Adapting Analysis Algorithms for Interactive Software Development Environments*†

Tim A. Wagner‡
Computer Science Division–EECS
University of California, Berkeley

Abstract

In an Interactive Software Development Environment, *incremental algorithms* are used to perform conventional compiler analyses rapidly and without the loss of cached information, restoring consistency to a structured document that has been modified. *Versioning* expresses the relationship of the document's content to its modification history. We present a means of integrating these formerly separate concerns: analysis is no longer a side effect, but an explicit update to the document that creates a new version.

We consider software artifacts to be annotated graph-structured documents. The contents of the leaves can be of diverse media. We describe variants of conventional algorithms for incrementally analyzing such documents, and describe how the ability to name and simultaneously access different versions of the document simplifies these algorithms considerably. The algorithms are also designed to maximize *component preservation*: detecting that components can be preserved, and reusing rather than recreating them. This avoids disruption in the user environment by renaming components in ways that destroy their associated information.

Locating changed areas, altering the current version of a document, and accessing versions other than the current one all rely on a single, efficient strategy. Each portion of the document has its history recorded in a local version tree, while a global version tree serves as a namespace coordinator to relate the local versions of the document's components. Making the primary distribution of update information *spatial* rather than temporal supports object-oriented change modeling and implementations customized for specific media. All clients—presentation services, analyzers, code generators, and external tools—use a single uniform interface to history information. The time to access to an arbitrary version of an atomic component is linear in the distance traversed within its local version hierarchy; structured components are explored in $O(\lg(|\,\text{nodes}\,|) * |\,\text{changed set}\,|)$ time. The history mechanism is general enough to support a single-level storage model that integrates source code control, filesystem checkpoints, and an editor's undo log. We discuss an implementation of this design in a working environment.

Keywords: **versioning, transactions, incremental algorithms, language technology, compilers**

1 Introduction

The analysis and consistency maintenance of software artifacts is central to the software development process. The *Ensemble* project at Berkeley is facilitating this goal by providing high-quality interactive tools for software authoring and understanding [8]. As part of the *Ensemble* effort, we have developed a new model of document history and have adapted a number of language-based analysis/transformation algorithms to

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

†An abridged version of this paper will appear in the proceedings of the Fifth European Software Engineering Conference.

‡Computer Science Division, Dept. of EECS, 573 Soda Hall, University of California, Berkeley CA 94720-1776.
Phone: (510) 642-4611 Email: `twagner@cs.berkeley.edu`.

interoperate successfully with our approach to versioning. This paper describes these algorithms as well as a general framework established to improve component preservation and change recording for all analyses and transformations.

Our prototype environment treats software artifacts as structured documents. We employ a graph-based document model in which the leaves contain a mixture of text and other media. Annotations (objects of any type associated with nodes of the document structure) exist to add additional information to this structure, as do entries in a database. Document components, annotations, and database entries are all interrelated, and an Interactive Software Development Environment (ISDE)[1] plays a key role in automatically deriving and checking important subsets of their relationships. This is accomplished through analysis and update algorithms, which transform documents by modifying their components (and (re)generating new components) in order to integrate changes made by the user to restore consistency.

All of our analysis/update algorithms are *incremental*: we believe that SDEs should be interactive, and incrementality enables the algorithms' performance to scale with document size. Equally important, however, is that incremental algorithms enable the system to preserve structured annotations—profiles, hyperlinks, comments, semantic attributes, and other information crucial to software development—by *preserving components* where possible. This means retaining elements of the document across a transformation when semantically meaningful, rather than re-creating them. Regenerating components rather than reusing them fails to preserve the context required by other ISDE services, ultimately resulting in degraded user services.

Also essential to the development process is the management of versions and configurations. The system must capture and organize both large-scale and small-scale updates. The former provide the basis for release and configuration management and are conventionally handled by a source code control system. The latter represent fine-grained operations that are typically managed by the "undo" and recovery activities of an editor. These levels are combined in our approach: the timing and granularity of checkpoints is a policy decision.

Conventional software development environments have no reason to consider the integration of analysis algorithms and history services, due to their limited document models and batch tools. Even interactive SDEs (including the earlier versions of our own system) have not treated analysis-based updates in a uniform manner with user changes. However, the incrementality that permits a fully interactive approach requires every analysis to construct a summary of relevant document updates made between its applications. The key insight is that generalizing these "tracking" services and integrating them with the history (undo) mechanism simplifies the design of both, providing additional features *and* enhancing overall performance.

The user and programmatic interfaces to software documents are unified by this approach: giving incremental analysis and update a first-class treatment means the effects of its execution (such as changes to the appearance of the document) can be undone using conventional editor services. Tools that compute differences between versions or that merge development paths (such as operation-based merging [16]) can now treat all updates in the same manner. Because annotations cached on document structure are easily maintained through this approach, information that is part of the user's context, such as debugger breakpoints or presentation elements, is neither discarded nor invalidated by the analysis-based transformation.

Analysis/transformation algorithms become easier to implement and to combine using this scheme. First, individual algorithms are easier to design: rather than attempting to read from the same version that is being destructively modified, they can modify one version while reading another *regardless of the relationship between them.* Secondly, using the history mechanism as an inter-service protocol to report changes *encapsulates* transformations: their implementations can be independent of one another. Finally, because the versioning system takes care of locating and summarizing changes, different transformations can be performed at different times and each can discover the exact set of changed components since its previous execution: there is no *a priori* commitment to the number, frequency of application, or the relationships among these algorithms.

[1] *Every* SDE is interactive in the sense that it provides the user with interactive tools, such as editors. We reserve the term 'ISDE' for environments in which the analysis, update, presentation, and code generation from software documents occur without perceptible delay.

Detecting component reuse is motivated by the goal of preserving document annotations and user context. Typically it is easier for analysis/update algorithms to regenerate information than to preserve it (batch techniques are the extreme case). But enhancing preservation makes the effects experienced by the user more natural and intuitive. It also improves the precision (and performance) of presentation services, subsequent analyses, etc., by decreasing the amount of new material they in turn must consider. Previously published algorithms preserve some unchanged components but regenerate others; their goal is standalone performance. The techniques presented here improve the detection and preservation of document components without increasing the asymptotic running time of individual analyses/transformations and typically *decrease* total delay.

Making analysis-based transformations first-class updates provides another important implementation benefit: a simple strategy for answering speculative queries. Rather than manufacturing variants of analysis algorithms to answer "what-if" questions, many speculative operations can be executed in a normal fashion and their results inspected. If the operation succeeded and was intended to be conditional, the system commits the new version and sets the cached document state accordingly. If the operation failed, or if the analysis was performed solely to answer a query, the new version is discarded instead. For example, to determine whether a missing type should be "integer", the system can insert its guess, incrementally re-analyze the document, and then observe whether the tentative change was appropriate. If not, the trial version and its effects are discarded.[2]

The approach presented here is a multilingual and multimedia one. It is independent of any particular language, but exploits language semantics to record changes compactly within the history log. This dramatically improves the performance over textual "diff" programs that, for example, do not understand the distinction between comments and program text. Since annotations, document content, and database entries can *all* be multimedia objects, the history mechanism has been designed to access component versions independent of their datatype. To add support for a new medium, one needs only to provide implementations for versioning any additional datatypes required by its representation. Thereafter, the generic history services and user interfaces can be used.

The remainder of this paper is organized as follows. We first describe the forms that documents may take and how the history services model document updates. Section 3 discusses the implementation of change management to support historical access by both user services and analyses. Section 4 describes the design of incremental analysis algorithms to interoperate with the versioning system and to improve the preservation of document components. Section 5 presents our research system and describes the experience of implementing this design within it. Section 6 reviews related work and Section 7 summarizes our major contributions.

2 A Model of Documents and Updates

A software development system is primarily concerned with the understanding and manipulation of electronic documents [23]. This section describes our model of a document, including the ways in which it may be updated and how the resulting versions are organized and named. Language-based analysis and transformation algorithms are introduced as agents that update documents, and the role of the history mechanism as an inter-algorithm reporting protocol is established.

2.1 Representing Structure and Content

A software development system is composed of many *objects*, most of which are collected into other objects, either structured documents or database relations. A structural document model is much more powerful than a textual one: for example, it can directly model the abstract syntax trees of programs or the hierarchical organization of natural language documents. The existence of structure is essential for many interactive

[2] Queries that produce a very large number of updates are not good candidates for this approach, and are better handled by *ad hoc* techniques.

services. Representing structure directly eliminates the performance bottleneck imposed by reproducing it on demand and the need for heuristics that "discover" structure without computing it precisely. Our model of structure uses tree-based representations. DAGs can be realized through component sharing, and arbitrary graph structure is created via inter-node references.

Each document carries a run-time representation of its type; for the documents of interest here, the type specifies the (formal) language in which the document is expressed. The complete document model permits components of the structure to be represented in different languages; however, to simplify the discussion we will assume that each document is expressed in a single language. The language of a document specifies the general form of its presentation, determines its editing operations, provides a database signature for it, and specializes its analysis and transformations—either parameterizing a generic service with language-specific information or providing a special variant of the analysis appropriate to the language. Language information is expressed through a collection of *declarative specifications*; these are compiled off-line, and the ones needed for the current set of documents are dynamically loaded by the system.

Any node in a document can contain other objects, but typically the leaves hold most of what is conventionally thought of as "content" (for instance, text, tokens, graphics primitives, audio/video recordings, etc.). In addition, objects can be annotated either by the user or through the actions of language-based services (analyses use annotations both to convey information about the document and also to preserve their own time-consuming computations). A database records information better expressed through an extended relational model; database references may occur in objects and database entries may consist of object references.

None of these representations of information are secondary: all of them are versioned and user-editable, and may include multimedia representations. Analysis-based transformations may examine and update any portion of a document's representation.

2.2 Modeling Updates and Versions

The components of a document, and consequently the document itself, pass through different states, corresponding to their contents at different points in time. A document has a distinguished *current* state. Modifications are performed and other states are accessed with respect to a model of the document's history. This model describes how changes to the document are related to one another and how different states of the document are named externally by the user and internally by clients.

Each major update to a document results in the creation of a new *revision*; taken together, these revisions constitute the development history. Standard configuration management composes these revisions into software releases. Each revision is the product of a *transaction*—a long-lived update that creates a new revision from an existing one through the application of zero or more *operations*, followed by a transaction commit called a *checkpoint*. In a typical source code control system, the individual operations that constitute a revision are inaccessible. However, a new revision under construction *does* have its individual operations exposed: these correspond to document states managed by the "undo" services of an editor.

Operations are composed of zero or more *actions*. Like transactions, operations are atomic, serializable, and persistent.[3] However, operations are shorter-lived, normally execute on a single machine, and do not model shared resources. An operation is terminated by committing it; the document state created by the commit is called a *version*. Because each committed operation (checkpointed transaction) is associated with a unique version (revision), we will frequently ignore the distinction between a state of the document and the process that created it. In summary, *actions* cause *changes*, *operations* (sequences of actions) create *versions*, and *transactions* (sequences of operations) create *revisions*. Figure 1 illustrates the relationship between operations and transactions. Both revisions and versions may have more than one possible modification

[3] Editors typically achieve a rough approximation to persistence by periodically flushing the contents of modified buffers to stable storage. By using a background process to write versions from extant transactions to a stable medium, we can recover the full development state, unifying the editor's crash recovery mechanism with what has conventionally been considered source code control. Unlike conventional systems, in our design there is no limit on the number of versions recovered.

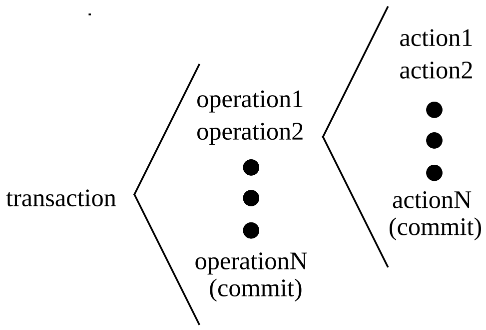

Figure 1: The relationship between transactions, operations, and actions. Only a single transaction is shown, but multiple, concurrent transactions may be active. The actions within an operation are not individually accessible. The operations within a committed transaction are typically inaccessible as well.

path, leading to hierarchical structure in the development history. (Merging paths is not treated in this paper.) When there are several alternatives, we refer to them as *variants*.

This paper assumes a design based on a single-level store, as suggested for simple text editors by Fraser [6]: there is no distinction between source code control and editor services. Whether information resides in processor memory or stable storage is a performance (and security) consideration, not a user interface concern. Revisions are thus simply names for "interesting" versions: if desired, versions between revisions may be collapsed to save storage. Multiple transactions and multiple operations within a given transaction are permitted. This framework can accommodate nested transactions [5] as well, although our prototype implementation does not support them. Figure 2 shows a small example where a single transaction is active; the breakdown by conventional services is also shown.

Environments normally limit the operations that enjoy transaction semantics to relatively simple activities on a fixed set of media (such as textual modifications to a document). Our approach places no limit on the complexity of an operation: the approach is capable of treating the interacting effects of cooperating update algorithms as easily as keystrokes. The frequency and granularity of operations is determined by policy, not by the history mechanism or by the transformations. For example, each analysis-based update can be a separate activity or part of a larger update to the document. The determination of when to commit or checkpoint may be made automatically by the system or left to the user.

Configurations are versions of modules that collectively define a project. In our approach, configurations would be arranged in hierarchies to provide description reuse. (Configurations are versioned documents in their own right, with modules named using inter-document references.) Each node in the configuration hierarchy defines one or more incarnations of the project by naming modules, whose versions are defined either explicitly, via absolute links to the desired version of the module, or implicitly, by naming a branch or subtree in the module's version hierarchy. The latter two are essentially templates that permit a concise description of an unbounded sequence of configurations. Figure 3 illustrates the several cases. When an interactive development session is active, one can also name the current version of a module; this would be used to implement the conventional compilation cycle.

Investigating sharing and locking protocols for cooperative work was not an explicit focus of this research; however, we note that our approach is compatible with optimistic locking models [18].

2.3 Analyzing and Transforming Documents

A document may be updated directly, by user changes, or programmatically, by some service within the ISDE. There may be many such services, not all of which are known when the ISDE is built. Consequently, it is important that there be a well-understood protocol for communicating the effects of changes; this is one argument for a general history mechanism. Several transformations may work in tandem, in which case they

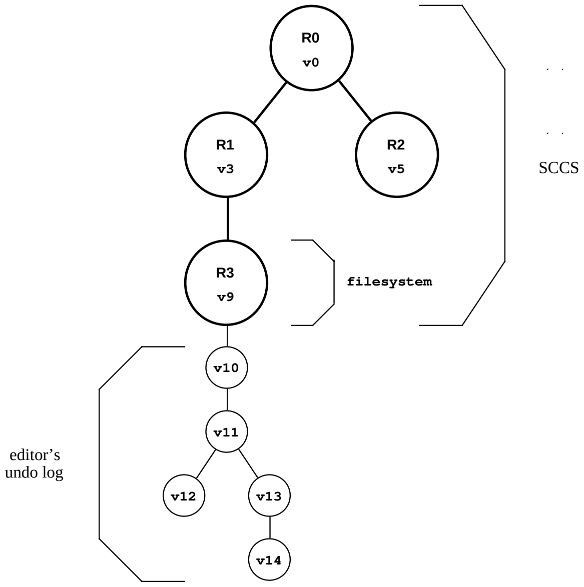

Figure 2: The tree of versions deriving from a single transaction. This example shows four revisions, R0 through R3, and their corresponding versions vi. Versions between revisions (those not selected for checkpointing) are presumably inaccessible to the user and are not shown. A single transaction is active, and its initial version is v9. All of the versions descending from it are accessible. The brackets indicate the division of labor in a conventional (not single-level store) approach: the revisions are stored in a source code control system (SCCS), and the most recent revision is duplicated in a filesystem snapshot. Some or all of the recent versions would be accessible as editor states.

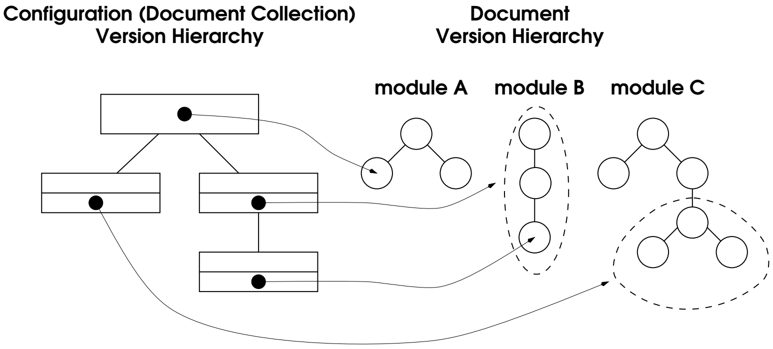

Figure 3: A sample configuration hierarchy. Configurations are composed of modules, and each configuration is represented as a list of $<module, version>$ items. This naming can be explicit, but is usually simplified through some combination of inheritance within the configuration hierarchy and implicit version naming mechanisms (templates), whose binding to actual module versions is computed only on demand. In this example, the base configuration names a single module, A. It has two variants: the first adds the latest version from a subtree of module C's history, and the second adds the latest revision from a chain in module B's history. This second configuration has its own subclass, which further refines the version of module B to a particular implementation.

will also communicate directly with one another.

The implementation of an analysis-based transformation follows a general form:

- Use a history mechanism to compare the current version of the document to its state when this analyzer last executed. This determines which components have been changed.
- Extend the changed areas to *affected areas* by including additional components dependent upon the changed ones.
- Begin a new version to avoid simultaneous reading and writing of the same version.
- Read the previous document content from the previous version.
- Write the current document content (i.e., create or modify components in the current version).

This breakdown illustrates two crucial points about such algorithms. First, multiple versions of the document must be accessed simultaneously; Figure 4 illustrates the relationship of the three versions of interest to a typical analysis-based transformation. If the ISDE lacks a history mechanism *per se*, the analysis and update must *simulate* multiple-version access in conjunction with its own activities. Secondly, the performance of these algorithms and the precision with which they report their effects on the document to subsequent analyses depends to a large degree on how successfully they detect and incorporate reusable material.

Designing analyses to be incremental has important performance benefits. In the case of a generic parser, it converts an $O(N)$ batch algorithm into $O(\lg(|N|) * |\text{affected set}|)$. (The *changed set* represents the set of components reported by the history mechanism as updated; the *affected set* is a superset that includes additional components the analysis determines were dependent upon the changed material.) But to coexist well in a setting where there may be many different analyses and transformations, these algorithms should be precise: spurious updates unnecessarily inflate the changed set for subsequent analyses. Designing each transformation to preserve as much of the preceding document as possible retains more information and improves the performance of the overall system. This also provides benefits for the user; for example, a presentation of the document that highlights changed areas will be easier to read if components are not changed unnecessarily.

Providing a single history/change reporting service that all analysis and transformation algorithms can use simplifies their implementations. Individual transformations can also omit common "cleanup" steps that are performed by the system: immediately before each operation is committed, the system locates and tags deleted objects, (re)balances repetitive structures (see Section 3.1), and retracts paths that no longer lead to changed material. These steps keep the history representation up-to-date and prevent performance from degrading as sequential transformations are applied.

3 Recording Version Information

Information recording the differences between document versions is distributed across many *version trees*. These trees, along with a global tree to unify them, jointly provide the history-related services for a document. This section describes the version trees and summarizes the history services provided to clients.

3.1 Local Version Trees

The representation of document state must be distinguished both temporally (by versions) and spatially (by components of the document). Many existing representations implement the temporal division first, such as the EMACS undo log or the RCS and SCCS source code control protocols. Prioritizing temporal distribution severs the connection between a component and its versions, necessitating component indices for each time-slice. This is not especially troublesome for a text-only representation, but is problematic with a richer data model. Our representation makes the spatial division first: each component of the document then implements its change history independently and in a type-safe manner.

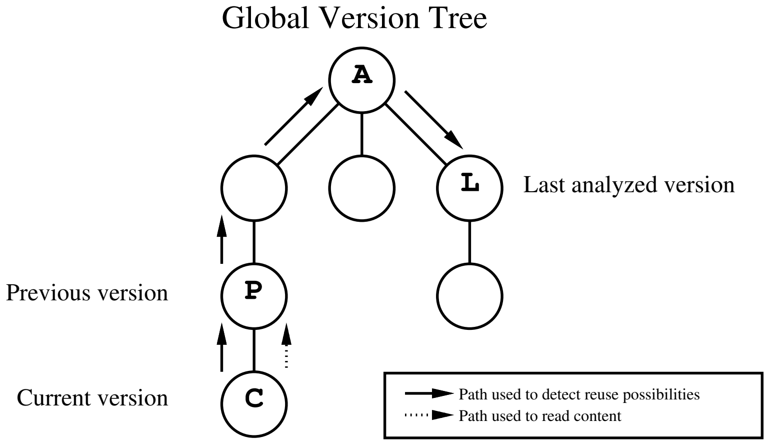

Figure 4: An incremental algorithm accesses historical information. The previous version (P) is read to provide the contents, while updates are made to the current version, C. The last version analyzed (L) assists in making preservation decisions. It may also contain annotations placed by the analysis as part of its internal data. This example was chosen to illustrate the hierarchical form the global version tree can assume; A denotes the common ancestor of the current and last analyzed versions.

There are a number of other advantages to this approach. First, it permits version control to be implemented at the user level, without specialized language constructs or compiler support. In our implementation, versioning utilizes the normal C++ type system, rather than requiring a dynamic type description mechanism. Our approach also permits *fine-grained versioning*: it is capable of tracking changes to individual fields within document nodes. Finally, our alternate decomposition is the key to multimedia support: since each component contains its own type-specific local version tree, different media are able to choose their own representation schemes. Even for a given medium, such as text, different languages or document sections can employ different storage representations. For example, updates to paragraphs of natural language have different time/space tradeoffs than changes to identifiers in a program. By subclassing generic version classes, new datatypes (and hence new media) can be added, even while the system is active.

The complexity results presented here refer to the number of nodes examined and the size of the version trees. Obviously, the representations chosen for particular datatypes will affect the running time of history-related services (such as changing the current version of the document) by affecting the speed with which a component can be updated. However, there is one case in which the representation affects the global complexity results as well: in order to limit the search overhead to $O(\lg N)$ on structured components, it is important that any repetitive sections ("lists") be represented by balanced binary trees. This has the added advantage that structural nodes all have fixed arities. The interfaces to be described in Section 3.4 also require the existence of versioned boolean values, in order to indicate the nodes within structured components that lie on a path from the root to a changed region.

Individual nodes typically contain a mix of versioned and transient fields (the latter being, in effect, a distributed cache). Each component is responsible for making its current state available to clients and for converting its representation into a read-only form when the containing operation commits (normally the read-only form is more compact than an actively-edited one since it need not support updates). The representation in the local version tree can be entire snapshots (useful in the case of small datatypes, such as C `int`s) or deltas (changes from the previous state). When only differences are recorded, it is necessary for the component to maintain its current state explicitly; Figure 5 illustrates delta-based versioning for a string datatype.[4]

[4]Nodes in both the local and global version trees are often referred to as *deltas* since it is suggestive of the change-based representation used for compound datatypes. When discussing the implementation of version hierarchies, we will typically use the terms 'node' and 'delta' interchangeably, and in preference to conceptual terms such as 'revision' and 'variant'.

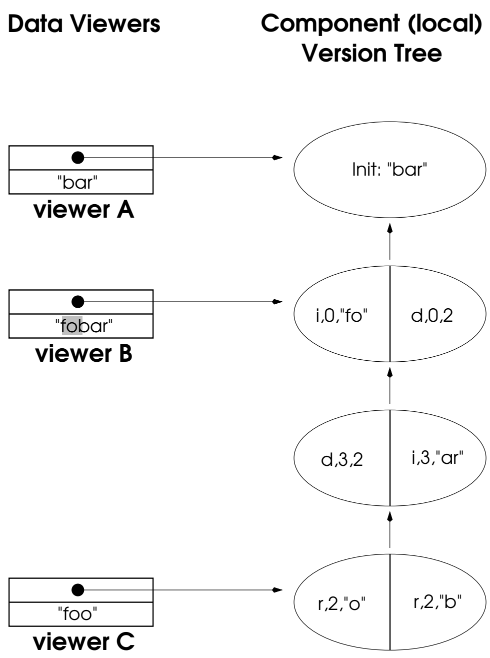

Figure 5: Example of versioning a string datatype. Each node in the local version tree encodes a two-way delta that can be used to apply or remove this change to the underlying string. Here the forward delta is shown on the left as an operation code (insert, delete, or replace), offset, and string or character count triple. The backward delta is shown to the right, though it would not be represented separately in an actual implementation. The root of the local version tree contains the original string contents. All viewers cache a logical copy of the string they are viewing, though they may represent the strings in different ways. Here viewers A and C use simple snapshots, while viewer B shows a delta (deletion of two characters) being applied to a base ('fobar').

A component's version tree will normally include distinct nodes to indicate special types of updates. These include a creation node as the root and one ore more deletion nodes (which always appear as leaves). Purged information (branches of the development history that have been explicitly pruned) appear similarly as "information unavailable" leaves.

3.2 Global Version Trees

The local version trees, taken together, represent the history of a document. However, they require additional administrative support in order to be a useful service: this is a role of the *global version tree*. It names each committed operation with a distinct node; the local version tree of each component modified by a given operation references the corresponding node in the global version tree. To clients, the global version tree is the manager of a document's version namespace: it is used to map universal ID's into particular versions and to create, access and name versions.

Every action (update to a component) occurs as part of an operation. To modify a component, its update method must be given a handle to the proper operation. Operations are named in the same manner as versions, by nodes in the global version tree. Having an integrated namespace for both operations and (committed) versions permits the update and query methods on components to use similar signatures. Operations are begun implicitly when a tool attempts to modify an already committed version: because it is marked read-only, the global version tree will create a new child of it to represent the changes that are about to be made. This new node then serves as a name for the operation until its commit (or abort) point.

The global version tree can be used in a conventional arrangement where source code control and editor maintain separate storage, but its design encourages a single-level store that removes the distinction. Referential locality permits caching only recently-visited subtrees, while the remainder of the development history resides in secondary storage. To provide intra-transaction recovery, data associated with recently committed versions can be incrementally flushed to stable storage.

3.3 Projecting Global Versions Onto Local Ones

Client applications and tools request information through the global namespace, but specific results come from the data stored in local version trees. To view a component's value at a time other than the present, a client indicates the desired global version. Consequently, one of the central algorithms is the *projection* of a global version onto a local one. Each local version tree represents a homomorphic contraction of the global tree under the *ancestor-of* relation, so this projection is always well-defined. Figure 6 shows a global version tree and one of its local trees, along with the explicit associations between them.

Until an operation is committed, the node in the global version tree representing the uncommitted version will maintain a mapping to all of the affected local version trees (this is necessary in order to transmit the commit or discard action to them). Space restrictions prohibit explicitly mapping *every* global delta to its corresponding local deltas (especially since we desire that version construction be inexpensive enough to use whenever it is semantically meaningful). Consequently, this projection is computed on demand for committed versions of the document.

For performance reasons and to simplify query methods in the common case, each document has a cached version representing its current state (of course, any number of versions could be so cached). This corresponds to a single node in the global version tree *and each local version tree will have the projection of this version as its current delta*; this is the invariant on which our projection algorithm relies. The projection algorithm used in our prototype is shown in Appendix A. It works by walking upward in both the global version tree and a local version tree until an associated pair of nodes is found. At that point, it retraces its path in the global tree, determining the correct local alternative to choose at each step. When the original node is reached in the global tree, the local projection has been discovered.

The time it takes to calculate a projection depends on how recently the current local value was updated and

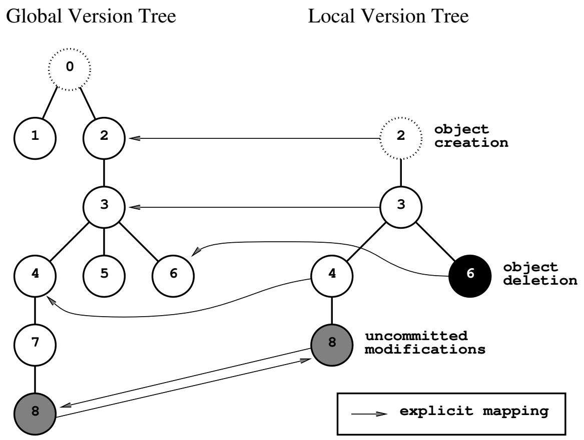

Figure 6: Each local version tree is a projection of the global one. A single local tree is shown here: its root node represents the creation of the object, and one terminal node indicates that the object was deleted in that version. The other terminal node shows an uncommitted operation: it is the only case where a global-to-local association is explicitly recorded.

how distant its corresponding global version (inverse projection) is from the desired version. The running time is also dependent on the branching factor of the local version tree, but this factor can typically be ignored: clients simply do not create a vast number of simultaneous alternatives for a single component. To simplify the analysis, we can assume that the algorithm traverses two distinct paths in the global version tree: between the current version and the desired one, and between the current version and the inverse projection of the local component's current delta. We can then state the worst-case running time as:

$$O((p_1 + p_2)b)$$

where

p_1 denotes pathlength(global-current, desired-version),
p_2 denotes pathlength(global-current, project^{-1}(local-current)), and
b denotes the local branching factor

Because most queries are for recent versions of recently changed components, this is often a near constant-time operation.

3.4 Accessing Historical Information

The access to historical information needs to provide user-level functionality (e.g., serving as the basis for the undo log) and to be sufficient for the needs of analyses, as described in Section 2.3. There are three primary interfaces to historical information:

- Locating and summarizing changed areas
- Retrieving a different version of a component
- Altering the current version of a document or document sub-component

Many clients do not track *every* change; rather, they update periodically or on demand. When this happens, they need to know which components have been altered since the last time they executed.[5] To discover whether a component was changed, the system determines the number of deltas that exist on the local path

[5] This is the most common type of request to find changed locations: one in which the query covers changes that occurred between a past version and the current one. However, the algorithm is fully general and can summarize changes between *any* two versions.

between the projection of the starting version and the projection of the ending one. Since each delta indicates a revision, a path of length greater than one indicates that modifications occurred in that time period.

Locating the changed areas of structured components (such as our software artifacts) utilizes *path bits*: boolean annotations on nodes, which are themselves versioned, that guide a client in performing a walk over the structured component to locate any changes in its structure, content, or annotations. Because repetitive structures are represented via balanced binary trees, locating affected components requires only $O(\lg|\text{nodes}|)$ overhead. All of the clients, especially the incremental algorithms, rely on this property to make their own update computations efficient.

Retrieving the non-current version of a component proceeds similarly, but the target and source times are switched: the search begins with the current version and ends at the desired one. For datatypes versioned by snapshot, the system need only project the desired global delta onto the local version tree and retrieve the value stored there. When changes are stored differentially, the entire path through the local version tree must be constructed and traversed in order to compute the new value.

Altering the current version involves the same algorithm as retrieving a non-current version of a component; it merely updates the local version tree's cached value instead of returning the result. Altering the current version of a structured component's root brings the entire component up to date through a recursive traversal; again, the path bits make this an efficient operation by limiting the update to only those regions that require it.

The global version tree *per se* is not an undo log. Deciding which operations are undoable, how many may be undone, etc. is a policy determination. However, the implementation of the undo log is made trivial by the presence of the global version tree: the undo log simply records a (variable length) path through the global version tree that represents visited document states. A direct manipulation-based GUI for version control and selection is currently being designed alongside the conventional serialized undo model.

The user interface benefits from a generalized history mechanism, but it is the presence of incremental algorithms that justify its incorporation as a standard element of any ISDE. These analysis algorithms can exist without access to a versioning scheme of the generality presented in this paper, but when they do, they must simulate a portion of the historical information in parallel with the actual transformation of the document. This added burden complicates their design: it forces them to examine the structure or content while simultaneously updating it. When two or more analysis algorithms work in tandem to achieve a result,[6] the problem is exacerbated: every algorithm's implementation must avoid changing a component until *all* the algorithms have read it. The history mechanism encourages encapsulation even among cooperative analyses.

4 Incremental Language-Based Analyses and Transformations

The documents with which we are primarily concerned are written in formal languages, including formal specifications, configuration descriptions, and language definitions. An ISDE will model these documents as annotated, structured objects whose type specifies the language in which the document is written. A collection of analysis services that can derive and check the lexical, syntactic, and semantic properties of such objects is a core feature of an SDE. We will focus on incremental implementations of lexing and parsing: these exist in a functioning state in our prototype and also serve to illustrate the general framework that has been developed.

The impact of history services is especially important here: language-based analysis and transformations are frequently invoked, can make large-scale changes to the document structure, and typically precede the application of other services (presentation, profiling, optimizing transformations, etc.). These characteristics require that the incremental algorithms designed for an ISDE be both fast and precise.

[6] This is required by languages like C that have circular dependencies among the lexical, syntactic, and name resolution stages.

Method	Description
Name	retrieve name of the language
Version	retrieve name given to this release of the language specification
Grammar	retrieve a shared copy of the grammar
IncAnalysis(*subtree*)	restore consistency to document's representation within *subtree* (default scope is the entire document)
Language(*filename*)	construct an instance via dynamic linking
Language(*classname*)	construct an instance from a statically-linked language class

Table 1: Abstract interface to run-time language objects in Ensemble. This is the interface to the C++ base class `LANGUAGE`; additional base classes are interposed between it and any actual languages to define additional services, including the application of the generic analysis algorithms described in this paper. Writers of language descriptions may wish to make additional query facilities and run-time representations of the language (e.g., the textual representations for the concrete syntax of an operator in the language) available to clients. Such facilities are placed in the `PROGRAMMING_LANGUAGE` class, which is derived from `LANGUAGE`.

4.1 Language Object Model

Languages are specified via formal definitions, which are compiled and loaded into the running system. Languages are organized into a (relatively flat) hierarchy that permits minor variations in a language to be easily expressed and implemented. The hierarchy also enables the replacement of generic analysis services on a per-language basis when some feature of a particular language renders the generic algorithm unsuitable. The generic algorithms must of course be parameterized; the dynamically-loaded language information includes the code and/or data needed to do this. Our current prototype provides analysis and transformations for the lexical and syntactic properties of documents; others in the *Ensemble* group are implementing semantic [17] and presentation [9] services.

An object-oriented paradigm provides the joint benefits of abstraction and encapsulation. This design has been applied to implementations of language-based transformations (for example, parsers [1]). Applying an object-oriented paradigm to language *description* is also useful, since each major language typically has a constellation of minor variants. Our design embraces both of these ideas by viewing languages themselves as run-time instances from a class hierarchy. This extension provides the ability to replace parts of a language (for example, its lexical format) by treating the language as a first-class object that obeys a narrow abstract interface. Table 1 describes the current interface required of language objects in *Ensemble*.

The run-time representation of a language involves both a representation of its properties (for example, the grammar of a document is used by almost every service in an SDE) and a collection of instances of analysis/transformation algorithms. These instances share their language-specific information as class data, while maintaining per-document information as instance data or storing it with the document itself, in the form of structure, multimedia content within leaves, and as cached semantic values.

In addition to treating a language as an active, parameterized collection of analyzers and databases at run-time, an object design provides writers of language descriptions with the ability to encapsulate and — most importantly — *reuse* fragments of those definitions. Grammatical composition as supported by the Mango parser generator [1] is an example of an existing technique for composing and reusing fragments of language descriptions. The *Ensemble* system provides part of the framework for this type of descriptional composition through its language model, although designing composition algorithms has not been a major focus of our research.[7]

[7] The issue is actually more complex than this brief discussion suggests. There are a range of techniques from simple notational devices within a given language description formalism (e.g., `flex`'s support for named expressions) to reusable descriptive components such as Mango supports, to run-time support for layered or hierarchical descriptions [4] within or across languages. Our contribution is a language representation where sharing and specialization features are modelled directly upon the corresponding C++ language features, and the (apparently novel) realization that there are substantial benefits to a confluence of the descriptive and run-time hierarchies. A complete survey of the ideas surrounding the issue is beyond the scope of this paper.

In principle (and in practice, at the lowest levels of the C++ code), each analysis phase in each language is implemented by its own analysis object. However, it is neither necessary nor desirable to design and implement these algorithms by hand. Where possible, *generic* routines are used that compile high-level descriptions in regular expression notation, context-free grammars, or semantic description languages into C++ code. This code reuse is one of the principal benefits of the language hierarchy at run-time. Such code is then compiled and dynamically linked on demand (static linking is also possible, of course, and is used for a small number of built-in languages). In general, the loaded "code" is actually mostly data that parameterizes a generic run-time engine designed to implement incremental analysis efficiently.

The incremental algorithms originally designed to compute this type of language-specific information were intended primarily to improve performance over batch techniques, not to provide the best user and client services in an interactive environment. We have modified a collection of incremental algorithms to use the interface to history services described in this paper and to capitalize on the benefits of enhanced preservation detection. Section 4.2 investigates the design and implementation of an incremental lexical analysis algorithm; an incremental parsing algorithm is described in Section 4.3.

4.2 Incremental Lexical Analysis

Lexical analysis is based on regular expression pattern matching, as implemented by the Unix tools `lex` [15] and `flex` [20]. The description language permits various patterns to be described and associated with arbitrary user actions: in this setting, the actions are typically token constructors (special purpose actions to override the normal behavior may also be used; see Appendix B). Figure 7 illustrates lexical description for a simple example; the level of descriptive detail could be raised by interposing another layer of language specification between the language author and the low-level pattern/action formalism described here.

The lexical specification compiler `flex++` converts a document in this form into a collection of C++ routines and tables suitable for constructing lexemes from text in a left-to-right pass. It does this by associating the description's actions with a finite state machine representing the patterns. `flex++` actually generates a lexer class; each instance of this class represents single, complete lexer. The finite state machine (which represents the bulk of the code for a lexer) is shared among all the instances.

Using the output of this tool to construct an *incremental* lexer requires, among other things, that we be able to restart the analysis at an arbitrary point in the token stream. To do this, we need to restore the point in the finite state machine where the lexer began scanning the lexeme in question. This state is always encoded in some collection of variables within a lexer, but in the case of `flex++`-generated lexers it is especially easy: this is nothing more than the private (non-shared) fields of the lexer class.[8] The state of the lexer at the point where it began scanning a token's text is then compactly recorded within the token itself.

Making lexical analysis incremental also requires additional steps in the run-time application of the analysis and transformation. These include

- Locating areas that require re-analysis
- Conditioning updates to the lexical stream
- Starting lexical analysis at each changed region
- Terminating lexical analysis when a region is consistent

We will describe each in turn.

The first problem is to discover the regions in need of re-analysis after user or programmatic updates have been applied that make the document's textual and structural representations inconsistent. This means finding the beginning of each "stretch" (contiguous region of tokens in the document) that might require

[8] Some of these variables are merely used for interprocedural data passing; only those fields whose values are persistent between invocations of the lexing routine need be preserved.

Name definitions.

```
%%
    integer     [1-9][0-9]* | 0
    name        [_a-zA-Z][_a-zA-Z0-9]*
    whitespace  [ \b\t\n]
    bool        [tT]rue | [fF]alse
```

Special directives. The definition of an exclusive lexing state named `in_comment` *is shown.*

```
%x in_comment
```

Pattern/action pairs.

```
%%
{name}            MK_TOKEN(ID);
{bool}            MK_TOKEN(BOOL_CONST);
{integer}         MK_TOKEN(INT_CONST);
if                MK_TOKEN(KEYWD_IF);
else              MK_TOKEN(KEYWD_ELSE);
==                MK_TOKEN(OP_EQ);
!=                MK_TOKEN(OP_NEQ);
```

These commands discard whitespace from document. In this form,whitespace will be managed implicitly by the environment's editing and presentation services. States prefaced with `in_comment` *represent a separate~FSM.*

```
"/*"              BEGIN(in_comment);
<in_comment>.     /* ignore */
<in_comment>\n    /* ignore */
<in_comment>"*/"  BEGIN(0);
{whitespace}      /* ignore */
```

Figure 7: A sample lexical description. The formalism is derived from the description language used by `flex` and `flex++`. All the extensions provided by that program, including the use of arbitrary C++ code within actions, are available. The only restriction is inter-pattern dependencies (i.e., no global variables or REJECT directives).

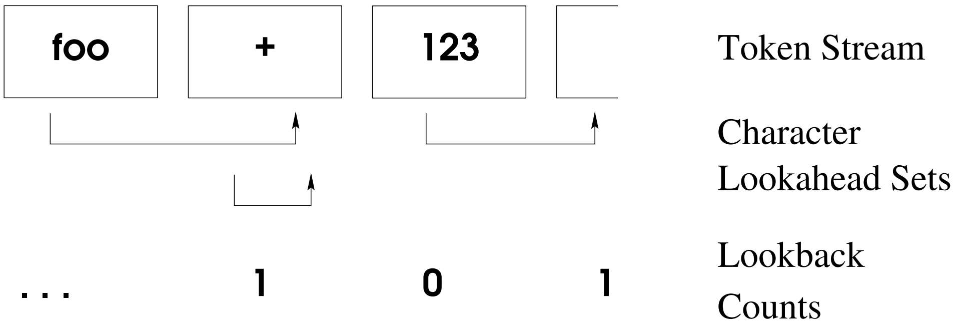

Figure 8: Computing lookahead sets and lookback counts in the lexical analysis routines. The identifier "foo" and integer constant '123' require a single character beyond their own text. The operator '+' is determined solely by its lexeme. The lookback counts invert the lookahead mapping and simultaneously shift units from characters to tokens.

updating. The entire set of tokens updated by the analysis is called the *update set*; the "leaders", those tokens that start each contiguous run, are referred to as the *root set*.

To compute the root set, we start with a subset of it: those tokens that received direct or indirect modifications since the previous application of the lexical analysis/transformation. The author and number of the intervening changes are immaterial. This subset is referred to as the *edited set*, and consists of tokens from three situations. First and most obvious are tokens that received textual modifications since the previous analyzed version. Second, deletions of one or more tokens that ended exactly at a token boundary must include the token following the deleted range in the edited set (otherwise that edit, and the tokens affected by it, may go unnoticed). Finally, it is possible for structural edits to affect tokens even though they do not alter the text within them. When a structural update has been performed, two token locations (which may specify the same token) must be included in the edit set: the first token in the yield of the moved subtree, and the first token following the yield of that subtree. (The deletion rule is computationally a special case of this rule, though it usually arises from a conceptually distinct class of edit operations.)

Once the edit set has been constructed, it is expanded to include all the tokens whose content might have depended on one or more items in that set. One token depends on another when its *lookahead set* — the set of characters examined by the lexical state machine while constructing the token — includes one or more characters from the second token. Figure 8 demonstrates this relationship graphically. In the absence of global data (which is disallowed in lexical pattern rules), lookahead sets completely characterize inter-token dependencies.

Lookahead sets can be extracted trivially from the lexing machine. In order to use them, however, we need to invert the information, turning it into *lookback counts*. These counts store, in each token, the number of tokens preceding it that must be examined if its content changes. Lookback counts are coarser than the lookahead sets, since all the characters within a token are merged into an equivalence class with respect to the dependency relation. This dramatically lowers the space requirement for incrementality without degrading the time or accuracy of the incremental lexer.

When this phase completes, all necessarily starting points for incremental lexing have been discovered. The next step is to iterate over the root set, in left-to-right order, restarting lexical analysis at each element of that set. The expectation (born out in practice) is that an area of new, modified, or deleted material can be rapidly stitched back into the surrounding lexical stream. This is due in large part to the fact that lookback counts are typically 1 for most programming language tokens.

Restarting the lexer consists of copying the saved state from the first token of the next region to be updated into the lexer's internal fields. The normal lexing routines are used to sequence along new or updated material. Eventually the lexer will return to existing tokens from the previous version, and will attempt to stop as rapidly as possible. The stopping test is simple: the lexer must be at a token boundary, and its

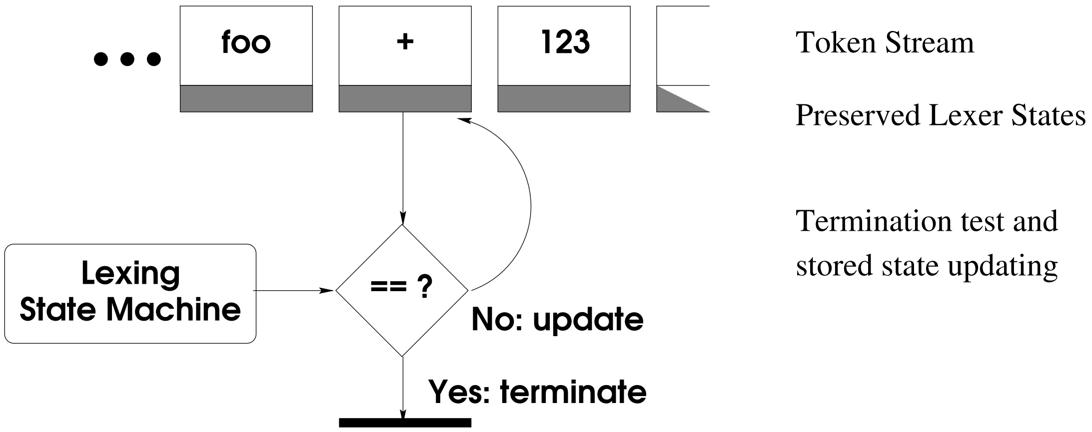

Figure 9: Determining stopping conditions during incremental lexical analysis. When the token type and lexical state match the previous contents, the machine can skip ahead to the next area requiring an update.

internal state must match that of the next token. When this test passes, the lexer can stop examining the current region and move on to the next one. Figure 9 illustrates this sequence of events.

4.2.1 History Services and Analysis

One issue ignored in the previous discussion of incrementalizing lexical analysis is the problem of simultaneously reading and writing a single data structure. Determining what has changed since the previous analysis requires that a record be kept of the token stream as it existed then. Constructing the update set requires examining the previous version of the token stream (its content immediately prior to the start of the analysis operation). Finally, any new tokens (or deletions of old tokens) that occur must be made to the new version, the one that will represent the result of the analysis transformation itself.

Any implementation of incremental lexical analysis must interact with these three versions of the document, although implementations might differ greatly in how the versions are represented and accessed. Our original implementation, which preceded the version system described here, required dedicated edit tracking to maintain the difference between the current and previously analyzed versions of the document. It also required redundant structural representations to avoid the problem of making updates to the lexical stream while reading different portions of it. This was one of the prime motivations for the design of the history mechanisms: they eliminate the need for every analysis algorithm to maintain such information privately, while providing it in a more general and efficient form for *all* clients. In its current form, the incremental lexer simply reads the previous version of the document while constructing its update set, writes to the current version under construction, and uses the change reporting mechanism to determine the contents of the edit set. The version system maintains consistent views of the different versions while sharing as much as possible between them, freeing the lexer from handling that task. This separation pays off handsomely: the new version of the lexer is less than half the code of the previous implementation.

4.2.2 Token Reuse and Change Reporting

Incrementalizing lexical analysis improves performance by trading an incremental update that is roughly the size of the changed material (though slightly slower to process on a per-token basis) for a batch update that is the size of the entire token stream. Since the normal case is that only a tiny fraction of the entire stream is modified, this is a highly effective tradeoff.

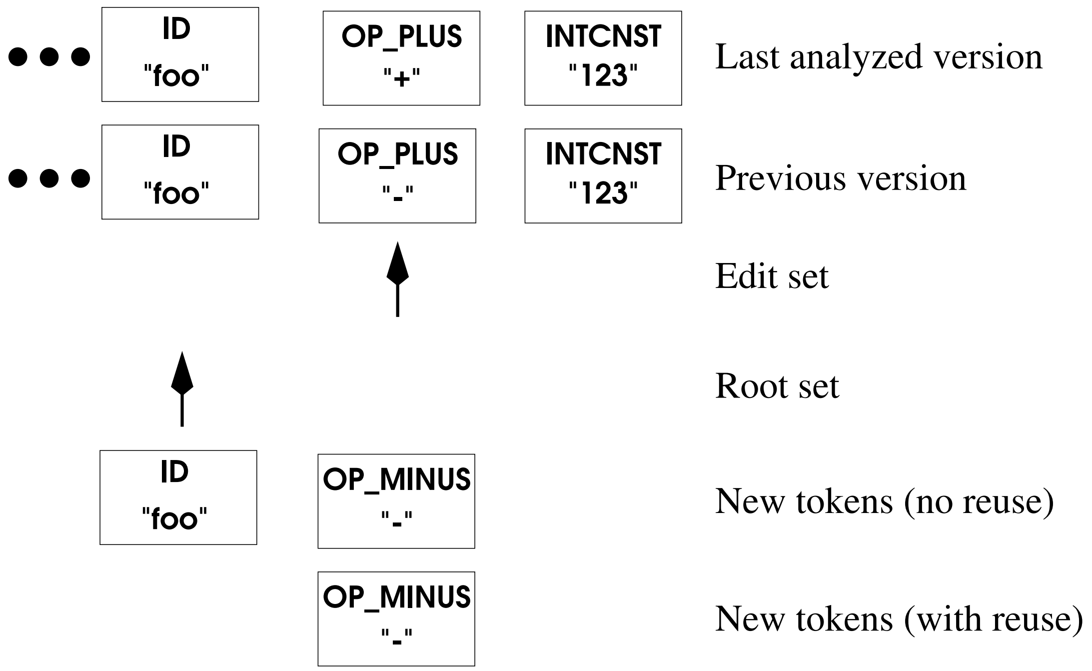

Figure 10: Applying reuse heuristics to eliminate the spurious recreation of tokens.

The second goal of incrementality is to create a system where the user's context[9] is preserved across editing operations. Batch operations, in addition to being slow, essentially destroy and re-create data structures "from scratch". It is difficult to restore explicit or implicit annotations when this style of update is used. Incremental solutions, by their very nature, preserve the bulk of a data structure across updates to it, making relationship between two (or more) versions much easier to decipher. Exogenous information associated with those data structures is then much easier to transfer to the new version: in many cases, trivially so because it exists as attributes of retained sub-components.

If incrementality is viewed from this frame of mind — that its goal is to produce a small and semantically meaningful delta between two versions of a data structure — then it becomes useful to further exploit the performance tradeoff. By doing slightly more analysis than absolutely necessary, it is possible for the analysis algorithms to improve the accuracy of their updates. This general desire takes on different forms in different analyses. In the lexer it means reusing existing tokens, instead of creating new ones, when it is intuitively plausible that that was the user's intent. This step alone will not significantly improve information preservation, but it prevents the "snowball effect": having minor corruption in sequentially applied transformations produce larger and larger losses.

Figure 10 illustrates a simple case: changing an operator from + to -. Because the previous token's lookahead overlapped the operator character, the lexer must re-analyze it when the operator changes (to see why, imagine that the operator's character had been changed to a letter instead of another operator). However, the update is superfluous; the result is exactly the same as before. While this example re-creates only a single token, the case is a common one that occurs at nearly every root (restart) point.

The reuse heuristic employed by the lexer is straightforward: it compares a prefix of the old and new token streams at each restart point. If they overlap, and the types of the tokens match, then the old tokens are reused, along with any textual modifications necessary to make them consistent with the current document text. This handles cases of deleting and re-writing text, modifying the spelling of identifiers, and the unnecessary restart problem described above, as well as several other common cases.

[9]The term 'user' is meant to suggest any entity editing and observing the document.

4.3 Incremental Parsing

The generic incremental parsing algorithm is a split-and-recombine technique based on ideas from Jalili and Gallier [12] and Ghezzi and Mandrioli [7], and modified for better user interaction [22]. Rather than parsing only terminal symbols, the parser accepts any sentential form: its input may include a mixture of terminals and nonterminals. When an entire subtree is unchanged and makes sense syntactically in the new version of the tree, the parser is free to retain it; otherwise the root of the subtree is discarded and the process continues with the former root's first child.

The lexical and syntactic analyses are performed simultaneously in our system, rather than as two separate passes (this is needed for languages such as C, Haskell, and Icon that have circular dependencies among the analysis phases). The analysis consists of the following high-level sequence, which takes place over each (monolingual) tree within a compound document:

- Locate, using a depth-first walk of the document as it existed immediately prior to the start of re-analysis, all the (possibly) changed regions. The lexer and parser must cooperate in determining the possibly affected regions based on the areas that the history mechanism indicates have changed since the previous analysis.
- Walk the document tree again, using the incremental lexer and previous document contents to generate the input to the incremental parser.
- Make a final pass across the new version of the document to update any dynamic dependency information and perform other post-processing, such as re-balancing repetitive structures to prevent performance degradation. (This may actually take the form of several passes over the changed portions of the document.) This phase can assume that the version of the tree it reads is intact and structurally well-formed.[10] The incremental lexer will update its lookback counts at this time.

Viewing this sequence, it is clear why the creation of a separate, fully-realized history mechanism resulted in the simplification of the previous design. Without its assistance, the lexer would attempt to use tree structure being consumed by the parser to locate its text! The history mechanisms are used in locating changes since the previous analysis, maintaining the three separate versions needed by the above steps, and registering the updates performed during the re-analysis itself.

As with *Ensemble*'s generic incremental lexer, the generic parser is based on an existing tool (`bison`), augmented with a more complex run-time mechanism. The incremental parser is parameterized by a parse table, a C++ subclass whose virtual functions implement a standard interface specifying the actions to take for a (state, lookahead) pair. Subtrees that have not been altered since the previous parse are moved onto the parse stack if the parse table indicates that they are syntactically valid at that point in the parse; otherwise, they are broken down into their components. If an error is detected when considering a terminal symbol, the parse is truly invalid and error recovery takes place. Errors are handled by *encapsulation*: the damaged area is discovered and marked, and the parse continues beyond it. Nonterminal symbols within the damaged area that are still discernible may be examined via a context-free *trial parse*. A complete discussion of error handling is beyond the scope of this paper, but its design elements—history-sensitive analysis and encapsulation instead of recovery—illustrate important ways in which existing batch algorithms need to be tailored to operate effectively in an interactive setting.

In the conventional approach to incremental parsing, nodes between changed sites and the root are always regenerated: every parser reduction creates a new node, even when it is simply restoring existing productions. Creating new copies of these nodes effectively renames them, discarding any cached annotations they might possess. Lost annotations can include the data structures of other analysis algorithms, which are then forced to redo (possibly time-consuming) computations. More important, however, is that this behavior always renames those nodes most likely to contain user annotations that require persistence: functions, classes,

[10] In the presence of errors, this means that they are duly marked, with any preservable substructures occurring as children of special error nodes.

Multiplication has highest priority and is left-associative.
```
%left *
```
Addition and subtraction are left associative and have the same priority.
```
%left + -

expr : expr '*' expr
     | expr '+' expr
     | expr '-' expr
     | '(' expr ')'
     | INTCONST
     | ID
     ;
```

Figure 11: Sample grammar. This grammar governs the subtrees used by the examples in this section and Appendix C. It is a simple subset of an expression grammar written using `bison`'s notation and operator specification language.

modules, or other syntactic elements that occur high in the document structure (since the set of changed material is essentially an implicit annotation, that, too, is rendered less informative by re-creating paths).

As with the lexer, the implementation of the parser performs additional work to attempt preservation. By detecting when a reduction indicates new material and when it is simply validating an existing region of the document, the nodes on the path between the root and an updated component can be preserved. Figure 14 shows the heuristics that the parsing algorithm applies at reduction time. These heuristics are quite general: their only dependence upon the parsing algorithm is that it have a recognizable point when a reduction is finalized. Common updates, such as editing the spelling of an identifier, are handled so successfully that the parser makes no modifications to the document whatsoever: subsequent analyses will detect only a change in the token's spelling. Figure 12 illustrates this idea graphically: neither lexing nor parsing attempt node reuse in the subtree shown there. (Figure 11 contains the grammar for this example). In Figure 13, the reuse heuristics described here have been applied to the same example, resulting in fewer updates and more constrained path markings.

Appendix C contains additional information on using `bison` to construct incremental parsers.

5 Implementation Status

The versioning and incremental language analysis algorithms described in this paper were prototyped within the context of *Ensemble*, a system that integrates document processing and software development environments [8]. *Ensemble* places emphasis on both conventional editing operations (textual and structural changes supplied explicitly by the user) and automated derivation of structural and semantic information. The document model applies equally well to structured natural language documents and software documents, and all the components of the system, including the presentation mechanisms, work from this structure-based perspective. All *Ensemble* documents, even programs, may contain multimedia components (for example, graphic, audio, or video commentary can be incorporated in code or design documents). All media use the same interface to the history mechanisms.

Ensemble as a software entity is a single-process Unix program whose kernel is written in C++. It has been ported to both the SunOS and HPUX operating systems. *Ensemble* uses X windows and its look-and-feel is that of the Tk widget set[19]. A complete extension language, based on Xlisp, is available[3].

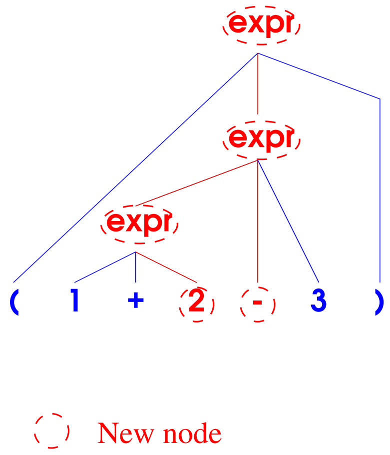

Figure 12: Subtree illustrating the no-reuse case. Without the presence of heuristics in the incremental lexing and parsing algorithms, substantial portions of the structure can be destroyed and recreated with a structure isomorphic to the original. The path is identical to the set of new nodes.

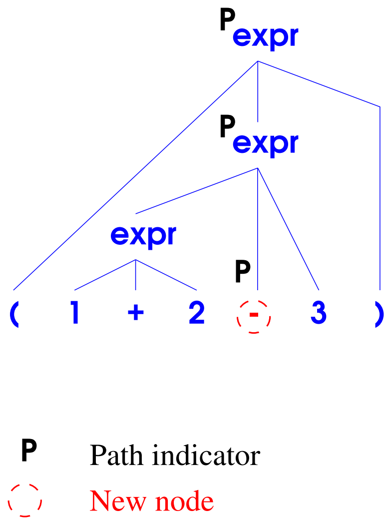

Figure 13: This is the same subtree shown in Figure 12, but with reuse heuristics applied. Note that the only new node results from new text, so that the updated region corresponds better to the user's intuition of his edit's effect. The path has been reduced to focus solely on the new token.

The central check: a reduction is creating `new_node` *to hold* `kids`*...*
could an existing node be reused instead?

```
reuse_check (new_node, kids) {
  foreach kid in kids
    var old_parent = parent of kid in last analyzed version;
    if old_parent's type == new_node's type and can_grab_parent(old_parent, kid))
      return reuse_isomorphic_structure(attach_kids(old_parent, kids));
    else
      return attach_kid(new_node, kids);
}
```

Prioritize the opportunities for reuse: give children that remained with their parent preference in reusing it. Otherwise, use FCFS.

```
bool can_grab_parent (parent, kid) {
  if parent missing or already reused then return false;
  if parent occupies its same location from the last analyzed version then
    foreach child of parent in the last analyzed version, except for kid,
      if child is still there now return false;  Yield the reuse opportunity.
  return true;  We can reuse it!
}
```

Can any of the old subtree rooted at 'node' be restored?

```
reuse_isomorphic_structure (node) {
  foreach kid in node's children
    var old_kid = corresponding child in last analyzed version
    if kid != old_kid && kid's type == old_kid's type &&
       kid was created in this version &&
       old_kid has not been reused then {
      replace kid with old_kid;
      reuse_isomorphic_structure(old_kid);
    }
}
```

Figure 14: Reuse heuristics. The parser applies these techniques at reduction time, when a new node would normally be generated to hold the subtrees being popped off the parse stack. First, a node with more than one child attempts to reuse the parent production if its type remains valid. If that succeeds, a pattern matching is attempted to restore any child productions of the old node that would be valid (this finds chain and epsilon production reuse candidates missed by the first heuristic). Successful matches of this sort apply the same check recursively.

Ensemble is a multilingual ISDE: its kernel contains the generic language analysis services described here and in [17]. The individual specification files for each language are combined into a single object file suitable for either static or dynamic loading. Presentation specifications are also dynamically loaded, as are customized editing and user interface controls. The latter are loaded directly by the extension language as Xlisp source files. Current language support includes Fortran, SQL, and *Ensemble*'s own specification languages. Our current research includes support for circular language definitions required to handle C and C++.

Documents are re-analyzed at the user's request (typically rather frequently), and involve multiple edit sites with multiple types of edits (e.g., both textual and structural edits in the same analysis update). *Ensemble*'s language analysis algorithms have been modified as described in this paper to use the versioning system, simplifying their implementation considerably over earlier designs [2]. Heuristic designed to improve the preservation of document components is an integral part of all the analysis algorithms.

The version and history services described here are fully implemented in the current prototype, and are fast enough to support interactive response times. The global version tree of each document is used as the basis for an EMACS-like undo system: the undo log is a description of the route traversed by the user over the global version tree. In addition, we are currently implementing a Tk-based direct access metaphor as a tree viewer/editor on the global version tree.

6 Related Work

Previous work in our group has used a more limited text-oriented undo model [2]. While simpler to implement, this approach does not support a multimedia document model nor does it support context-sensitive analysis.

Many proposals for the handling of non-traditional data models and version/revision support have been discussed in the database community [13]. Our work bears some similarity to the idea of main memory databases and the use of recovery logs for them [11].

Magnusson, et.al. have described the sharing and collaborative framework of the Mjølner project [14, 18]. Our concept of integrating analysis-based transformations with history services is compatible with such an approach. Our analyses are designed to maximize component (and information) preservation within structured objects, rather than duplicating paths as the Mjølner system does.

Document processing systems have explored the role of structure in the context of natural language rather than programming language documents [21]. Our approach provides interoperability with these designs, and also suggests a framework in which automated transformations (e.g., page breaking) can be considered as first-class edit operations similar to the incremental analysis performed in formal languages.

Fraser and Myers [6] present a single-level storage model for integrating an editor's undo/redo facility with a source code control system. They use a text-based model (developed from `vi`) where revisions are described in line-oriented terms. Hierarchical document models are a more natural representation for structural changes and support the specialization of media-specific updates.

7 Conclusion

We have described a technique for integrating version management with the incremental analysis of formal languages. The approach is a natural generalization of history-based services. Redesigning incremental algorithms to work with this versioning system exposes their fundamental dependence on multiple versions and the relationship between reuse detection and the preservation of document information. The versioning system has been designed to support multimedia objects, multiple edit sites per operation, and the efficient traversal of structured components.

Acknowledgments

We are grateful to the *Ensemble* developers for helping to create an appropriate testbed for this project. Special thanks are due to Vance Maverick, Ethan Munson, Bill Maddox, and Todd Feldman for their comments on drafts of this paper.

References

[1] Ole Agesen. Mango: A parser generator for Self. Technical Report SMLI TR–94–27, Sun Microsystems Laboratories, Inc., 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):95–127, January 1992.

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

[4] R. Farrow, T. J. Marlowe, and D. M. Yellin. Composable attribute grammars: Support for modularity in translator design and implementation. In *ACM Nineteenth Symposium on the Principles of Programming Languages*, pages 223–234, 1992.

[5] A. Fekete, N. Lynch, M. Merritt, and W. Weihl. Nested transactions and read/write locking. In *Proc. Sixth ACM Symp. on Principles of Database Systems*, pages 97–111, 1987.

[6] Christopher W. Fraser and Eugene W. Myers. An editor for revision control. *Software—Practice and Experience*, 9(2):277–295, 1987.

[7] Carlo Ghezzi and Dino Mandrioli. Augmenting parsers to support incrementality. *Journal of the ACM*, 27(3):564–573, July 1980.

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

[9] 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.

[10] Görel Hedin. *Incremental Semantic Analysis*. PhD thesis, Department of Computer Science, Lund University, March 1992.

[11] H. V. Jagadish, Avi Silberschatz, and S. Sudarshan. Recovering from main-memory lapses. In *Proceedings of the 19th VLDB Conference*, pages 391–404, Dublin, Ireland, 1993.

[12] Fahimeh Jalili and Jean H. Gallier. Building friendly parsers. In *Conference Record of the Ninth ACM Symposium on Principles of Programming Languages*, pages 196–206, 1982.

[13] Randy H. Katz. Toward a unified framework for version modeling in engineering databases. *ACM Computing Surveys*, 22(4):275–408, 1990.

[14] J. L. Knudsen, M. Löfgren, O. L. Madsen, and B. Magnusson. *Object-Oriented Environments — The Mjølner Approach*. Prentice-Hall, 1993.

[15] M. E. Lesk and E. Schmidt. LEX—a lexical analyzer generator. In *Unix Programmer's Manual*. Bell Telephone Laboratories, 7th edition, January 1979.

[16] Ernst Lippe and Norbert van Oosterom. Operation-based merging. *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, 17(5):78–87, December 1992.

[17] William Maddox. Colander2: A reference manual. Technical Report (internal), Computer Science Division—EECS, University of California, Berkeley, 1993.

[18] 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.

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

[20] V. Paxson. `flex` man pages. In flex-2.5.2.tar.z, Free Software Foundation, November 1995.

[21] 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.

[22] 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.

[23] Jim Welsh and Jun Han. Software documents: Concepts and tools. *Software—Concepts and Tools*, 15:12–25, 1994.

A Projection Algorithm

Distributing change information spatially requires the ability to convert a time, denoted by a node in the global version tree, into the state of any given document component. This is accomplished by the *projection algorithm*, which maps the global version node into a corresponding local one. The algorithm works by finding an association between the global and local trees, then moving simultaneously in each to reach the desired version. The starting point in the local tree is its current version; the starting point in the global tree is the desired version. (In what follows, we will use denote the inverse projection as `project`$^{-1}$. The mapping in this direction is explicitly maintained: each node in a local version tree identifies the global version in which the local change occurred.)

Let Tg be the target global version.
Let Cl be the current node in the local version tree

Phase 1: Find matching nodes in local and global trees.

```
Simultaneously traverse the paths in the global version tree from Tg
and project⁻¹(Cl) toward the root until they meet.  Let LCA denote
the meet point.  Maintain the equivalent node in the local tree (L)
during this upward traversal.

Let Pg denote the global path between LCA and Tg.
Let BadPathL denote the local path between project(LCA) == L and Cl.
```

Phase 2: Retrace global path's steps to return to the target.
Simultaneously track the (correct) local path each time it grows.

```
Traverse Pg, beginning at LCA.  For each node N of this path do {
  if L has no children return L.    Local tree "bottomed out".
  if L has only one child {
  The child either extends the local path or terminates the downward search.
    continue along Pg until either project⁻¹(L) or Tg is reached
    if Tg is reached, return L.
    else L ← its child; continue outer loop.
  }
  At most one of L's children will be added to the local path.
  Determine where the (child[i], root) path intersects Pg.
  (The search must be performed in parallel to achieve the stated time bounds.)
  Forall children of L except the one on BadPathL do in parallel
    Walk upwards in global version tree from project⁻¹(child[i]) until
      one reaches N or all have reached Pg above N.
  if child[i] reached N then L ← child[i]; continue outer loop.
  else return L.    Can't extend local path; L must be the answer.
}
return L.    Global path reached the target, so L is the correct projection.
```

B Using Flex Under Ensemble

`flex`'s input format is a relatively low-level mechanism for specifying regular expression rules and associated patterns to fire when the expressions are discovered in the source. It was not conceived of as a language specification formalism, and permits actions and descriptions that cannot be satisfied by an incremental analysis system or do not make sense in the context of an interactive software engineering environment.

This section is not a user's guide, but attempts to give a brief overview of the way in which `flex` can be used, in conjunction with the run-time mechanisms described in this report, to produce an a lexical language description that admits to incremental analysis.

The pattern description language is supported in its entirety. The rules, however, are more constrained. `flex` supports arbitrary C (or C++) code in the pattern rules; we restrict that to code which does not involve information passing between rules. The description is not checked to ensure that this restriction is met, but there is rarely a need to resort to code in any case. The formalism supported by `flex` is normally powerful enough to describe the lexical structure of a programming, scripting, or formal language. Rules normally consist of start state controls (which are built into `flex`) and *token constructors*: these are the basis by which text is associated with a C++ class, to form a leaf of the document tree.

`flex` contains specialized support for efficient buffering of its input, as well as an array of buffer management functions, available to the description writer, for managing multiple input sources. None of this is relevant to the normal case in an ISDE, where the input is derived from the content of the (textual) leaves of a document.[11] Likewise, some of the special commands are simply not useful in this context. These include `REJECT` and other options that conflict with the idea of a partitioning of the input stream into tokens.

The issue of whitespace deserves special attention. In a model where whitespace is implicit (and thus managed by the system), there is no need for explicit whitespace characters to aid in presentation. In this case, only a limited form of whitespace token will exist, and then only to assist the system in a proper partitioning of the text into tokens; the user will not be able to affect the text of whitespace tokens, since they will not be a part of the editable representation on-screen.

The second model of whitespace is to preserve its historic use as a presentation directive, whereby the author can control layout via a monospaced font and explicit whitespace characters. Stream-style comments are also typically a part of this model. In this case, it is important that whitespace be fully integrated into the document tree. To accomplish this, whitespace tokens are associated with the token that *precedes* them (a special sentinel token, BOS, marks the beginning of the document and thus such a token always exists). Since there may be multiple, contiguous whitespace tokens, they are arranged in a binary tree; at most one attachment need be made to any real leaf. This is done by forming an extra nonterminal — essentially a special production of the form `leaf -> real_leaf whitespace_tree`. Stream-style comments are handled in the same manner. To indicate to the system which tokens are to be processed in this manner, a category is associated with each token constructor:

Text Identifiers, numeric constants, strings, and other tokens whose spelling can change from edit to edit.

Constant Text Operators and keywords, whose spelling is implied by their type.

Comment Stream-style comments that do not appear in the grammar *per se*; they are processed as described above, but annotated differently from pure whitespace to assist the presentation engine if it desires to distinguish them.

Whitespace Whitespace (spaces, tabs, newlines) that is processed and represented as described above.

C Using Bison Under Ensemble

Although it is built from a more complex automata than lexing, parsers are somewhat easier to integrate into an incremental setting because their input language is more standardized and their dynamic dependencies are statically determined by the parser generator. The current implementation uses `bison` to specify grammars, and existing `bison` input files are mostly compatible (reduction time rules are simply ignored, as are type directives). Precedence and associativity operators are obeyed; see below for a discussion of how this impacts the design of the incremental parser's run-time system.

[11] It is relevant in the case where a document is being *imported* into the system from an ASCII source, in which case `flex` is used as a normal batch tool with its usual properties.

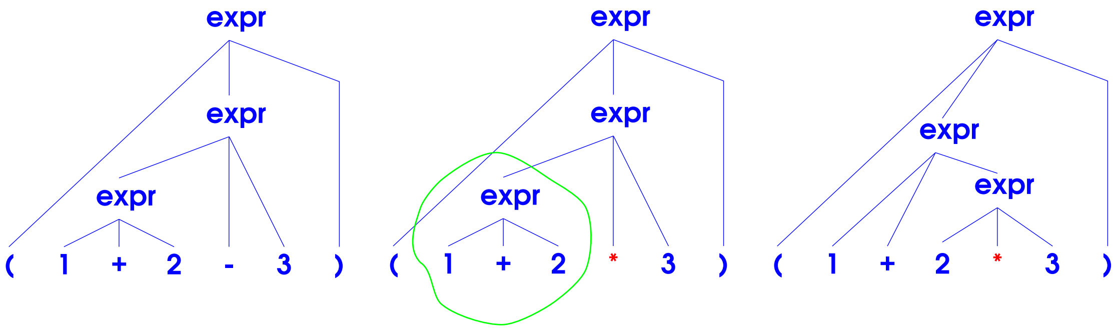

Figure 15: A sequence illustrating the need for checks outside of the grammar *per se*. In the first panel, the expression `1 + 2 - 3` has been modified to replace the `-` with a `*`. Without additional steps, the (incorrect) re-parsing is shown in the second panel: the leftmost operator's subtree has been left intact, violating the operator precedence rules. In the third panel, the correct structure is achieved by the additional validation described in the text. This example is based on the grammar from Figure 11.

Unlike `flex`, `bison` produces only data, not code, to parameterize the run-time system. These tables are used as is, augmented with an additional table to indicate which reduction to use when the next item is a nonterminal (in the batch case, only terminals appear in the input stream). The major change to `bison`, however, is its use as a *grammar generator* in addition to parser generation.

Normally, `bison` dumps a simple integer/string representation of the grammar when it creates a parsing implementation, mostly to assist the programmer in debugging his grammar. However, both descriptions and run-time services in an ISDE are conditioned upon the structure of the document, making the grammar a central component. Since *Ensemble* is an object-oriented system, individual nodes in the document correspond to productions in its grammar. As an implementation strategy, the grammar is a web of canonical node instances: to create nodes for use in a document, a canonical (grammar) node is asked to create a duplicate of itself. To make this approach work, a new back-end was added to `bison` that writes the grammar as a collection of C++ classes, one per production (or token). Instances of these nodes are typed, and the right-hand side of the grammar indicates the number and type of fields within the node corresponding to that production. (The Mjølner system [10] uses additional superclasses to group information about all the productions sharing a left-hand side; this information is simply made available as shared class variables in our implementation.)

`bison` contains a "mini-language" for expressing the precedence and associativity of operators. While this is not the most general technique known (it cannot represent all possible combinations of the pre-, in-, and post-fix operators), it is simple, in widespread use, and contributes to a vast reduction in the size of the concrete parse tree.[12] The current version of *Ensemble*'s incremental parser was designed to support this specification scheme. Doing so requires additional checks to be made at certain points in the analysis; the "border" between two subtrees must be verified if they were not adjacent in the previous version of the document. Moving, adding, or deleting subtrees can change the operators exposed along their edges, requiring a re-formulation of the parse tree in the new version. (That is, even though the two trees are syntactically correct, they do not obey the additional rules given to `bison` regarding operators. Figure 15 illustrates such a sequence.) To avoid this problem, the parser validates new combinations by breaking down the edges of the newly-adjacent subtrees to their tokens; this is just another use of the algorithm the parser normally employs to validate syntactic correctness.

Upward compatibility with existing `bison` input was an overriding goal in the choice of a formalism: not only do differences, however well designed, require modifications to the parser compiler, they also imply a tedious re-writing of existing inputs, which are numerous and available for most languages of academic and commercial interest. In order to accommodate additional information, a parallel specification file is used, keyed

[12] For a typical imperative language, this amounts to greater than a four-fold reduction in the average size of a concrete parse tree.

by productions and symbols in the associated grammar. Among other things, this additional specification format can used to control class generation, specify token classes (see Appendix B), and insert additional code into the generated grammar file. Another extension in progress is a list (repetition) specification, which indicates that a linearly recursive production in the grammar is to be represented as a balanced binary structure within documents constructed from that grammar. (For both lexical and syntactic descriptions, an integrated and tailored language specification formalism is eventually intended to replace direct writing in the low-level tool formalisms, which will then be treated merely as inter-tool protocols.)