Submitted to 1994 Symposium on Foundations of Software Engineering

A Document Architecture for Integrated Software Development*

Brian M. Dennis Roy Goldman Susan L. Graham Michael A. Harrison
William Maddox Vance Maverick† Ethan V. Munson Tim A. Wagner

Computer Science Division–EECS
University of California, Berkeley

Abstract

The software development process generates a wide variety of artifacts. Supporting the management and editing of these artifacts in a single, tightly-integrated development environment is a widely shared goal.

To support a software object means not only managing its evolution, representation and storage, but also its display and user interaction. To integrate support for the representation, display, and editing of all the objects in the development process, we regard objects as *documents* — structured compositions of primitive types.

We present a document model for software development that accommodates textual documents in natural and formal languages, multimedia documents, and the results of programmatic analysis. We also present an architecture that supports presentation and editing of these documents, in such a way that the services for each primitive type are reusable in all classes of documents.

This document architecture is implemented in a working prototype system, Ensemble. Our experiences with Ensemble provide solutions to some of the key architectural problems in integrated environments.

1 Introduction

Large software development efforts produce a wide array of objects. While program source code is an important product, other objects are created in much greater quantity and variety. Software requirements, specifications, and design records, testing scripts and sample data, documentation and training materials are all part of a project; but they are typically managed separately, by a variety of incompatible tools. A widely shared goal in software research is to support interoperable handling of these objects, and indeed the entire software development process.

The integration of a software environment involves at least three levels of the system: object model, presentation, and control. The system must maintain and respect the dependencies and relationships among the software objects it manages: for example, between source code and documentation, or between source code and the results of analysis on the code. For a developer to learn something from these software objects, they must be displayed, and most of them must be user-editable. In an environment supporting such a diverse collection of objects, presentation and control can grow very complex. The growing use of multimedia elements in training materials and in the early stages of software design suggests that environments must move toward support for multiple media. Thus, from our point of view, the integration of a software environment means that its services for presentation and control are widely shared by the various object classes, and also that it is easy to extend the system to handle new kinds of objects.

*This research has been sponsored in part by the Advanced Research Projects Agency (ARPA) under Grant MDA972-92-J-1028, by the NSF under grant CDA-8722788, and an IBM Fellowship. The content of this paper does not necessarily reflect the position or the policy of the Government or IBM.

†For correspondence: maverick@cs.berkeley.edu or Vance Maverick, Computer Science Division, 571 Evans Hall # 1776, University of California, Berkeley CA 94720-1776

One approach to building an integrated environment is to compose it from *tools*, as in the Field project [26]. Each object class is handled by a separate tool, which manages display, editing, and storage. Integrating the system becomes a question of defining the interaction of these tools. While tool-based systems can be effective, the separation of tools hinders integration. If a new object class is added to the system, a new tool must be built, even if the new class is similar in structure or content to previous classes. If code from other tools can be reused, this reuse is not systematic, and the code will be duplicated rather than shared. Further, the simple model of communication between tools makes fine-grained sharing of complex data difficult.

More tightly integrated systems require a common object model. For instance, the PLEIADES object management system [31], part of the Arcadia environment [15], defines a complex and flexible model of objects as unrestricted abstract data types (ADTs). The PLEIADES paper does not address integration at the presentation and control levels. However, the Chiron user interface management system [16], which is also part of Arcadia, manages presentation and control with *artists*, each of which handles a single abstract data type. Artists, in this model, are just as separate as tools in Field; there is no mechanism by which artists for related abstract data types may share their implementation. Also, they are written in a general-purpose programming language (a superset of Ada), so adding support for a new ADT requires a developer's knowledge of the system.

One system which has achieved partial integration at all three levels, at the cost of a very narrow object model, is GNU Emacs [28]. In Emacs, all objects are modeled as text buffers. Accordingly, presentation can be handled by a single text display system. Control is managed by an extension language, which loads a different extension package ("mode") depending on the type encoded in the filename of the object. Synchronization between objects, if any, is handled by hand-written extension code, without any direct system support. The most severe deficiency of Emacs, as part of an integrated software development system, is that a textual representation is unnatural for important object classes, such as parse trees and call graphs.

Our goal in this paper is to show that the editing of software objects can be integrated at all three levels, using an object model much richer than that of Emacs, but more constrained than that of PLEIADES. We model every software object as a *document*, a structured entity composed of elements drawn from a global set of primitive types. Every document can be displayed to the user, and if appropriate, may be edited. Taking this approach requires

- A document model which is broad enough to encode any of the artifacts of software development, but narrow enough to make specifying presentation and control straightforward. This document model must be able to express the results of syntactic and semantic analysis of programs.
- Presentation services powerful and flexible enough to display every class of document supported by the system using whatever media are appropriate.
- Control services which can be used to define user interaction styles that exploit the rich structural information of the documents.

In this paper, we present the document architecture of Ensemble, a prototype system under development at the University of California [11]. Sections 2–4 present the document, presentation, and control subsystems in turn. In Section 5, we show how these three systems cooperate to handle user actions and maintain consistency. Section 6 discusses the extension of our document model to non-textual media. Finally, we summarize the architectural lessons of Ensemble in Section 7.

2 The Document Model

A document in Ensemble is a tree of typed nodes whose leaves are drawn from a global set of primitive types. Every document belongs to a *document class*, which is defined by a grammar. For example, Figure 1 shows the tree of a simplified article document. Its leaves, the `Title` and `Paragraph` nodes, belong to the text primitive type. The grammar which defines the article class specifies that articles are composed of a title and one or more sections, where each section has one or more paragraphs.

We have chosen a hierarchical structure primarily because it is a natural basis for presentation and control specifications and because it is well-suited to the types of objects we wanted to address first: programs and

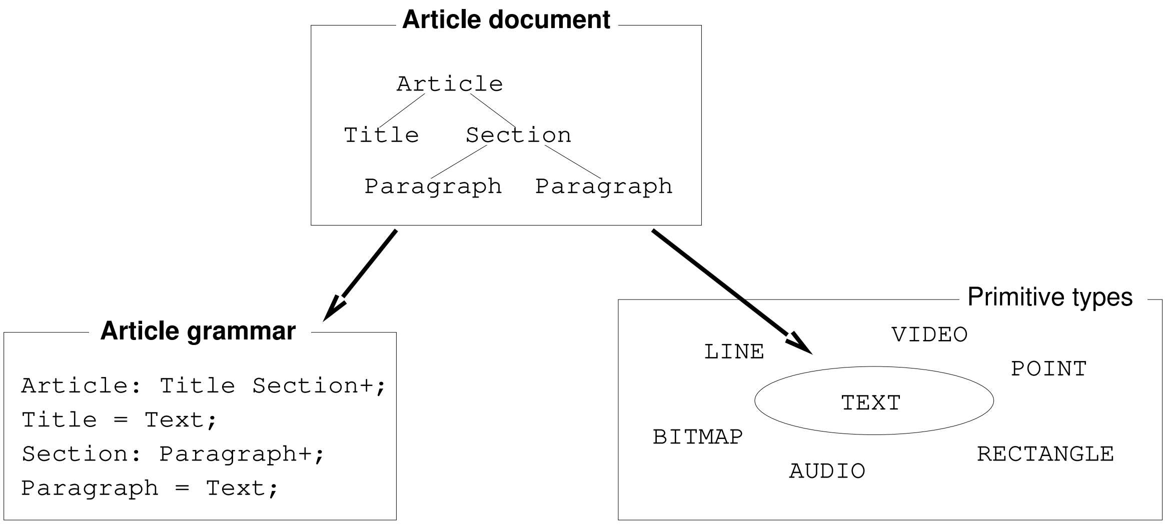

Figure 1: This document tree is an instance of Article class. It is a tree and its leaves are drawn from the global set of primitive types. Its structure is defined by a *grammar*.

system documentation. For documents with more general graph structures, such as call graphs and even many natural-language documents, a spanning tree of the graph is used as the primary structure; other edges are secondary links.

The concept of document class is central in Ensemble. The structure, presentation, and user interface of every document is primarily determined by its class. The class's grammar defines, for instances of the class, the set of legal structures and the primitive types of the leaves. Document classes for programming and other formal languages require additional specifications defining lexical conventions, static semantic restrictions, and program database schemas. Ensemble uses a document's class to assign it the correct formatting and user control configuration, as we will see in Sections 3 and 4.

Each document leaf is an instance of a primitive type, such as text, video, audio, or any of a number of graphic primitives. These types are introduced into Ensemble as part of the support for individual media. The relationship between the primitive types and Ensemble's media is discussed in Section 3.

Ensemble documents may be *compound*: that is, they may contain other documents. This is the means by which Ensemble provides interoperability between documents of different classes. For instance, a compound document can be used to place fragments of a program in the documentation describing the program. Support for compound documents is an area of active commercial development [21], but the commercial solutions lack the uniform document model found in Ensemble.

Ensemble has a novel document representation that allows analysis of formal documents to be "undone," just like any other editing operation that alters the document's structure or content. Under this representation, each low-level, atomic edit operation creates a potential version of the document. These versions form a history of the document's evolution and are accessible through an undo/redo interface. Furthermore, a transaction may be committed at any point, thereby identifying the current document content as the next major revision of the document. Because syntactic analysis uses these same low-level edit operations to reconcile the document's tree with its text stream, syntactic analysis can be undone. Details of this approach are given by Wagner and Graham [33].

2.1 Language Object Model

Ensemble's document model, like its low-level editing model, is *language-independent*. This facilitates the construction of a multi-lingual development environment. However, many system activities are constrained or parameterized by the language in which a document is written. Thus, the document class is more than just a naming mechanism: it also embodies a runtime representation that collects in a single place all the language-specific information about that document class.

Language objects are constructed from high-level descriptions, which are converted into `C++`, compiled, and dynamically loaded into the running image. Languages are organized into a class hierarchy that provides the usual benefits of an object-oriented framework: abstraction (the same basic interface and support is reusable across languages) and encapsulation (specializations peculiar to a particular language are hidden from the public interface, and are only loaded when a document of that language is in use). Inheritance also provides a unique mechanism for relieving the burden of language description when several document styles are related: common features may be placed in a parent class.

This approach provides a solution to the problem of reusing the low-level editing, version management, and analysis support across different languages, while still providing customization for each document class. In addition, Ensemble can provide different kinds of language services for different classes of documents. As an example, analysis of various kinds is a fundamental part of the software development process for programs.

Language-specific features, including high-level editing operations tailored to a specific document class, are also provided by dynamically loaded extension code. The extension language is described in Section 4.

2.2 Analysis of Formal Languages

Ensemble uses incremental analysis to determine information about formal language documents and to restore the consistency between their structure and content after some sequence of editing operations has occurred. While the general interface to the versioning and presentation system is identical across languages, each language must parameterize the analysis tools with specific code and/or data. Thus, the runtime portion of the document class for formal languages must include this information.

For formal language documents, the leaves of documents contain mostly text and the document structure corresponds to the abstract syntax tree.[1] Because these documents can be large and complex, achieving interactive performance and retaining document information requires that the analysis operations be incremental.

The analysis process per se is user-initiated for most languages. The system as a whole is *error-tolerant*: there is no requirement that any document or sub-portion of a document be valid at any time [32]. The user is free to modify partial documents and to leave them indefinitely in that state. The role of the system is to provide as much analysis and support for the existing portions as it can, and to degrade in a graceful fashion when faced with incomplete information.

All edit operations, including incremental analysis, produce a uniform *change report* to interested clients. This report details the relationship between the previous version and the current version of the document; it also preserves nodes (including those along the paths to changed regions) wherever possible rather than duplicating them as was done by Magnusson, Asklund, and Minör [20]. The change report guides the semantic analysis and presentation systems in making their own updates: the summary information enables these and other clients to update their own annotations or databases efficiently by labeling paths to changed areas.

As with other language support services, analysis uses common algorithms across different languages wherever possible. However, existing formal languages sometimes have features that are ill-suited to incremental environments. In such cases, language-specific overrides either customize or replace the standard runtime analysis support. When this is necessary, our design encapsulates such code in its particular language class, rather than in the implementation of Ensemble itself.

2.2.1 Semantic Analysis

Formal documents such as programs generally have a context-sensitive structure that cannot be captured completely by a context-free grammar. The additional context conditions, such as name scoping and type conformance rules, are conventionally enforced by a *static semantic* analysis, which examines the document's abstract syntax tree created by a preliminary context-free syntactic analysis and reports violations of the

[1] Navigational *abstraction services* provide a view of the structure tailored to a particular client's needs. The design of Ensemble will support extensions in which the leaves are nontextual.

contextual constraints. The information obtained from the static semantic analysis is important not only to determine well-formedness, but also as the basis for semantic-based presentation, navigation, and modification, for interactive user queries, and for additional analyses. For example, semantic analysis is needed for language-based cross-references and call graphs, and for services based on slicing [34].

Ensemble incorporates a flexible multilingual static semantic analysis facility, called Colander II [19]. Colander II consists of an analyzer generator that produces directly-executable analyzers from high-level declarative specifications, which can then be dynamically loaded into the Ensemble environment as needed. Unlike similar systems [14, 27], Colander II is not biased toward any specific user service such as the reporting of compile-time errors. Its charter is instead to maintain a richly-annotated view of the analyzed document. For example, error diagnostics are but one kind of information, neither more nor less privileged than any other. Colander II takes a database-oriented approach, in which the way analysis results are presented is largely independent of the way they are computed or the uses to which they will ultimately be put. As a result, an ad-hoc query facility can be supported, thus generalizing existing browsing tools, as well as providing the ability to add new analysis functionality supporting additional user services without altering the pre-existing analysis specifications.

As with other analyzers already available in Ensemble, Colander II analyzers execute incrementally, meaning that the evaluator attempts to minimize the amount of computation performed in order to restore the consistency of the analysis database after a change has been made to the underlying document. Because of the nature of the analysis, there are significant space/time tradeoffs involved in making the results of previous computations available for possible later reuse, Colander II provides for fined-grained programmer control of these tradeoffs via a system of pragmatic annotations in the analysis descriptions.

Analysis specifications are written in a style similar to attribute grammars, but with extensions inspired by the work of Ballance [3] and Horwitz [13]. Our specification metalanguage provides unusually rich modeling resources adapted from deductive and object-oriented database technology, including objects and relations. While justifiable entirely on the grounds of modeling expressiveness, judicious use of objects and relations can make the dependency structure of computations on aggregate values more explicit. In this way, we expect to overcome the well-known limitations of attribute grammars in an incremental setting.

2.2.2 Supporting External Analysis Tools

Not all analysis tools are internal to the system. An outside agent can interoperate with documents by receiving change reports to guide its own analysis (either incremental or batch); this combines the abstract update interface with a notification protocol [30]. Annotations placed by external agents also become part of the document content: placing and updating these annotations create new document versions and subsequent change reports to other clients interested in those annotations.[2]

3 The Presentation Subsystem

3.1 Documents, Presentations, and Views

The presentation subsystem of Ensemble manages the display of documents. To do so, it maintains two classes of objects, *presentations* and *views*. As illustrated by Figure 2, each Ensemble document may have many presentations and each presentation may have many views. Changes made to the document appear immediately in every view of every presentation of that document.

A *presentation* is an abstract, formatted version of a document. It is formatted in the sense that it embodies a collection of decisions about the appearance of the document's primitive objects and about their layout and composition into a unified whole. However, these decisions are made in a device-independent manner, with layout being performed in an abstract space and no pixels or ink actually being generated.

A *view* is a connection between a presentation and a device, such as a screen window or printer, and is responsible for drawing the document on that device. Typically, it displays just a part of the area formatted

[2]Non-incremental analyses may choose to employ a separate, parallel change report in order to avoid compromising the incrementality of additional tools receiving the change report.

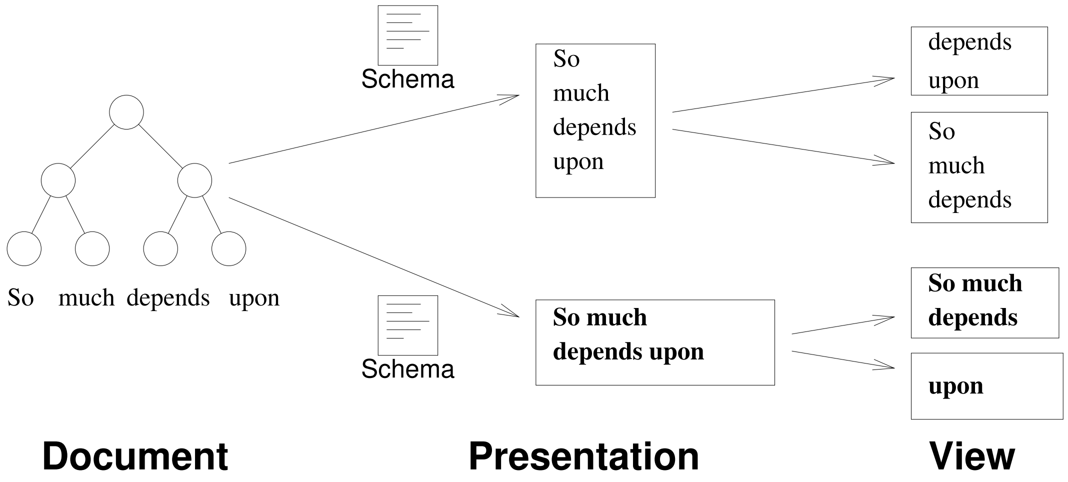

Figure 2: Each document may be formatted in multiple *presentations*, and each presentation may be visible in multiple *views*. Here we show two presentations and four views of a simple text document.

by the presentation, translating the presentation's coordinates to those of the device. A window-system view also includes such user-interface objects as scrollbars and menus for viewing and editing the document.

3.2 Extensibility to New Media

In designing Ensemble, we wanted to provide support for many different media. At the same time, we knew that only a few media would be supported by early versions of the system. Consequently, extensibility to new media was a central goal of the design. Ensemble's document model, which supports documents composed of primitive data from many media, is one way this goal is achieved. Extensibility in the presentation domain is possible for two reasons.

First, Ensemble has an explicit description of the formatting information required by each medium. It specifies the set of primitive types that the medium supports, the formatting parameters used to determine the appearance of these types, and the dimensions in which they are laid out. Proteus (see next section) uses this description to configure its services appropriately for each medium.

Secondly, there is a clear division between medium-independent and medium-specific tasks in the presentation subsystem. The primary medium-specific task is the formatting of primitive objects. The medium-independent tasks include management of formatting information, event processing, management of cursors and selections, and support for compound documents.

3.3 The Proteus Presentation System

The generality of Ensemble's support for multiple presentations is one of the system's most distinctive features and is a major extension of the multiple representation model proposed by Chen and Harrison [6] and implemented in the VORTEX system [5, 7]. In order to provide this support, a clear separation must be made between the document's content and logical structure on the one hand and the specification of its appearance on the other hand.

Proteus [12] is the module of Ensemble that manages appearance specifications and, as a result, is critically important to Ensemble's support for multiple presentations. The appearance specifications handled by Proteus are called *presentation schemas*. Like style files for batch text formatters, presentation schemas are stored separately from and can be modified independently of the documents to which they apply. They are written in a simple, declarative language and each one applies to an entire document class.

Limited support for multiple presentations can be found in several other systems. The SUIT user interface toolkit [24] provides multiple widgets for certain simple objects. Chiron [16] can support an arbitrary number of artists for the same abstract data type. However, an artist defines both presentation and interaction and is written in a superset of Ada. Thus, artists are more heavyweight than Ensemble presentations and are not intended for customization by end users. The Grif structured document editor [10, 25] introduced the presentation schema approach. It allows a single schema to define multiple presentations (called *views* in Grif), but there are restrictions on the ways in which they can differ. Grif's schema language is less general than that of Proteus and the system's design makes extension to non-textual media difficult.

3.3.1 Presentation Services

Proteus provides four presentation services: tree elaboration, attribute propagation, box layout, and interface functions. Each presentation schema essentially describes a particular way of applying these four services to a document class.

The *tree elaboration service* is used to generate material that is not present in the document itself, such as section numbers or decorative borders. This service creates the *presentation tree*, which includes all the nodes of the document plus new nodes containing the generated material. In the implementation of the presentation tree, the document's nodes are shared by all of its presentations while each presentation has a private set of generated nodes.

The *attribute propagation* service is used to define the parameters required to format document elements. The parameters of each medium's formatting operations are modeled as attributes of the nodes in the presentation tree. The value of these attributes may either be constant or they may be constrained to depend on the attribute values of other nodes.

The *box layout* service is a generic layout service based on the concept of nested boxes. Each medium lays out its objects in one or more dimensions. The box layout service supports four attributes for each of these dimensions whose generic names are *extent*, *minimum*, *maximum*, and *center*. However, the four attributes are renamed for each dimension so that, for instance, the horizontal dimension in the text medium gives them the names *Width*, *Top*, *Bottom*, and *HMiddle*. The values of these attributes can be defined and used just as with other attributes, except that special mechanisms handle the fact that there are only two degrees of freedom among the four attributes. Thus, the layout of a presentation is specified by defining constraints between the dimensional attributes of different nodes of the presentation tree.

The *interface function* service provides callbacks from Proteus to other parts of Ensemble. It allows the appearance of a document to be based on information not directly managed by Proteus, including medium-specific information, editing state, and language analysis results.

3.4 Events

When a document is changed, all its presentations must be informed, so they can recompute their formatting. When a presentation changes, all its views must redisplay themselves. Ensemble implements these one-to-many communications with *events*, similar to those described by Sullivan and Notkin [30]. There is a set of events associated with each document. Presentations of that document *register* handlers for these events. When a document event (announcing, for example, a text insertion) is fired, a handler method in each registered presentation is invoked. Similarly, presentations provide a set of events for communication to views.

3.5 Cursors and Selections

A cursor is the visible representation of a location in a document. A selection represents a continuous range of locations. These user-interface objects are created by the control subsystem, but their implementation is part of the presentation architecture. They are instances of a general class called *parasites*, whose display is managed with documents, presentations, and views, though they are not actually part of the document. An important goal in the design of parasites is that the control subsystem be free to attach them at any level of the hierarchy, so that from the user's point of view, a cursor may seem to belong to the view, the presentation, or the document.

When a parasite is changed — for instance, when the user moves a cursor — it must be redrawn on each view which shows it. Accordingly, documents and presentations both provide special events for erasing and redrawing parasites. To erase a cursor which is attached to a presentation, the presentation's erase-parasite event is fired, with the cursor as an argument. Each view of this presentation catches the event and erases its image of the cursor. If the cursor is attached to a document, the procedure begins with the document's erase-parasite event; each presentation of the document catches this, and fires its own event to delegate the erasing to the views.

3.6 Compound Presentations

Ensemble generally represents multimedia objects as compound documents: regions of different media are complete subdocuments of the appropriate medium. Subdocuments may be structurally nested to arbitrary depth; the subdocuments form a tree within the tree structure of the document.

The subdocument tree is mirrored at the presentation level, in a tree of subpresentations. The subpresentations adhere to a protocol by which they notify their parents when their sizes change. At the view level, there is a tree of *renditions*, objects which are specialized for drawing data in a given medium onto a given device. There is one class of view for each device type, and one class of rendition for each combination of device and medium.

4 The Control Subsystem

4.1 ExL, The Ensemble Extension Language

To a user of Ensemble, the system components we have discussed so far appear to be a collection of passive services. To turn these services into an integrated, interactive editing environment we need a uniform mechanism to define a control interface. In particular, we need a mechanism that supports editing of documents based on the document's class and on the actual structure of a given document instance. We also need a mechanism for user preferences and customization. The Ensemble Extension Language, ExL, fills these roles.

ExL is inspired by GNU Emacs Lisp [18] but is semantically a subset of Common Lisp [29]. Details of the implementation can be found in Dennis's report [8]. ExL supports documents, views, presentation, nodes, and languages as primitive datatypes. Many of the low-level editing services on these datatypes, described earlier, have been exported as primitive functions in ExL itself. ExL also incorporates an interface to the graphical user interface toolkit Tk [23], thus providing a means to capture user input.

The ExL interpreter is the driving engine of Ensemble. All input events are processed through ExL code. User customizations and extensions to the system are written in ExL. The interpreter is implemented in approximately 10,000 lines of `C++` code and currently supports 1,000 lines of ExL code. As an example, text editing interaction is written in a few hundred lines of ExL.

Using ExL to build the modules that control and coordinate Ensemble services and abstractions makes available syntactic abstraction (macros), first-class closures, latent typing, uniform syntax, idioms for tree manipulation, and garbage collection. These language features ease the task of building editors out of Ensemble services. The large majority of user-level editing functionality is written in ExL. We have also implemented ExL so that the core services can uniformly interact with user customizations. `C++` code can call on the ExL interpreter to examine variable settings or invoke functional "hooks" written by the user. Setting preference variables and writing extension code are identical operations in the Ensemble system, presenting one simple paradigm of customization.

4.2 Presentation Customized Control

A key issue in the design of Ensemble's control subsystem follows from the separation of presentation and control. Standard editing operations are a mixture of presentation-oriented operations (e.g. delete a line) and structure-oriented operations (delete an item from a list). The way a document is presented also influences the operations available over different portions of the presentation. How does Ensemble maintain the correspondence of presentation and structure for editing when they have been separated to support multiple presentations?

We solve this problem by turning user interaction into a two-phase process. In the first phase, the input event is used to determine the action to perform. When a user event is captured (through the Tk/ExL interface), the view transforms the location of the event from the device coordinate system into presentation coordinates. Based on this location, the system selects an ExL abstraction called a *control sheet*, which maps the event into a user action, expressed as an ExL closure. Control sheets can be organized into *control stacks*, which provide a mechanism for layering within the event resolution process.

In the second phase of interaction, the presentation subsystem is called on to transform the event position into a structural location in the document, and invokes language services to change the document. Presentation coordinates do not necessarily map one-to-one to structural locations. Therefore, an editor usually supplements the presentation subsystem's services with auxiliary data structures to assist in mapping to particular structural locations.

For example, in natural language text documents, such as articles, the text associated with the leaf nodes is usually so long that it must be broken into lines. *Lines* are a presentation notion; the document representation simply encodes text as a sequence of characters. A tricky situation occurs at the end of lines. In the presentation, there are two presentation locations, the end of one line and the beginning of the next, which map to one document location, a point between two characters. Thus, we have implemented in ExL an abstract data type called the *twig* which maintains the correspondence between the two dimensional space of a formatted paragraph and the linear locations of document text. Abstractions such as twigs help Ensemble maintain the separation between presentation and structure.

4.3 Structural Document Editing Based on Document Class

Most document systems that recognize different classes of documents use the class to customize editing. GNU Emacs is a simple example; a document belongs to a particular class if its filename matches a regular expression. An editing "mode", or set of editing operations, is defined for each class. The Pan program editor [2] uses a more sophisticated definition of a document class, defined by a language. However, Pan has no mechanism for providing different editing operations in different parts of the same document, a necessary feature in a system supporting compound documents.

In Ensemble, each document is distinguished by its class and each presentation of the document is distinguished by its medium. Our experience with the system has shown that, by and large, the style of interaction that is appropriate for a document is not determined by the document's medium per se (e.g. text, graphics, video), but rather by the types of primitive objects it contains (text string, graphical shape, video clip) and its class (FORTRAN program, Gantt chart, security camera record).

The first level of document-class-based control is determination of the actions appropriate to a given document. When a document is first loaded into the system, an ExL *control initializer* is executed that determines which control actions are appropriate to the document, and enters them into the document's control sheet. Since the grammar of the document's class is a rich definition of all its possible forms, the control initializer can query the grammar to determine which operations are germane to the document instance.

In addition, individual editing operations themselves may consult the document's grammar and other specifications. For example, they may consider whether edits will keep a given document within its document class. This mechanism also allows actions to discover which operations, of the set defined for the whole document, are valid at specific locations.

A short example may help here. One document class supported by Ensemble is the class of FORTRAN programs. When a FORTRAN program is first opened, its document class is attached and the document is formatted by the presentation subsystem. Then Ensemble turns to establishing control bindings. With both the class and the instance in hand, the control initializer first determines that various FORTRAN-based operations are needed and incorporates them. The initializer can also enable customized operations that maintain, for example, local comment conventions. Doing so may require that the initializer traverse the document locating and checking comment regions.

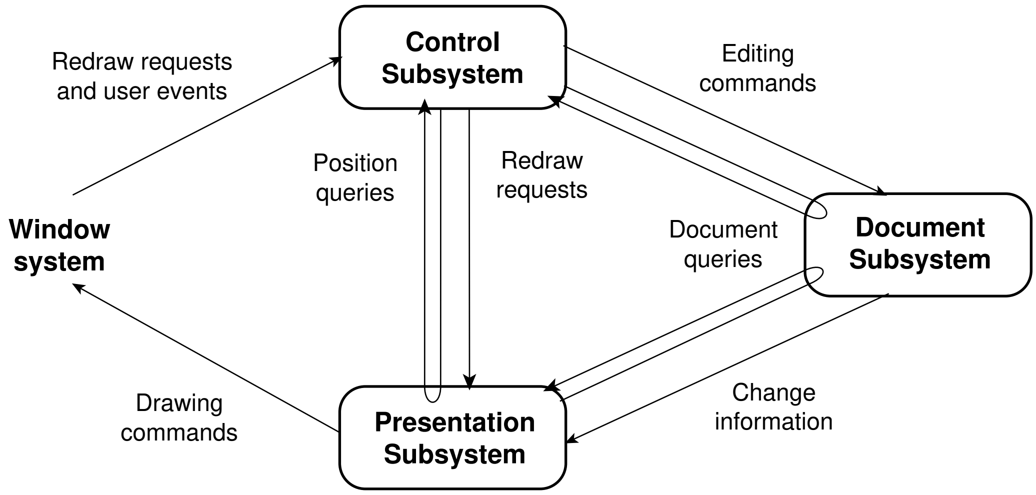

Figure 3: The overall architecture of Ensemble. Straight arrows indicate commands or informative events. U-shaped arrows indicate query-and-response interactions.

5 Integrating Document, Presentation and Control

We have now presented the three major subsystems of Ensemble: document, presentation and control. In this section, we show how they interact to form an editing environment. Figure 3 shows the relationship of these three subsystems.

In handling user actions, the usual flow of control is counterclockwise. Consider what happens when the user inserts a character into a text document. The event (typing a key) is caught by the control subsystem, which generates editing commands. These change the underlying document, which propagates change information to the presentation subsystem. Formatting changes are recomputed here and translated into commands for the display device. Similarly, if the user invokes analysis or a service provided by some other tool, that "edit" event is passed by the control subsystem to the document subsystem as a command, the document subsystem changes the document, and, if the changes are relevant for presentation, the change information is propagated to the presentation subsystem.

Information flows in other directions as well, through queries. The control and presentation subsystems, for example, query the structure of the document. Also, the control subsystem uses a narrow query interface to glean essential layout information. If the user clicks the mouse on a text display, the control subsystem queries the presentation subsystem to translate the location of the click into document coordinates: for instance, which letter of which paragraph was hit.

Apart from this interface, control and presentation are entirely separate. In other words, Ensemble follows the model-view-controller (MVC) architecture espoused by the architects of Smalltalk [17]. In practice, most Smalltalk programs merge the view and the controller [9]. In Ensemble, though, there is a special design constraint which motivates the separation: the need for multiple presentations of each document. When these presentations are closely related — for instance, one- and two-column versions of the same document style — much of the control code will be the same. The independence of the presentation and control subsystems makes it easy to reuse the control code with a different presentation.

6 Extending the Document Model to Non-Textual Media

So far, all our examples have been drawn from textual document classes. The Ensemble document model is designed to be extensible to new media, such as graphics and video, which play a growing role in the modern software development process. In this section, we discuss the special implementation problems they present. Some Ensemble development has been carried out in other media, such as three-dimensional animation and

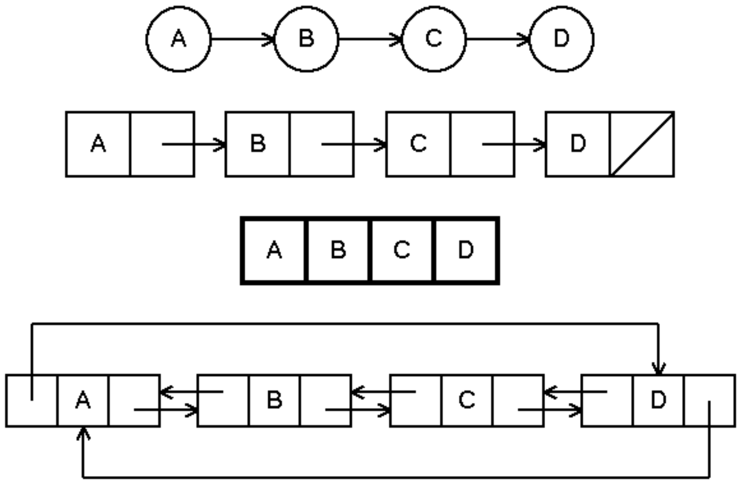

Figure 4: Four presentations of a simple list structure. All were extracted from a screen dump of an actual Ensemble session.

computer music, but they do not form part of the present system.

6.1 Graphics

Like all other graphics editors, Ensemble builds pictures from a set of primitive graphical types. The graphics medium contains points, line segments, rectangles, and ellipses. Also, it contains *compound primitives*, for standard shapes which are composed of several primitives. For example, the "polygon" compound primitive is a subtree which contains point primitives as children. Special rendering routines interpret the points as vertices for the polygon and then render the data appropriately. A similar compound primitive is also included for connected line segments that don't form polygons. Bézier curves are supported as variations of these compound primitives. With this set of primitives we can construct all basic shapes demanded from a typical graphics system, and additional primitives can be added as needed.

The appearance of a graphics primitive is completely specified by its type and its presentation attributes. Unlike a text node, which has some content (its characters) which is invariant, any of these attributes — size, orientation, location, color, fill pattern, etc. — may be computed by the presentation system. If the user wishes to make an attribute invariant, for example to color a rectangle blue in all presentations, this attribute value may be stored in the document, as a presentation override.[3]

As an example of multiple-presentation graphics in Ensemble, consider the documentation of a data structure in a program or specification. The programmer knows that the data will be a sequence of labeled nodes. The choice of implementation — linked list, doubly linked list, vector, Lisp-style external list — is secondary. For documentation purposes, it may be simply a matter of presentation.

In Figure 4, we illustrate this with four presentations of a simple graphics document. The grammar of this document specifies a sequence of labeled nodes; the arcs, circles and boxes of the figure are all created by presentation rules. All four presentations are synchronized by the document; if the user deletes the node labeled A from one, it will disappear from all of them, since each is derived from the core document. In this example, the rule-based layout of Proteus is acting like a constraint system, such as ThingLab [4], or the Garnet user-interface specification system [22].

[3]A *presentation override* is a persistent value for an attribute that is provided by the user and overrides the specifications of any presentation schema.

6.2 Video

Video is an increasingly important medium in software development. Sales, training, and even documentation are handled with video documents; many commercial software packages are shipped with a video cassette. These videos are part of the system of documents surrounding the application, and should be managed together with the rest of them by the software engineering system.

The wide variety of video documents may be regarded as a range between two extremes. At one end are the "art" documents supported by such commercial video-editing systems as Adobe Premiere [1]. The goal of these programs is to give the user absolute control over the appearance of the document; for such applications as movies and commercials, users will invest as much effort as is necessary to achieve the ideal result. At the other extreme are more utilitarian documents, such as instructional video or on-line catalog entries. Because these documents are often members of a series, and because they are created in large quantities, the software that creates them should make it easy to achieve a correct, stylistically uniform presentation.

Ensemble's document model is a natural one for uniformly structured video documents. For instructional video, title slides may be generated automatically, and the sequencing of shots from a recorded lecture may be managed with a minimum of user input.

Grammar-based "structure" is less relevant to commercials and movies, but Ensemble's multiple-presentation architecture is clearly useful. Many systems, including Premiere, provide two views of a video. In one, the user manipulates video clips on a system of "tracks", where the X axis represents time and the Y axis displays the control structure. In the other view, the video is completely formatted in time, and the user plays it back using tape-like controls. From the point of view of Ensemble, these are simply two presentations of the same document.

At the time of writing, Ensemble has limited support for video. Through presentation schemas, the user can specify the layout of several video clips in time — sequencing them, centering one on another, etc. This is simply a demonstration that the Proteus formatting model is valid for a new medium. More advanced video editing will follow in the next version of Ensemble.

7 Conclusion

We have described the document architecture of Ensemble, an integrated environment for software development. By applying a document model to all the artifacts of software development, Ensemble is able to achieve better integration than is possible with a tools-based approach. The uniformity of this model makes it possible to construct generic services for versioning, presentation and control. The model is complex enough to encode useful structural information, but is simple enough to make specification of presentation and control relatively straightforward.

Ensemble is a working prototype system, whose core implementation consists of approximately 100,000 lines of `C++` code. We believe that it demonstrates the value of the document approach to software objects and expect it to serve as an effective testbed for future research in software environments.

Acknowledgments

The Ensemble system is the product of many hands. The current implementation includes contributions by Kannan Muthukkaruppan, John Pasalis, Tom Phelps, Lacky Shah, Boris Vaysman, and Robert Wahbe. Contributors to earlier implementations included Wayne Christopher, Takashi Ohtsu, Derluen Pan, and Steven Procter.

References

[1] Adobe Systems, Mountain View, CA. *Adobe Premiere v. 3.0 Reference Manual*, 1994.

[2] Robert A. Ballance, Susan L. Graham, and Michael L. Van De Vanter. The Pan language-based editing system. *ACM Transactions on Software Engineering and Methodology*, 1(1):96–127, January 1992.

[3] Robert Allan Ballance. *Syntactic and Semantic Checking in Language-Based Editing Systems*. PhD thesis, University of California, Berkeley, 1989. Available as technical report UCB/CSD 89/548.

[4] Alan H. Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.

[5] Pehong Chen, John L. Coker, Michael A. Harrison, Jeffrey W. McCarrell, and Steven J. Procter. The VorTeX document preparation environment. In *Proc. of the 2nd European Conference on TeX for Scientific Documentation*, pages 32–24, Strasbourg, France, June 19–21 1986. Published as *Lecture Notes in Computer Science No. 236* by Springer-Verlag, 1986.

[6] Pehong Chen and Michael A. Harrison. Multiple representation document development. *IEEE Computer*, 21(1):15–31, January 1988.

[7] Pehong Chen, Michael A. Harrison, John L. Coker, Jeffrey W. McCarrell, and Steven J. Procter. An improved user environment for TeX. In *Proc. of the 2nd European Conference on TeX for Scientific Documentation*, pages 45–54, Strasbourg, France, June 19–21 1986. Published as *Lecture Notes in Computer Science No. 236* by Springer-Verlag, 1986.

[8] Brian M. Dennis. Exl: The Ensemble extension language. Master's thesis, University of California, Berkeley, 1994.

[9] L. Peter Deutsch, April 1994. Personal communication.

[10] Richard Furuta, Vincent Quint, and Jacques André. Interactively editing structured documents. *Electronic Publishing*, 1(1):20–44, April 1988.

[11] Susan L. Graham. Language and document support in software development environments. In *Proceedings of the Darpa '92 Software Technology Conference*, Los Angeles, April 1992.

[12] Susan L. Graham, Michael A. Harrison, and Ethan V. Munson. The Proteus presentation system. In *Proceedings of the ACM SIGSOFT Fifth Symposium on Software Development Environments*, pages 130–138, Tyson's Corner, VA, December 1992. ACM Press.

[13] Susan B. Horwitz. Generating language-based editors: A relationally-attributed approach. Technical Report TR 85-696, Department of Computer Science, Cornell University, 1985.

[14] Ian Jacobs and Laurence Rideau-Gallot. A Centaur tutorial. Technical Report 140, INRIA, July 1992.

[15] R. Kadia. Issues encountered in building a flexible software development environment. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, pages 169–180, Tyson's Corner, VA, December 1992. ACM Press.

[16] Rudolf K. Keller, Mary Cameron, Richard N. Taylor, and Dennis B. Troup. User interface development and software environments: The Chiron-1 system. In *Proceedings of the Thirteenth International Conference on Software Engineering*, pages 208–218, Austin, TX, May 1991.

[17] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, pages 26–49, August/September 1988.

[18] Bil Lewis, Dan LaLiberte, and the GNU Manual Group. *GNU Emacs Lisp Reference Manual*. Free Software Foundation, Cambridge, Massachusetts, 1.01 edition, April 1990.

[19] William Maddox. *An Incremental Analysis Facility for Interactive Software Development Environments*. PhD thesis, Computer Science Division-EECS, University of California, Berkeley, 1994. in preparation.

[20] Boris Magnusson, Ulf Asklund, and Sten Minör. Fine-grained revision control for collaborative software development. In *First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 33–41, 1993.

[21] Microsoft Corporation, Redmond, WA. *Object Linking & Embedding, Version 2.0, Programmer's Reference*, 1992. Pre-release version.

[22] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer*, pages 71–85, November 1990.

[23] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[24] Randy Pausch, Nathaniel Y. Young II, and Robert DeLine. SUIT: The Pascal of user interface toolkits. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 117–126. ACM Press, November 1991.

[25] Vincent Quint and Irène Vatton. Grif: An interactive system for structured document manipulation. In J. C. van Vliet, editor, *Text processing and document manipulation*, pages 200–213. Cambridge University Press, April 1986.

[26] Steven P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, pages 57–66, July 1990.

[27] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, 1989.

[28] Richard Stallman. *GNU Emacs Manual*. Free Software Foundation, Cambridge, Massachusetts, fourth edition, February 1986. Describes version 17.

[29] Guy L. Steele Jr. *Common LISP, The Language*. Digital Press, 1984.

[30] Kevin J. Sullivan and David Notkin. Reconciling environment integration and component independence. In *Proceedings of the ACM SIGSOFT Fourth Symposium on Software Development Environments*, pages 22–33, Irvine, California, December 1990. ACM.

[31] Peri Tarr and Lori A. Clarke. PLEIADES: An object management system for software engineering environments. In *First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 56–70, 1993.

[32] Michael L. Van De Vanter, Susan L. Graham, and Robert A. Ballance. Coherent user interfaces for language-based editing systems. *International Journal of Man-Machine Studies*, 37:431–466, 1992.

[33] Tim A. Wagner and Susan L. Graham. Integrating incremental analysis with version management. Submitted to *Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1994.

[34] W. Yang, S. Horwitz, and T. Reps. A program integration algorithm that accomodates semantics-preserving transformations. In *Proceedings of the ACM SIGSOFT Fourth Symposium on Software Development Environments*, pages 133–143, December 1990.