Appears in the Proceedings of the Darpa'92 Software Technology Meeting

Language and Document Support in Software Development Environments*

Susan L. Graham†
Computer Science Division (EECS)
University of California, Berkeley, CA 94720

Abstract

Languages and documents are used extensively in software development. *Pan* is a language based editing system that provides the benefits of powerful incremental analysis of formalized language documents in a text-editing context. In the *Ensemble* project, we are extending the *Pan* language technology, creating new language-based capabilities, and incorporating new document technology based on experience with the VORTEX system. High quality multimedia presentation is being used to enhance understanding of software documents. Language-based methods provide a well-founded basis for delivering the document services.

1 Introduction

Both software development and the advanced environments that support development processes use languages, both formalized and natural, for specification and also for human communication. The formalized languages in question cover a wide range, including both traditional and new programming languages, languages used in other phases of the software life cycle, emerging languages such as those being developed in the ProtoTech projects, and a plethora of languages used within software environments. Natural language is used both as an alternative to formal notation, as in design documents and user manuals, and for informal human communication.

In the *Pan* system, we provide a variety of services that pertain to the understanding, use, and translation of formalized languages and language objects and to the presentation and modification of formalized language documents. In our research on formalized languages, we seek generality well beyond the domain of programming.

In the *Ensemble* project, we are improving and extending the language technology implemented in the present *Pan* system. We are generalizing and improving the "semantic" analysis techniques. We are adding the ability to support execution, using new techniques for *dynamic compilation* based on pattern matching and transformation technology. We are enriching the ability to present information by developing new document technology based on experience with the VORTEX system. Our earlier research on interactive document processing focused on the preparation and high-quality formatting of traditional static documents in the tradition of TEX. In order to use the document technology to support software development in an interactive environment, we are moving beyond static representations (text and simple graphics) to consider documents that incorporate sound and animation. In the sections that follow, we summarize briefly some of the things we learned from the *Pan* and VORTEX systems, and the research we are doing in *Ensemble*.

*DARPA/SPAWAR Contract N00039-88-C-0292
†graham@cs.berkeley.edu

2 Lessons from the Pan System

The *Pan* system [4, 8] is a language-based editing system that exploits language-specific information to support interactive browsing, manipulation, and modification of language-structured documents. It is based on a philosophy that the language-based services should be available as needed but should not force the user into a proscribed way of working. The system provides comprehensive language analysis in the context of a full-fledged Emacs-like text editor. Language syntax is specified declaratively. The analyzer is generated from the specification by the *Ladle* processor [6]. Using the technology of grammatical abstraction [2], an abstract syntax representation is maintained, although it is the concrete syntax that is both presented to the user and analyzed. Language documents are analyzed incrementally as they are changed and the results are available to the user only when he or she wants to see them. In contrast, most language based editors provide either partial language analysis services, as in the case of language modes in Emacs, or limited ability to edit formal languages textually, as is the case with other multi-language editing systems.

The system was proposed originally as a platform in which to develop a new way of specifying the static semantics of formalized languages [1, 3]. Using the formalism of logical constraint grammars, as embodied in the *Colander* language and processor, static semantic rules, and also other language based properties, are specified by a form of logic programming. A description processor generates an evaluator that incrementally evaluates and maintains the semantic properties of a language document as it is written or modified. The evaluator uses a structured logic database. Via the database, the information in the document is accessible to other tools and to interactive querying.

One can regard such a system as a user interface to the large collection of language-based artifacts that are built up in the course of software development. Part of our research agenda has been to discover how to provide powerful services to knowledgeable users while maintaining a coherent user model of the system and not interfering with the user's freedom and flexibility [15]. Among the important services are those that facilitate user understanding of complex software systems [14].

Our experience in building and using *Pan* has demonstrated the benefits of our approach, in particular, the use of a logic programming based incremental consistency system and the use of analysis to reduce restrictions on users. Our experience has also made evident the following issues, some of which are requiring additional attention in the *Ensemble* project.

Errors are typical. Most of the time, a document that is being created or modified will not be well-formed with respect to the formal language in which it is written, either because it is incomplete, or because the author is in the process of making a set of related changes. The user may choose to have the erroneous state persist for a very long time. Consequently, all system services that make sense must continue to be provided, particularly in light of the fact that an error in one part of the document may not invalidate the structure or properties of other parts. That means it must be possible to do structural navigation and semantic analysis on incorrect structures, and to use semantic information that is inconsistent or incomplete. The *Pan* implementation provides a partial solution but does not go far enough.

Compiler-based models are inadequate. The separation of language analysis into lexical, syntactic, and semantic phases is a useful conceptual framework, but not a good operational model in an interactive software development environment [10]. The design of some languages and the incompleteness of some language documents causes the need for semantic information in making structural choices during analysis or structural information to make lexical choices. The presence of errors means that a subsequent phase must occur even if the results of the previous one are incomplete or inconsistent.

Presentation is important. The way in which information is presented to users has an important effect on the usability of a system and on understanding the contents of its documents. The availability of nonobtrusive visual cues to signal

system state, and the use of high quality formatting, together with non-textual presentations and diverse views, can enhance understanding and usability considerably.

3 Lessons from VORTEX

Our research on interactive creation and modification of documents themselves originated with a system called VORTEX (for Visually ORiented TEX) [7]. The system was intended for the interactive development of documents using the TEX/LATEX description language and algorithms. For compatibility, we chose to use not only the formats generated by the TEX family of processors, but also the TEX and LATEX specification languages. VORTEX was a very large incremental multiple–representation system. It allowed users to change the source (TEX or LATEX) representations, or to directly manipulate an image of each page. Among the new methods developed to handle incrementality are methods for handling bibliographic citations[11] and numbering [12].

Efficiency is hard to achieve. Unlike the *Pan* system, in which multiple representations are generated on demand (i.e., no concrete syntax tree is maintained), the multiple representations in VORTEX are all explicitly represented, requiring each to be updated when another one changes. To maintain correspondences, the runtime data structures are large and the update algorithms are slow.

TEX and LATEX are the wrong languages. The computational models of TEX and LATEX are heavily biased towards nonincremental left-to-right compilation. For efficiency, many of the algorithms used by TEX-based systems are inherently batch-oriented. The languages do not lend themselves to local incremental update, nor are the processing algorithms robust in the presence of syntactically and semantically incomplete document specifications. The "back mappings" needed to support direct manipulation are complex and difficult to maintain.

Extensions to other forms and other media are awkward. On a workstation, one has the possibility to view multi-color documents, documents that change dynamically, and non-paginated documents whose dimensions change in response to change in window size. None of these aspects are easily supported in TEX, which uses paragraph-oriented and page-oriented computations, and has no real notion of color or dynamic change.

4 *Ensemble*

To facilitate broad applicability, richer language technology is being developed for *Ensemble*. The power of the syntax mechanisms is being improved. The *Colander* specification language provides a good proof-of-concept but also has some weaknesses. Most notably, it lacks the high-level abstraction and structuring that would facilitate reuse. In developing a specification for Ada for use by the Arcadia project, we are using many aspects of our Modula2 specification as an example, but the specifications do not share "code". In addition, *Colander* associates information with trees, rather than graphs, and it provides a weaker query interface than we would like. In redesigning both the *Colander* language and the implementation of the processor and the evaluator, we are not only providing more abstraction, but also more powerful query facilities for the logic database.

We are extending support to execution by combining the user interaction facilities with our new methods for dynamic compilation. Translation to executable code is realized as a sequence of description-driven translation phases, described by efficient pattern-based transformations [9]. We are developing methods to compose these transformations, so as to provide a mapping between source code and target code. Using the composed transformation, we can provide debugging support in which the user indicates all actions (for example, setting conditional breakpoints, printing or changing variable values, modifying the program) by changes to the source text, and the dynamic compiler "patches" the target code without losing or invalidating execution state or resorting to interpretive execution [5]. We intend to use the logic database to record execution information, so that

the facilities of the system are available for understanding execution activity.

In *Ensemble*, there is a clear distinction between the content and structure of a document, its appearance(s), and its presentation(s)[1]. All documents, whether formal language based (for example, programs) or natural language based (for example, user manuals) have structural descriptions (that is, an abstract syntax) specified in a structural schema. Appearance is specified by a presentation schema, which contains information about such aspects as layout, font, color, or duration. An incremental formatter renders and maintains the presentation for the user. As a simple example, Figure 1 contains a structural schema and a presentation schema for a simple memo. Either all memos, or a given instance, could have several associated presentation schemas. The algorithms are provided by the formatter.

Documents can have a variety of modes (*genres*), each of which raises some separate implementation issues. Examples of genres are ordinary text, formal language text, tables, graphs, and other simple 2D graphics, animations, etc. Each genre will have both structural schemas and presentation schemas. *Compound documents* contain more than one genre – for example, formal language text together with natural language commentary and graphic intermodule structure. Since we expect compound documents to be common, considerable attention has been placed on smooth transitions between genres.

Documents are normally displayed and modified in formatted form (the so-called direct manipulation interface). Since the various document descriptions are also documents, many of the system services can be applied to them as well. For example, structural views of documents can be provided, either in tree-or-graph form as an abstract syntax description or in textual form as a "source" document. In many instances, a document can be modified either by direct manipulation (text or structure editing) or by description-based modifi-

[1] The idea of separating the structure of an object from the way it is rendered can be found in systems from a variety of domains, including the Chiron user interface management system [16] and the Grif structured document editor [13]. However, the ways in which this idea is used differ in the three systems.

```
STRUCTURE Memo;
   Memo: To From Subject Body;
   To: text;
   From: text;
   Subject: text;
   Body: Paragraph+;
   Paragraph: text;

PRESENTATION Memo FOR Memo;
DEFAULT BEGIN
        Width = Parent . Width;
        Height = AllChildren . Height;
        Top = LeftSib . Bottom;
        Left = Parent . Left;
        Justify = Parent . Justify;
        Break= No;
        FontFamily = Parent . FontFamily;
        Style = Parent . Style;
        Size = Parent . Size;
        LineSpacing = Parent . LineSpacing;
        END;
RULES
   Memo :   BEGIN
            Left = Parent . Left + 72;
            Top = Parent . Top + 36;
            FontFamily = "Times";
            LineSpacing = 1.2;
            Style = "Roman";
            Size = 11;
            Justify = No;
            END;
   To :     BEGIN
            Left = RightSib . Left;
            Top = Parent . Top ;
            END;
   From :   BEGIN
            Left = RightSib . Left;
            END;
   Subject: BEGIN
            Left = Parent . Left + 54 ;
            END;
   Body :   BEGIN
            Top = LeftSib . Bottom + 10 ;
            Break = Yes;
            Justify = Yes;
            END;
END
```

Figure 1: Schemas for a simple memo

cation. As in *Pan*, the same services can be applied to the description languages, thereby providing a means to understand, debug, and enhance the languages known to the system.

References

[1] R. A. Ballance. *Syntactic and Semantic Checking in Language-Based Editing Systems.* PhD thesis, Computer Science Division—EECS, Univ. California, Berkeley, CA 94720, Dec. 1989. Available as Tech. Report No. UCB/CSD 89/548.

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

[3] R. A. Ballance and S. L. Graham. Incremental consistency maintenance for interactive applications. In K. Furukawa, editor, *Proc. of the Eighth International Conference on Logic Programming*, pages 895–909, Cambridge, Massachusetts and London, England, 1991. The MIT Press.

[4] R. A. Ballance, S. L. Graham, and M. L. Van De Vanter. The Pan language-based editing system. *ACM Trans. on Software Engineering and Methodology*, 1(1):95–127, Jan. 1992.

[5] J. Boyland, C. Farnum, and S. L. Graham. Attributed transformational code generation for dynamic compilers. In R. Giegerich and S. L. Graham, editors, *Code Generation - Concepts, Tools, Techniques. Workshops in Computer Science*. Springer-Verlag, Berlin, Heidelberg, New York, 1992. To appear.

[6] J. Butcher. Ladle. Master's thesis, Computer Science Division—EECS, Univ. California, Berkeley, CA 94720, Nov. 1989. Available as Tech. Report No. UCB/CSD 89/519.

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

[8] L. M. Downs and M. L. Van De Vanter. *Pan I* version 4.0: An introduction for users. Tech. Report No. UCB/CSD 91/659, Computer Science Division—EECS, Univ. California, Berkeley, CA 94720, Aug. 1991.

[9] C. D. Farnum. *Pattern-Based Languages for Prototyping of Compiler Optimizers.* PhD thesis, CS Division, EECS, University of California, Berkeley, Dec. 1990.

[10] B. T. Forstall. Programming language specification for editors. Master's thesis, Computer Science Division—EECS, Univ. California, Berkeley, CA 94720, Nov. 1991.

[11] M. A. Harrison and E. V. Munson. On integrated bibliography processing. *Electronic Publishing*, 2(4):193–210, Dec. 1989.

[12] M. A. Harrison and E. V. Munson. Numbering document components. *Electronic Publishing*, 4(1), Jan. 1991.

[13] V. Quint and I. 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, Apr. 1986.

[14] M. L. Van De Vanter. *User Interface Design for Language-Based Editing Systems.* PhD thesis, Computer Science Division—EECS, Univ. California, Berkeley, CA 94720, 1992. In preparation.

[15] M. L. Van De Vanter, R. A. Ballance, and S. L. Graham. Coherent user interfaces for language-based editing systems. *International Journal of Man-Machine Studies*, 1992. To appear.

[16] M. Young, R. N. Taylor, and D. B. Troup. Software environment architectures and user interface facilities. *IEEE Trans. on Software Engineering*, 14(6):697–708, June 1988.