting the limitations of many analysis methods.

grammar and contextual constraints. These layers are reflected in turn by separate analysis techniques, reflecting the traditional division of analysis in compilers into "syntactic" and "static-semantic" analysis.

Pan builds internal tree representations defined by information supplied with each language's context-free grammar. *Pan* implicitly extends the definition of each language to include special document components presented to the user as "syntactic errors." When a variance of syntax is encountered during incremental syntactic analysis, a new structural component is created to represent that variance. The new component is named after the specific kind of variance found, such as "malformed statement". A component representing a variance may include well-formed subcomponents produced during prior analysis. For example, a "syntactic error" named "malformed block" might contains well-formed "statements," still accessible to the user as "statements." As much derived information as possible is retained along with these subcomponents for later analysis. This is an important implementation strategy for bounding the effects of minor or ephemeral syntactic variances.

Pan's semantic analyzer attempts to prove that every description-defined constraint on each document component is satisfied[15]; failures are represented using constraint variances, presented to the user as "unsatisfied constraints." For example, many programming languages require that all variables be declared. In a *Pan* language description, this requirement is enforced by placing a constraint on each document component where a use can appear. If an undeclared variable appears in the document, a constraint variance is tagged onto the relevant component as one of many properties that may be recorded there.

Well-formed documents in *Pan* differ from ill-formed documents only by the absence of variances. In particular, both text- and language-based operations may proceed in the presence of variances. *Pan* offers the user three different ways to communicate *about* variances: announcement, navigation, and highlighting. As with most of *Pan*'s interface, these mechanisms are generally optional and under user control.

Announcement

At the conclusion of each incremental analysis, an ephemeral and unobtrusive message notes the number of variances present. In addition, the panel flag "!" appears when the internal tree representation contains any "syntactic error" components and the flag "#" appears when the document contains any "unsatisfied constraints." When neither flag is present (assuming consistency), the document is by definition well-formed.

Highlighting

A user may request that the text associated with all "syntactic errors" be rendered continuously with characters in a special color (red at present). This form of highlighting, adjusted after each analysis, is designed to maintain document legibility but to give rapid feedback about the location of variances. Experienced programmers often identify specific problems at a glance, once their attention is drawn to them.

[15]A *Pan* language description may include useful additional constraints that do not derive from the language definition, but rather from local and possibly personal conventions for the *use* of each language.

An independent option requests that document components with any "unsatisfied constraints" have their text highlighted. Since the location and extent of these is is generally orthogonal to "syntactic error" components, this highlighting is rendered as a pale yellow background shading that is independent of character color. The effect is similar to marking with a "highlighter" pen.

Navigation

The user may investigate specific variances and their diagnoses by navigation. After selecting the built-in operand level "**Syntax Error**", the user presses the left mouse button anywhere in the display. As with other operand levels, this action sets the edit cursor at the nearest component of the appropriate class, selects (underlines) associated text, and sets the text cursor at its beginning. Unlike ordinary operand levels, this command also presents in the panel a diagnosis, "malformed statement" for example[16]. While the operand level remains at "**Syntax Error**", commands **Next** and **Previous** traverse "syntactic errors," announcing the diagnosis at each.

The built-in operand level "**Unsatisfied Constraint**" works analogously, even though it is defined in terms of information in *Pan*'s database instead of in terms of syntactic structure. This operand level is the prototype of a new class of operand levels, defined according to arbitrary database predicates.

Open issues

Like most language-based systems, the distinction between syntactic analysis (incremental parsing) and contextual constraint checking is a fundamental organizing principle for *Pan*'s language descriptions and analysis mechanisms. It has become clear that the current user interface for variances exposes this distinction inappropriately.

To the naive user any variance is simply an "error." For such users we might redefine the user interface to treat them that way. But more experienced users seem to place variances into categories according to various criteria, for example severity, non-locality, or perhaps even level of surprise. The present categorization approximates some of these but doesn't present a user-oriented model of the kind described in Section 6. As we experiment with extra-lingual constraints, for example the imposition of locally-defined stylistic conventions, we find the current distinction even less appropriate.

A more general mechanism would permit the author of each language description to specify named categories for variances, in the same spirit that "operand classes" present a user model of language-based structure.

9 Mixed-mode Editing

Pan's fundamental approach to language-based editing is that it should broaden the user's

[16] A "malformed statement" is in both the "**Statement**" and "**Syntax Error**" operand levels. Recall that operand levels are defined as potentially *overlapping* classes of structural components.

options without affecting the user's text-oriented options any more than absolutely necessary. The user should be able edit textually any time, any place in the document presentation; it should be equally possible to edit in terms of the language any time, any place. This section introduces *Pan*'s simple language-oriented mechanisms, with special attention to their coexistence with text-oriented editing.

Shifting perspectives

A text-oriented operation followed by a structure-oriented operation implies a shift of perspective about the document, on the part of both user and system. Humans are adept at shifting perspective, and do so frequently to suit the cognitive task of the moment. For example, studies of experienced programmers reveal that both reading and authoring activities involve a variety of fine-grained cognitive tasks, with rapid switching among them [42, 54]. *Pan* supports these activities by always being ready to operate in either perspective.

In contrast, many language-based editors that provide mixed text- and structure-oriented operations require user activity, both mental and physical, to shift the system's perspective. This can distract, slow down, and possibly confuse the user. For example, some editors require that a structural component be specially selected for text-oriented editing; the textual content of the component then becomes the "focus" until the user wishes to resort to structure-oriented editing, at which time some other overt action may be required. Some editors provide a textual focus only in a separate window, preventing the user from editing textually in the natural visual context.

Mixing commands

Any *Pan* editing command, text- or structure-oriented, may be invoked without prerequisite. Two mechanisms make this work. The first, automatic reanalysis (introduced in Section 7) ensures that derived information is consistent with the text before performing any operations that require it. In many editors a similar transition can by interrupted by analysis that encounters ill-formed text. Section 8 describes how *Pan* avoids this problem. The second mechanism is the dual nature of *Pan*'s edit cursor.

A dual aspect cursor

Pan's edit cursor has two aspects. It always has a textual location, displayed as an inverted box (Figure 5, page 25). It may also have a location corresponding to some structural component. At present, the cursor's structural location is revealed by turning the component's textual presentation into the current text selection. Any operation that sets the structural cursor also positions the text cursor at the first character in the structure's textual presentation. Figure 5 shows a structural unit, a "“Statement”", having just been selected.

Any editing operation that requires a cursor location simply uses the appropriate aspect: text or structure. If the cursor has no structural aspect, then one is inferred from the text cursor's location by the same mechanism used when the user selects a structural component by pointing with the mouse. This design resolves the "point vs. extended cursor" problem [61] by providing both behaviors simultaneously.

People seem to have little difficulty with the implied ambiguity, presumably for the same reasons that they can shift perspectives themselves so effectively and can manage complex, possibly ambiguous domains [20]. Raskin reports that a dual-aspect text cursor, where the behavior depends on the next user action, solved a particular user-interface problem [51]; the important thing seems to be that the both aspects are visible and predictable.

The operand level revisited

One challenge for *Pan*'s user interface is to make structure-oriented commands available as conveniently as the familiar text-oriented ones, via bindings to keyboard sequences, mouse buttons, and menus. Other than a few generic structure-oriented commands (such as Analyze-Changes) most such commands are defined in terms of language-specific operand classes, as defined by the author of the underlying language description (see Section 6). For example, the command Select-Declaration should be available when editing languages for which the concept of "declaration"' has been defined, and not available for languages in which it is not. However, this approach does not generalize, since a confusing proliferation of commands would result from *Pan*'s support for many languages.

Pan's solution is a second level of indirection added to the basic command binding mechanisms, used by generic versions of five basic commands: Next, Previous, Select, Extend Selection, and Delete[17]. The user may specify the current "operand level" in each window by menu selection (or key binding), where the language-specific options derive from the language description (e.g. "Expression", "Statement", "Declaration", and "Procedure"). For example, when the left mouse button is bound to the generic version of Select and when the current operand level is "Statement", a left mouse button press invokes a structural command that selects the statement nearest the mouse location.

The operand level mechanism for command dispatch is a very weak input mode. It modulates the effect of the five basic commands, but has no other effects and in particular implies no restrictions on user actions. For example, the user may edit textually independent of the current operand level.

We sometimes find it convenient to violate the model presented above by strengthening slightly the effect of the operand level. With this adjustment, any change to the operand level causes an implicit Select operation at the new level. This has the visible effect of moving the edit cursor in some cases. It isn't clear yet whether the convenience justifies the possible confusion.

Simple editing

The prototype implementation supports no user commands that modify internal document structure directly. A Delete command, invoked with a structural selection, achieves the same effect by removing the text associated with the selected component. The structure corresponding to the deleted component persists until the next reanalysis, but it is invisible

[17]The generic versions of these commands are called Next-@Level etc. and are bindable just like any other *Pan* command.

to the user because automatic reanalysis will delete it before any commands can use it. **Cut** simply places text in the clipboard.

The command **Paste** simply inserts text from the clipboard. If the context is appropriate, subsequent incremental analysis derives the equivalent structural information quickly.

This implementation costs a small amount of analysis time by discarding derived information when the user moves structural components. On the other hand, it guarantees the integrity and well-formedness of the document's internal representation, since the language definition is already built into *Pan*'s parser.

Complex mechanisms for direct structural editing can be a source of confusion to the user, since those editing operations may "fail," something *Pan* commands rarely do. Worse, they may fail for the kinds of reasons we attempt to hide from the user. For example, it seems reasonable to copy the list of identifiers appearing in the formal parameter list of a procedure definition and paste it into a call to that procedure. Although the two lists of identifiers might appear identically and be closely related conceptually, there may be sound implementation reasons for different internal representations in the two contexts. We prefer to avoid strategies that involve guessing the user's intent.

When a structurally inspired **Cut** and **Paste** sequence in *Pan* violates the underlying language definition, the operations succeed and the problem is diagnosed by precisely the same mechanism that handles other language violations (discussed in Section 8).

The cost at present of this text-based implementation is the loss of any non-derivable annotations on document components during **Cut** and **Paste** sequences. We have developed, but not yet added, a strategy that avoids this information loss and provides functionality that is equivalent to direct structural operations. The successor to *Pan I* addresses the problem in more fundamental ways.

Other language-based operations

The ultimate advantage of language-oriented editing lies in a rich and open-ended collection of "services" that draw upon a rich repository of information to assist users in commonly performed tasks. The best developed collection of these services in *Pan* at present deals with the location and diagnosis of variances, as described in Section 8. This section describes a few other examples that either have been prototyped or will be soon.

One of the few forms of query supported by ordinary text editors is textual search. Searching in *Pan* can draw on *any* information in the repository. For example, one search command locates in a document the declaration and all uses of a programming language variable, which the user identifies by pointing. The results of this and other language-based queries are made available by an interface similar to the one used for variances (described in Section 8). Text associated with all components of the current query result appears with different color characters, blue at present. The user may choose the operand level "Query Result" and navigate through the components. A closely related command allows the user to point at a variable and move the cursor to the corresponding declaration; this is only one example of a command that follows hypertext-like links defined by the underlying language.

Like all full-functioned text editors, *Pan* also supports textual replacement based on regular expression matching. However, one often intends that the replacement depend on the language structure, not on the textual structure, even when the two are similar. For example, word replacement in natural language documents is tricky using regular expressions. One wants to avoid replacing occurrences embedded in other words, so a simple specification of the search string does not suffice; at the same time, it is difficult to describe all characters (including beginning and end of line) that might mark word boundaries. The answer involves specific commands that use *Pan*'s derived information to replace only whole components. Even more powerful versions replace only those occurrences of a name that are related according to the rules of the language, for example when renaming a variable in a program.

Other open issues

At present, during text-oriented editing, both a cursor and selection are used, but during structure-oriented editing only a cursor is available. Some prototyping experiments have suggested possible uses for a structural selection, but the addition threatens the user interface with extra complexity.

Setting the structure cursor implies both a text cursor location and a text selection. In one sense the linkage is limiting, but it is also simplifying. We rely upon to get the effect of structure-oriented editing in the current implementation (see above). This is an example of wrapping general mechanisms in specific services that are easy to understand.

One of the reasons for both decisions, linking the structure cursor to the text cursor/selection and the omission of a tree selection, is a shortage of presentation enhancements on a monochrome monitor that can display all of them intelligibly at the same time. It can be done with color. Even so, we suspect that these four separate mechanisms should never be revealed to the user as four independent interface concepts; rather, they should be driven by various specialized services that use them in concert in predictable and useful ways.

10 Future Directions

In addition to the general language-based services already described, we have built a number of special-purpose services that add to *Pan*'s general utility. These include a browsing interface to the file system in the form of a "directory editor," an integral help system for both naive users and developers that provides access to documentation and system source code, and a hypertext-like browser for UNIX[18] online manual pages. We expect to see more of these practical additions.

We are currently working with prototypes for general language-based queries, a syntax-directed style of editing, and other new language-based services. Architectural revisions in progress include support for multiple views (or projections) per buffer, graphical views for tree- or graph-structured data, and the application of language-based techniques to provide

[18]UNIX is a trademark of AT&T Bell Laboratories.

visual access to internal data structures. New language description techniques are also under development.

Ongoing research projects use the leverage gained from the *Pan* system. Projects near completion include a study of program presentation that uses derived information to provide elision, alternate textual representations, and mappings to hardcopy using typesetting [11]; the development of advanced document analysis techniques to specify and control user-centered program viewing [63]; and integration with a persistent database.

An important group of issues concerns extensions to *Pan*'s language description mechanisms. First, we wish to add new descriptive layers for various purposes. For example, we intend to build browsers suitable for program call graphs and inheritance hierarchies (lattices) for languages where this is meaningful. A new description layer might define how the information is to be derived from the database and supplied to a graphical viewer. Although the present language-description facilities provide simple support for description layers, the problem of layering language descriptions requires further work.

Second, while the current implementation supports sharing among documents written in a common base language, it should be extended to deal with both documents written in multiple languages and with sharing among documents written in different languages.

Finally, we hope to move beyond the limitations in our language-description techniques that restrict *Pan* to "classical" computer languages—languages having a well-defined (and simple) syntax together with some set of well-formedness constraints. This would require adaptation of our techniques to other classes of languages: those with non-standard context-sensitive syntactic aspects, graphical languages, and even natural languages.

Continuing work at UC Berkeley, part of the *Ensemble* project, is generalizing *Pan*'s approach in three ways:

1. Much richer mappings among document structure, presentations, and specification of appearance, building heavily on the experience gained from the VORTEX document system [16, 17].

2. The extension of editing and viewing to a wide range of media—text, graphics, sound, and video.

3. Integrated support for compound documents, where different languages and document types may be composed.

11 Acknowledgements

Pan is the work of many individuals besides the authors. Christina Black, Jacob Butcher, Bruce Forstall, Mark Hastings, and Darrin Lane have all made substantial contributions. The encouragement, suggestions, and support of Bill Scherlis have played a major role as well.

References

[1] ACM. *Proc. of the ACM SIGPLAN SIGOA Symposium on Text Manipulation* (Portland, Oregon, June 8–10, 1981). Appeared as SIGPLAN Notices, 16(6), June 1981.

[2] APPLE COMPUTERS, INC. *MacWrite Manual.* Cupertino, California, 1984.

[3] BAECKER, R. M., AND MARCUS, A. *Human Factors and Typography for More Readable Programs.* ACM Press, New York, New York, 1990.

[4] BAHLKE, R., AND SNELTING, G. The PSG system: From formal language definitions to interactive programming environments. *ACM Trans. on Programming Languages and Systems 8*, 4 (1986), 547–576.

[5] BAHLKE, R., AND SNELTING, G. We don't need structure editors — we need smart text editors. *Presented at the CHI '90 Workshop on Structure Editors* (Apr. 1990).

[6] BALLANCE, R. A. *Syntactic and Semantic Checking in Language-Based Editing Systems.* Ph.D. dissertation, Computer Science Division (EECS), University of California, Berkeley, California, 94720, Dec. 1989. Available as Technical Report No. UCB/CSD 89/548.

[7] BALLANCE, R. A., BUTCHER, J., AND GRAHAM, S. L. Grammatical abstraction and incremental syntax analysis in a language-based editor. In *Proc. of the SIGPLAN '88 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, June 22–24, 1988), ACM, pp. 185–198. Appeared as Sigplan Notices, 23(7), July 1988.

[8] BALLANCE, R. A., GRAHAM, S. L., AND VAN DE VANTER, M. L. The Pan language-based editing system for integrated development environments. In *ACM SIGSOFT '90: Fourth Symposium on Software Development Environments* (1990). To appear.

[9] BALLANCE, R. A., AND VAN DE VANTER, M. L. Pan I: An introduction for users. Technical Report No. UCB/CSD 88/410, Computer Science Division (EECS), University of California, Berkeley, California, 94720, Mar. 1988.

[10] BALLANCE, R. A., VAN DE VANTER, M. L., AND GRAHAM, S. L. The architecture of Pan I. Technical Report No. UCB/CSD 88/409, Computer Science Division (EECS), University of California, Berkeley, California, 94720, Mar. 1988.

[11] BLACK, C. L. PPP: The Pan program presenter. Technical Report No. UCB/CSD 90/589, Computer Science Division (EECS), University of California, Berkeley, California, 94720, Sept. 1990.

[12] BORRAS, P., CLÉMENT, D., DESPEYROUX, T., INCERPI, J., KAHN, G., LANG, B., AND PASCUAL, V. CENTAUR: the system. In *ACM SIGSOFT '88: Third Symposium on Software Development Environments* (1988), P. Henderson, Ed., pp. 14–24.

[13] Budinsky, F. J., Holt, R. C., and Zaky, S. G. SRE—a syntax-recognizing editor. *Software—Practice & Experience 15*, 5 (May 1985), 489–497.

[14] Butcher, J. Ladle. Master's thesis, Computer Science Division (EECS), University of California, Berkeley, California, 94720, Nov. 1989. Available as Technical Report No. UCB/CSD 89/519.

[15] Chandhok, R., Miller, P., Pane, J., and Meter, G. Structure editing: Evolution towards appropriate use. *Presented at the CHI '90 Workshop on Structure Editors* (Apr. 1990).

[16] Chen, P., Coker, J., Harrison, M. A., McCarrell, J. W., and Procter, S. The VORTEX document preparation environment. In *Proc. of the 2nd European Conference on TEX for Scientific Documentation* (Strasbourg, France, June 19–21 1986), vol. 236 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 32–24.

[17] Chen, P., and Harrison, M. A. Multiple representation document development. *IEEE Computer 21*, 1 (Jan. 1988), 15–31.

[18] Conradi, R., Didriksen, T. M., and Wanvik, D., Eds. *Advanced Programming Environments* (Berlin, Heidelberg, New York, 1986), no. 244 in Lecture Notes in Computer Science, Springer-Verlag.

[19] Constable, R. L., Allen, S. F., Bromley, H. M., Cleaveland, W. R., Cremer, J. F., Harper, R. W., Howe, D. J., Knoblock, T. B., Mendler, N. P., Panangaden, P., Sasaki, J. T., and Smith, S. F. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[20] diSessa, A. A. Models of computation. In *User Centered System Design: New Perspectives on Human-Computer Interaction*, D. A. Norman and S. W. Draper, Eds. Lawrence Erlbaum Associates, Hillsdale, NJ, 1986, pp. 210–218.

[21] diSessa, A. A. Notes on the future of programming: Breaking the utility barrier. In *User Centered System Design: New Perspectives on Human-Computer Interaction*, D. A. Norman and S. W. Draper, Eds. Lawrence Erlbaum Associates, Hillsdale, NJ, 1986, pp. 125–152.

[22] Donzeau-Gouge, V., Huet, G., Kahn, G., and Lang, B. Programming environments based on structure editors: the MENTOR experience. Research Report No. 26, INRIA, July 1980.

[23] Donzeau-Gouge, V., Huet, G., Kahn, G., and Lang, B. Programming environments based on structured editors: the MENTOR experience. In *Interactive Programming Environments*, D. R. Barstow, H. E. Shrobe, and E. Sandewall, Eds. McGraw-Hill, New York, NY, 1984.

[24] Donzeau-Gouge, V., Huet, G., Kahn, G., Lang, B., and Levy, J. J. A structure oriented program editor: a first step towards computer assisted programming. In *International Computing Symposium 1975* (1975), E. Gelenbe and D. Poiter, Eds., North-Holland Publishing Company, pp. 113–120.

[25] Douglas, S. A., and Moran, T. P. Learning text editor semantics by analogy. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Boston, MA, Dec. 1983), ACM, pp. 207–211.

[26] Fraser, C. W., and Lopez, A. A. Editing data structures. *ACM Transactions on Programming Languages and Systems 3*, 2 (Apr. 1981), 115–125.

[27] Gansner, E. R., Horgan, J. R., Moore, D. J., Surko, P. T., Swartwout, D. E., and Reppy, J. H. SYNED—a language-based editor for an interactive programming environment. In *IEEE Spring Compcon '83* (1983).

[28] Garlan, D. Flexible unparsing in a structure editing environment. Tech. Rep. CMU-CS-85-129, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, Apr. 1985.

[29] Goldberg, A. Programmer as reader. *IEEE Software 4*, 5 (Sept. 1987), 62–70.

[30] Halme, H., and Heinnan, J. GNU emacs as a dynamically extensible programming environment. *Software—Practice & Experience 18*, 10 (Oct. 1988), 999–1009.

[31] Hansen, W. J. *Creation of Hierarchic Text with a Computer Display.* Ph.D. dissertation, Stanford University, June 1971.

[32] Hansen, W. J. User engineering principles for interactive systems. In *Interactive Programming Environments*, D. R. Barstow, H. E. Shrobe, and E. Sandewall, Eds. McGraw-Hill, 1984, ch. 8, pp. 217–231.

[33] Hilfinger, P. N., and Colella, P. Fidil: A language for scientific programming. In *Symbolic Computation: Applications to Scientific Computing*, R. Grossman, Ed. SIAM, 1989, pp. 97–138.

[34] Holt, R. W., Boehm-Davis, D. A., and Schultz, A. C. Mental representations of programs for student and professional programmers. In *Empirical Studies of Programmers: Second Workshop*, G. M. Olson, S. Sheppard, and E. Soloway, Eds. Ablex Publishing, Norwood, NJ, 1987, p. 33.

[35] Horton, M. R. *Design of a multi-language editor with static error detection capabilities.* Ph.D. dissertation, Computer Science Division (EECS), University of California, Berkeley, California, 94720, 1981.

[36] Kahn, G., Lang, B., Mélèse, B., and Morcos, E. Metal: A formalism to specify formalisms. *Science of Computer Programming 3* (1983), 152–188.

[37] KAISER, G. E., AND KANT, E. Incremental parsing without a parser. *Journal of Systems and Software 5*, 2 (May 1985), 121–144.

[38] KIRSLIS, P. A. C. *The SAGA editor: A Language-Oriented Editor Based on an Incremental LR(1) Parser.* Ph.D. dissertation, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1986.

[39] LAMPSON, B. W. *Bravo users manual.* Palo Alto, 1978.

[40] LANG, B. On the usefulness of syntax directed editors. In Conradi et al. [18], pp. 47–51.

[41] LEDGARD, H., SINGER, A., AND WHITESIDE, J. *Directions in Human Factors for Interactive Systems.* Springer-Verlag, Berlin, Heidelberg, New York, 1981.

[42] LETOVSKY, S. Cognitive processes in program comprehension. In *Empirical Studies of Programmers*, E. Soloway and S. Iyengar, Eds. Ablex Publishing, Norwood, NJ, 1986, pp. 58–79.

[43] LETOVSKY, S., AND SOLOWAY, E. Delocalized plans and program comprehension. *IEEE Software 3*, 3 (May 1986), 41–49.

[44] LEWIS, C., AND NORMAN, D. A. Designing for error. In *User Centered System Design: New Perspectives on Human-Computer Interaction*, D. A. Norman and S. W. Draper, Eds. Lawrence Erlbaum Associates, Hillsdale, NJ, 1986, pp. 411–432.

[45] MEDINA-MORA, R., AND FEILER, P. H. An incremental programming environment. *IEEE Trans. on Software Engineering SE-7*, 5 (1981), 472–481.

[46] MEYROWITZ, N., AND VAN DAM, A. Interactive editing systems: parts I and II. *ACM Computing Surveys 14*, 3 (1982), 321–416.

[47] NEAL, L. R. Cognition-sensitive design and user modelling for syntax-directed editors. In *CHI+GI* (1987), pp. 99–102.

[48] NOTKIN, D. *Interactive Structure-Oriented Computing.* Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, Pennsylvania, 15213, Feb. 1984.

[49] NOTKIN, D. The GANDALF project. *Journal of Systems and Software 5*, 2 (May 1985), 91–106.

[50] OMAN, P., AND COOK, C. R. Typographic style is more than cosmetic. *Communications of the ACM 33*, 5 (May 1990), 506–520.

[51] RASKIN, J. Systemic implications of leap and an improved two-part cursor: a case study. In *Proceedings SIGCHI Conference on Human Factors in Computing Systems* (Austin, TX, May 1989), pp. 167–170.

[52] Reps, T., and Teitelbaum, T. The synthesizer generator. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (1984), P. Henderson, Ed., pp. 42–48.

[53] Rich, C., and Waters, R. C. The Programmers Apprentice: A research overview. *IEEE Computer 21*, 11 (Nov. 1988), 10–25.

[54] Rist, R. S. Plans in programming: Definition, demonstration, and development. In *Empirical Studies of Programmers*, E. Soloway and S. Iyengar, Eds. Ablex Publishing, Norwood, NJ, 1986, pp. 28–45.

[55] Robert L. Mack, C. H. L., and Carroll, J. M. Learning to use word processors: Problems and prospects. *ACM Transactions on Office Information Systems 1*, 3 (July 1983), 254–271.

[56] Rose, C. *Inside Macintosh.* Addison-Wesley, Reading, Massachusetts, 1985.

[57] Soloway, E., and Ehrlich, K. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering SE-10*, 5 (Sept. 1984), 595–609.

[58] Stallman, R. M. EMACS: the extensible, customizable, self-documenting display editor. In *Proc. of the ACM SIGPLAN SIGOA Symposium on Text Manipulation* [1], pp. 147–156. Appeared as SIGPLAN Notices, 16(6), June 1981.

[59] Strömfors, O. Editing large programs using a structure-oriented text editor. In Conradi et al. [18], pp. 39–46.

[60] Teitelbaum, T., and Reps, T. The Cornell Program Synthesizer: a syntax-directed programming environment. *Communications of the ACM 24*, 9 (1981), 563–573.

[61] Teitelbaum, T., Reps, T., and Horwitz, S. The why and wherefore of the Cornell Program Synthesizer. *SIGPLAN 16*, 6 (June 1981).

[62] Teitelman, W. A tour through cedar. *IEEE Trans. on Software Engineering SE-11*, 3 (Mar. 1985).

[63] Van De Vanter, M. L. User-centered program viewing. Research Proposal, Computer Science Division—EECS, University of California, Berkeley, California, 94720, Nov. 1987.

[64] Van De Vanter, M. L. Error management and debugging in Pan I. Technical Report No. UCB/CSD 89/554, Computer Science Division (EECS), University of California, Berkeley, California, 94720, Dec. 1989.

[65] Waters, R. C. Program editors should not abandon text oriented commands. *SIGPLAN Notices 17*, 7 (1982), 39–46.

[66] Winograd, T. Beyond programming languages. *Communications of the ACM 22*, 7 (July 1979), 391–401.

[67] WOOD, S. R. Z—the 95% program editor. In *Proc. of the ACM SIGPLAN SIGOA Symposium on Text Manipulation* [1], pp. 1–7. Appeared as SIGPLAN Notices, 16(6), June 1981.