History-Sensitive Error Recovery

Tim A. Wagner, Susan L. Graham

Abstract— **We present a novel approach to incremental recovery from lexical and syntactic errors in an interactive software development environment. Unlike existing techniques, we utilize the history of changes to the program to discover the natural correlation between user modifications and errors detected during incremental lexical and syntactic analysis. Our technique is *non-correcting*—the analysis refuses to incorporate invalid modifications, while still permitting correct changes to be applied. Errors are presented to the user simply by highlighting the invalid changes.**

The approach is automated—no user action is required to detect or recover from errors. Multiple textual and structural edits, arbitrary timing of incremental analysis, multiple errors per analysis, and nested errors are supported. History-based error recovery is language independent and is compatible with the best known methods for incremental lexing and parsing, adding neither time nor space overhead to those algorithms. Effective integration with the environment's history services ensures that other tools can efficiently discover regions of the program (un)affected by errors, and that any transformations of the program required to isolate or present errors are themselves efficiently reversible operations.

Keywords— **Error recovery, software development environments, incremental parsing, incremental lexing, development log, program presentation**

I. Introduction

SYNTACTIC error recovery in batch systems is essentially a solved problem, involving a heuristic computation based on the configuration of the parser when the error is detected [1]. The best methods known rely on the ability to delay actions or reproduce part of the parse on demand, so that a variety of repairs may be tried at locations other than the detection point [2], [3]. Since the recovery routine has no knowledge of the user's changes with respect to previous versions of the prgoram, it attempts to correlate the problem with the detection point by comparing the results of different repairs. When the error is significantly complex or distant from its detection point, a less informative 'second stage' recovery may be needed.

In an incremental, interactive software development environment (ISDE), errors can arise as they do in batch systems, since arbitrary modifications to the text and structure of the program are permitted. Errors introduced by changes will be discovered when the user next requests incremental analysis. Many types of problems can occur, including a variety of static semantic errors (type inconsistencies, missing definitions). However, errors associated with the lexical and context-free syntax play a special role in an ISDE: the structural representation is a core data structure, and language specifications typically do not prescribe the representation of erroneous programs.[1] Thus the system is faced not only with the task of effectively detecting and reporting any errors, as in a batch compiler, but also with integrating some representation of the problem into the persistent, structural representation of the program. No satisfactory approach to this problem has previously been available.

Several systems have tried to minimize or circumvent the difficulty of error handing in an ISDE by limiting the class of modifications available to the programmer [4]; in the extreme case, only structural operations that preserve all correctness properties are permitted [5]. Our approach is the other extreme: we place *no* restrictions on the editing model, allowing arbitrary textual and structural modifications and arbitrary timing of the analysis. Multiple errors, including nested errors, may exist simultaneously and do not preclude the incorporation of other modifications. Errors may persist indefinitely; the environment must tolerate the presence of any invalid or inconsistent material and continue to provide as much functionality as possible [6]. The goal of the environment is to isolate problematic regions, inform the user of their location, and provide assistance by explaining the reason these modifications could not be successfully adopted.

Our approach is fully automatic—no user intervention is required to detect errors, and the user is free to correct errors in any order, at any time. In contrast, several researchers have addressed error recovery in an ISDE by attempting to utilize the interactive nature of the environment [7], [8], [9]. Unfortunately, these approaches all demand direct user intervention at each error site and therefore impose an unnecessary serialization on the analysis and the user's actions. A few research and commercial environments support unattended incremental error recovery in the context of incremental parsing [10], [11], [12] but there has been little discussion or analysis of the technologies employed. Moreover, no existing systems make use of the vast amount of information available in the development log being maintained by the environment.

The central idea in our approach is to recognize that some user modifications introduce (locally) valid changes while others do not: modifications success-

T. Wagner is a graduate student in the Computer Science Division–EECS Department at the Univ. of California, Berkeley. Corresponding address: Tim A. Wagner, 573 Soda Hall #1776, Univ. of California, Berkeley, CA 94720-1776 USA.
S. Graham is a professor in the Computer Science Division–EECS Department at the Univ. of California, Berkeley.

This research was supported in part by Advanced Research Projects Agency grant MDA972-92-J-1028, and by NSF grant CDA-8722788. The content of this paper does not necessarily reflect the position or the policy of the U. S. Government.

[1] Static semantic errors, on the other hand, can be represented without leaving the framework of the attribute grammar or similar formalism.

fully incorporated into the structure and content of the program representation are retained, while invalid changes remain in their 'unanalyzed' form. This is a *non-correcting* strategy: unlike most automated recovery schemes, it does not attempt to guess the programmer's intention. The well-known drawbacks of correcting strategies are avoided: no conjectures are necessary, spurious repairs never arise, and no heuristic 'language tuning' is needed. The correctness of *any* repair we perform can be easily established, even in an incremental setting.

Non-correcting approaches [13], [14], [15] have not received much attention in batch compilers. Despite the theoretical advantages described above, the practical limitations imposed by a batch setting cause even the best of these approaches to be less useful than correcting methods. The most critical problems involve errors in bracketing syntax and the fact that the initial error typically obscures detection of subsequent problems, since the following text is often a valid substring in *some* sentence.[2] Figure 1 illustrates these deficiences.

Our recovery scheme overcomes these deficiences by using *historical information* [17]: the sequence by which the programmer arrived at the current state affects the treatment and reporting of errors. Changes recorded in the development log permit comparisons between the current and previous versions of the program, providing a guide for determining the source of a given problem *in terms of the user's own modifications*. This approach can discover the relationship between the point where an incorrect change was applied and the point where the error was finally detected even when they are far apart or separated by intervening errors.

Determining the relationship between changes to the program and subsequent errors is a novel and powerful tool for error handling. While the best known batch correcting recoveries handle the example in Figure 1 better than their non-correcting counterparts, they will typically fix the initial error by adding an extra closing brace at the end of the program. Our approach instead 'corrects' the problem by refusing to insert the extra opening brace—a better and more comprehensible response given the actual change made by the user.

The dependencies between program components induced by lexical and syntactic analysis methods provide a natural way to discover and limit the scope of a given error. This *isolation* process makes it possible to treat unrelated errors independently and allows correct modifications outside the isolated regions to be successfully incorporated. Isolation is computed incrementally, by comparing the current (partial) structure of the program to the previous structure stored in the development log.

Initial (correct) program
```
int f () {
  g(a + b);
  if (c == 3) c = 4;
  else c = 5;
}
```

Introducing three errors
```
int f () { {        Inserted extra opening brace
  g(a + b [);]      Deletion
  [if (c == 3) c = 4;]   Deletion
  else c = 5;
}
```

Result: only one error is detected
```
int f () { {
  g(a + b  ERROR
  else c = 5;
}
```

Fig. 1. An example where batch non-correcting techniques fail. The first error (extra opening brace) is hidden by subsequent problems. The second error, a deletion, is detected, but the non-correcting nature of the recovery precludes discovery of the third error, since what remains after the deletion is a valid substring. Our approach discovers all three errors *without* attempting to correct the program; the visual presentation would be similar to the second version above with the explanatory text removed.

Not all of the modifications within an isolated region are necessarily incorrect, and even a well-chosen isolation region can be very large. Thus some mechanism to detect legal modifications within an isolated region is needed. Two techniques are used: *retention of partially-analyzed regions*, which avoids discarding legal updates prior to the detection point, and *out-of-context analysis*, which applies incremental analysis techniques to modified subtrees within the isolated region. Together, these techniques typically allow legal modifications to be incorporated, even in close proximity to one or more errors.

This approach is language independent. The recovery is guided by existing mechanisms for lexical and syntactic analysis; the language designer is not required to provide additional specifications in order for error recovery and reporting to function.[3] The approach is compatible with existing approaches to incremental lexing [18], [19] and both deterministic [20] and non-deterministic [21] parsing.

Errors are represented in a simple fashion: the invalid modifications are simply maintained as unincorporated edits. This suggests a presentation of errors that is at once trivial and powerful: the unincorporated edits are visually distinguished to indicate the recent user changes responsible for the problem. The need to generate explanatory messages and associate them with locations in the program text is thus avoided. (For newly-inserted material, error messages can be assigned in the conventional manner). This representation also reuses existed mechanisms: the presence of errors imposes no

[2]Right-to-left substring parsing (*interval analysis* [13], [16]) has been proposed to further constrain the location of detected errors, but common mistakes can result in intervals so large that the user must still locate the problem manually. Interval analysis does not address the detection problems of batch non-correcting recoveries.

[3]Large insertions of new text, which require batch analysis and for which batch techniques are the only possible solution, may rely on language-specific information to tailor their recovery. Section VII-F discusses the application of batch approaches within a contiguous region of inserted text.

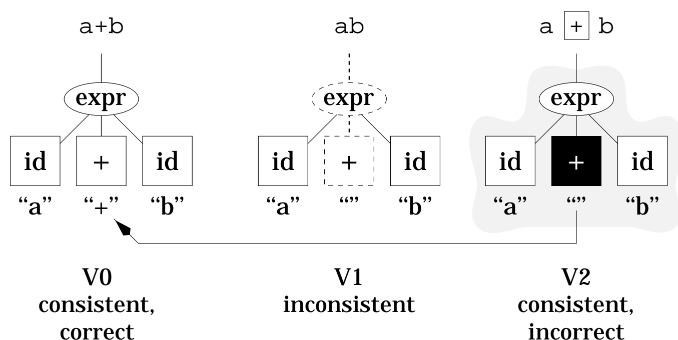

Fig. 2. The recovery and presentation of a simple error. In V1, the dashed lines indicate the path from the root to the site of the unincorporated modification. In V2, the isolated region is indicated by light shading; the token shown in black has been marked to indicate that it contains an invalid textual edit. The visual presentation corresponding to each version is shown above the subtree.

additional requirements for persistent storage, change reporting, re-analysis, or editing. Tools in the environment can locate errors efficiently, and can restrict their attention to the syntactically valid structure, since unincorporated modifications and isolated subtrees are clearly identified. Since both the representation and presentation of errors are integrated with the structure and content of the program, any transformations induced by error recovery are completely reversible [22], [23].

Figure 2 illustrates a simple example. In the initial version (V0), the program is in a consistent state. The user then modifies the program to create version V1. At this point, the structure of the program is no longer consistent with its textual content, due to the unincorporated deletion of the addition operator. In V2, the user requests that consistency be restored; incremental analysis detects the error at this time. The expression enclosing the deletion is then *isolated*, and the token containing the deletion is flagged as possessing a change that could not be successfully incorporated. The error 'message' is simply the difference in the content of the isolated region between V2 and V0 (shown as a box around the deleted operator).

The rest of this paper is organized as follows. We begin by discussing the program representation, history log, and edit/change-reporting model in Section II. In Section III we briefly review one method for incremental syntactic analysis, sentential-form parsing, to illustrate the relationship between the detection of an error and the recovery routines. Section IV describes the basic framework for our approach, including the representation of errors and algorithms for isolating and recording errors. In Section V we discuss techniques for incorporating additional (legal) modifications within an isolated region by retaining partially-analyzed results and by applying out-of-context analysis to modified, unanalyzed subtrees. Section VI considers a simple presentation scheme that combines analysis results, unincorporated material, and the contents of the distributed development log to display errors in an informative manner. Extensions to the basic framework are covered in Section VII, and Section VIII concludes the paper.

II. Editing Model and Change Reporting

This section reviews our representation of structured documents and a model for editing and transforming them. The object-based versioning services described here provide the incremental lexer and parser (and other tools in the environment) with the means both to locate and record modifications. The same interface used to undo textual and structural edits can be used to undo the effects of *any* transformation, including incremental lexing, parsing, and error recovery. The representation is based on self-versioning documents [23].

A. Representation

The algorithms described in this paper have been embedded in a C++ implementation of the *Ensemble* system developed at Berkeley. *Ensemble* is both a software development environment and a structured document processing system [24], [25]. Its role as a structured document system requires support for dynamic presentations, multimedia components in documents, and high-quality rendering. The need to support software necessitates a sophisticated treatment of structure: fast traversal methods, automated generation mixed with explicit (direct) editing of both structure *and* text, and support for complex incremental transformations [26].

Although the *Ensemble* document model supports attributed graphs, in this discussion we will restrict our attention to tree-structured documents, focusing primarily on the text and structure associated with programs. Each tree is associated with a language; in the case of programs, this is a run-time representation of the programming language, containing the grammar and an appropriate specialization of the associated analysis/transformation tools. The tree's structure corresponds to the concrete or abstract syntax of the programming language; the leaves represent tokens. (Any explicit whitespace tokens are integrated with the normal structure.) Tree nodes are instances of strongly-typed classes representing productions in the grammar. These classes are automatically generated when the language description (including the grammar) is processed off-line. Semantic analysis and other tools extend the base class for each production to add their own attributes as slots [27]. Information can also be associated with nodes via annotations or as references from a program database.

B. Editing Model

We permit an unrestricted editing model: the user can edit any component, in any presentation, at any time. These changes may introduce temporary inconsistency in the program representation. The frequency and timing of consistency restoration is a policy decision: in our current prototype, incremental lexing, parsing,

and semantic analysis are performed when requested by the user, which is usually quite frequently but not after every keystroke. Between incremental analyses, the user can perform an unlimited number of mixed textual and structural[4] edits, in any order, at any point in the program. Incremental performance is not adversely affected by the location of the edit site(s)—changes to the beginning, middle, or end of the program are integrated equally quickly.

Our approach handles all transformations, both user changes and those applied by tools such as incremental parsing, in a uniform fashion. Among other benefits, this allows the user to use existing undo/redo commands to return to any state of the program. This uniform treatment is critical to providing a rational user interface, and requires no additional effort in the implementation of the incremental tools—their effects are captured in the same development history that records all program modifications.

C. Program Versions and Change Reporting

An ISDE includes a variety of tools for analyzing and transforming programs. Some tools must be applied in a strict order—for example, semantic analysis cannot be applied until incremental lexing and parsing have restored consistency between the text and structure of a program component. Simple editing operations can also be viewed as transformations, a perspective that is particularly useful when discussing *change reporting*, the means by which tools convey to each other, via the history services, which portion of a program has been changed.

Modifications to the program are initially applied by the user, either directly or through the actions of one or more tools. The completion of a logical sequence of actions is indicated by a *commit* step; once committed, the contents of a version are read-only and are treated as a single, atomic action when changing versions. All versions are named, allowing tools to readily identify any accessible state of the program.

History (versioning) services provide the correspondence between names of versions and values. Their primary responsibility is to maintain the development log, retaining access to 'old' information. Updates to persistent information are routed through the history service, with the current value of versioned data always cached for optimum performance.

The history services also provide a uniform way for tools to locate modifications efficiently. This service is fully general, in that any tool can examine the regions altered between any two versions. Changes can be examined not just at the level of the entire program, but also in a distributed fashion for every node and subtree.

[4]There are no restrictions on structural updates save that a node's type remain fixed and that the resulting structure remain a tree. Structural changes not compatible with the grammar *are* permitted; special error nodes are introduced as necessary to accommodate such changes. (See Section VII-A.) Textual modifications are represented as local changes to the terminal symbol containing the edit point.

`bool has_changes([local|nested])`
`bool has_changes(version_id, [local|nested])`
These routines permit clients to discover changes to a single node or to traverse an entire (sub)tree, visiting only the changed areas. When no version is named, the query refers to the current version. The optional argument restricts the query to local or nested changes only.

`node child(i)`
`node child(i, version_id)`
These methods return the ith child. With a single argument, the current (cached) version is used. Similar pairs of methods exist for each versioned attribute of the node: parent link, versioned semantic data, etc.

`void set_child(node, i)`
Sets the ith child to `node`. Because the children are versioned, this method automatically records the change with the history log. Similar methods exist to update each versioned slot.

`void discard(and_nested?)`
Discards any uncommitted modifications to either this node alone or in the entire subtree rooted by it when `and_nested?` is true.

Fig. 3. Summary of node-level interface used by incremental analyses. Each node maintains its own version history, and is capable of reporting on both local changes and 'nested' changes—modifications within the subtree rooted at the node. The `version_id` arguments refer to the document as a whole; they are efficiently translated into names for values in the local history of each versioned object [23].

This generality is achieved by having each node maintain its own edit history [22], [23].

Change reporting is the protocol by which tools discover the modifications of interest to them. Change reporting is mediated by the history service; tools record changes as a side effect of transforming the program and discover changes when they perform an analysis. The history service provides two boolean attributes for each node to distinguish between local and nested changes. *Local changes* are modifications that have been applied directly to a node. For terminal symbols, a local change is usually caused by an operation on the external representation of the symbol. (In the case of programs, local changes usually indicate a textual edit.) Structural editing normally causes local changes to internal nodes. *Nested changes* indicate paths to altered regions of the tree. A node possesses this attribute if and only if it lies on the path between the root and at least one locally-modified node other than itself. Local changes are simply a derived view on the local history log, but nested change attributes must be incrementally computed as synthesized attributes (and must themselves be versioned). Figure 3 summarizes the node-level history interface needed by incremental analysis and error recovery.

Incremental analysis involves three distinct versions of the program:

Reference: A version of the program that represents a consistent state. Any version that concluded with an analysis operation may be used; our prototype selects the most recent consistent version as the reference for the subsequent analysis. Exception: an initial analysis of a newly-entered program has no reference version, since it represents a batch scenario.

Previous: The state of the program immediately prior

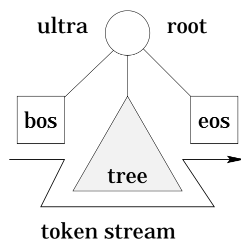

Fig. 4. The relationship between the three permanent 'sentinel' nodes and the parse tree structure. Two permanent tokens bracket the terminal yield of the parse tree, while a third sentinel points to both of these tokens as well as the (current) root of the parse tree. The sentinel nodes do *not* change from one version to the next.

to the start of re-analysis. This is the version read by the lexer and parser to generate their input. (The modifications accrued between the reference and previous versions determine the tokens and subtrees available for potential reuse.)

Current: The version being written (constructed) by the lexer/parser.

Tools in the ISDE use permanent *sentinel* nodes to locate starting points in the mutable tree structure. Three sentinel nodes, shown in Figure 4 are used to mark the beginning and end of the token stream and the root of the tree.

To create a new program, a null tree corresponding to only the sentinels in Figure 4 and an empty ('completing') production for the start symbol of the grammar is constructed. The initial program text is assigned temporarily as the lexeme of `bos`. Then a (batch) analysis is performed, which constructs the initial version of the persistent program structure; all subsequent structure is derived solely through the incorporation of valid modifications.

III. Incremental Parsing

Incremental parsing utilizes a persistent parse tree and detailed change information to restrict both the time required to re-parse and the regions of the tree that are affected. The input to the parser consists of both terminal *and nonterminal* symbols; the latter are a natural representation of the unmodified subtrees from the reference version of the parse tree. In this section we provide a brief overview of incremental sentential-form parsing; the details may be found in [20].[5]

To restore structural consistency, the tree is conceptually 'split' in a series of locations determined by the modifications since the previous parse. Modification sites can be either interior nodes with structural changes or terminal nodes with textual changes, and the split points are based on the dependencies determined by the read-ahead of the lexer and the (fixed) number of lookahead items used in constructing the parse table. The input stream to the parser will consist of both new ma-

[5]Our error recovery algorithm can also be used in conjunction with state-matching algorithms [7], [28] as well as non-deterministic parsing [21] (see Section VII-D).

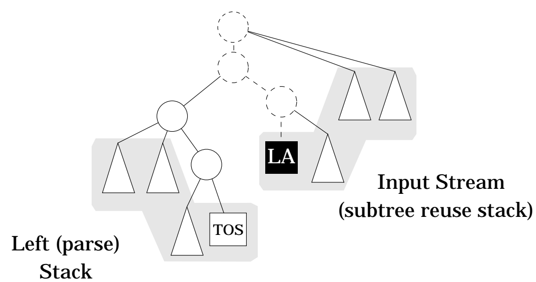

Fig. 5. Incremental parsing example. This figure illustrates a common case: a change in the spelling of an identifier results in a 'split' of the tree from the root to the token containing the modified text. The shaded region to the left becomes the initial contents of the parse stack, which is instantiated as a separate data structure since it may contain a mixture of old and new subtrees. The shaded region to the right provides the input stream for both the lexer and the parser. This stream (sometimes treated as a stack by the parser) is not explicitly materialized—its contents are derived by traversing the previous tree. Except when a new token has been created, the top element of the right stack serves as the parser's lookahead symbol. The subtrees in the input stream unchanged since the reference version constitute the potentially-reusable portion of the previous analysis.

terial (in the form of tokens provided by the incremental lexer) and reused subtrees; the latter are conceptually on a stack, but are actually produced by a directed traversal over the previous version of the tree. An explicit stack is used to maintain the new version of the tree while it is being built. This stack holds both symbols (nodes) and states (since they are not recorded within the nodes). Figure 5 illustrates a common case, where a change in identifier spelling has resulted in a split to the terminal symbol containing the modified text.

A sentential-form incremental parser works by parsing nonterminal symbols in addition to terminal symbols. For LR(0) parsers, the mere fact that the grammar symbol associated with a subtree's root node can be shifted in the current parse state indicates that the entire subtree can be consumed by the parser without further analysis. The situation is similar, though more complex, for the LR(1) case: the fact that a subtree with non-empty yield can be shifted indicates that all but the final sequence of reductions—those along the right-hand edge of the subtree—are known to be valid (and hence reusable without further investigation). A sentential-form parser can operate *optimistically*, shifting entire subtrees (including those with empty yield) even though this may occasionally permit invalid transitions. Any mistakes are discovered before additional material is shifted, and induce a limited form of backtracking.

When a non-trivial subtree occurring in the input stream cannot be shifted, it is removed and replaced by its constituents. If parsing cannot continue with a terminal symbol as lookahead *and* the material on top of the parse stack was optimistically shifted, then it is similarly broken down so that the top-of-stack symbol is a terminal. (Any reductions are undone to roll-back to the

state immediately following the shift of the rightmost terminal symbol to the left of the lookahead.) If a parse error is detected before further a shift occurs, the input is known to be invalid.

The incremental parser of [19] invokes error recovery in the same configuration (stack contents, parse state, and lookahead symbol) as a batch parser. For a complete LR(k) parse table, the symbol on top of the parse stack will be a terminal. When the parse table is SLR or LALR or contains default terminal reductions, the topmost stack symbol may be a nonterminal. At this point the parser will invoke the recovery routine. (Other than detecting the error at the appropriate location with respect to the input stream, no special requirement is placed on the incremental parsing algorithm.) As part of its processing, the error recovery routine will reset the configuration so that parsing can resume; note that the recovery may consume additional material from the input stack before it returns control to the normal parsing algorithm.

IV. Modeling Errors

Section II introduced a basic program representation and a model for the editing and transformation of programs through language-specialized analysis. Here we extend that representation to include *errors*, in the form of persistent, unincorporated modifications.

A. Maintaining Unincorporated Modifications

In a program without errors, the correctness properties of the incremental lexical and syntactic analyses guarantee that the text, tokens, and structure of the program are all consistent with one another and are valid with respect to the language definition. Any modifications performed by the user introduce temporary inconsistencies among (and possibly within) these different representations. When such modifications lead to another correct program state, the next invocation of incremental analysis will transform the program representation to the new state.

When one or more modifications introduced by the user do *not* result in a syntactically correct program state, the result of incremental lexing and parsing is unspecified: the language definition and the correctness proofs of these transformations do not address the construction of a persistent representation involving errors, despite its overarching practical importance in an incremental environment. Our solution is based on an observation that is simultaneously simple and powerful: *not all modifications need to be incorporated*. We permit the inconsistency induced by one or more user modifications to persist indefinitely; the goal will be to incorporate as many valid edits as possible while leaving *all* the invalid edits unincorporated. Clearly this cannot violate the correctness properties of incremental lexing or parsing as long as we consider all the unincorporated edits as pending modifications when incremental analysis is next invoked.

When only textual modifications and legal structural edits are permitted, the structure of the program representation remains correct (with respect to the grammar) at all times, although the lexeme⇔token mapping may be inconsistent until outstanding user modifications have been incorporated through analysis. (Section VII discusses support for structural edits that introduce non-grammatical tree structure.) The correctness of the program structure with respect to the grammar can then be established by simple induction. There is a simple relationship between the presence of errors and consistency properties: the program text as defined by the left-to-right concatenation of the lexemes constitutes a correct program if and only if the representation is free of unincorporated modifications following the analysis.

During re-analysis of a program containing errors, the incremental lexer and parser must investigate the site of each unincorporated change, since additional modifications may have changed the surrounding context in such a way that the former error is now valid. (In the next section we describe mechanisms to limit the scope of an error.) For these and other tools in the ISDE to locate errors efficiently, each node containing an error must be distinguished and the path between the root of the tree and each error-containing node must be marked. This is accomplished with a pair of boolean attributes similar to the `local` and `nested` attributes provided by the history services for change reporting.[6] The incremental lexing and parsing algorithms treat errors and error paths in the same fashion as local and nested change attributes in determining which regions of the program structure require re-analysis.

B. Isolating Errors

The drawback to the model described above is that it applies *globally*: every error site must be treated as a pending modification when re-analysis is requested, and no legal modifications can be incorporated until all the errors have been corrected. However, it is not necessary to treat the entire program as a unit; in this section we use incremental analysis and the relationship between the current analysis and the previous structure of the tree to *isolate* errors from one another.

Isolation makes our model of error recovery as unincorporated changes meaningful, by allowing legal modifications outside the isolated regions to be successfully integrated by the incremental lexer and parser. By separating the errors, isolation also improves the performance of subsequent analyses: it is not necessary to reinspect the errors in an isolated region unless that region contains new user modifications or is affected by changes to the surrounding context. The independence of isolated structure is also useful *within* the region,

[6] Changes to the `local_error` and `nested_error` attributes must *themselves* be captured in the history log—this allows the transformation induced by incremental analysis to be fully reversible, even when it involves error recovery.

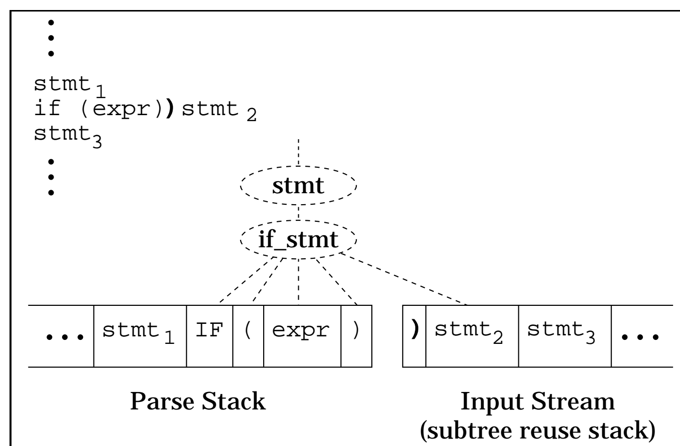

Fig. 6. Isolation using an internal node.

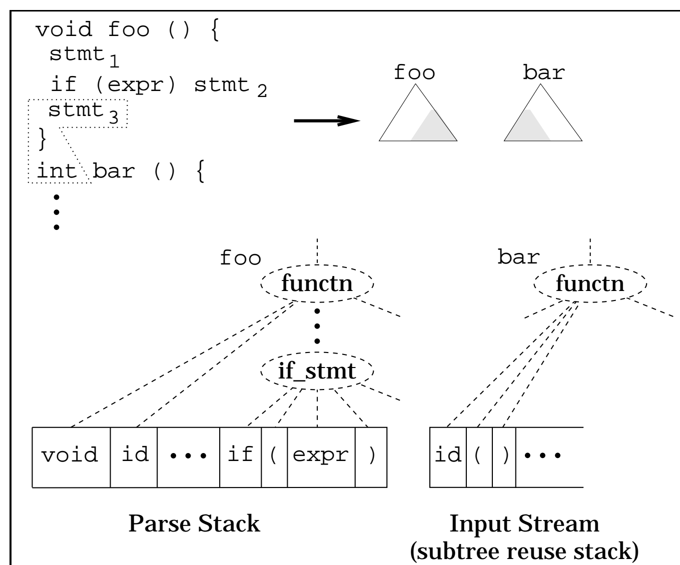

Fig. 7. Isolation in a 'stack recovery' situation.

since its recovery can be computed separately from the analysis of surrounding structure or other erroneous regions of the program by construction.

Isolation is defined by the analysis techniques: the dependencies between program components induced by lexical and syntactic analysis determine the size of an isolated region. However, computing an isolation region solely through the analysis of the current program state is problematic: this essentially implements a conventional (batch) non-correcting recovery, subject to the shortcomings described in Section I. Fortunately there is an additional source of information in an ISDE: the *previous* structure of the program is accessible, and can be used along with the dependency information to separate and contain errors.

Once an error has been detected by the lexer or parser, the previous structure provides a useful guide for constraining its effect. The isolation algorithm locates a well-formed subtree that existed in the previous version which can be retained in the current version of the program structure to contain the site of the error. (The 'matching condition' used by some state-matching incremental parsers computes a similar relationship between the old and new trees [28], [29].) When isolated regions are small, each is likely to contain only a single error (thus preventing the recovery of one problem from contaminating another) and most correct modifications will lie outside all isolated regions (allowing them to be successfully incorporated).

Figure 6 contains a simple isolation example. Here the user has mistakenly inserted an additional right parenthesis following the test expression in a condition. (The syntax of C is used in this and other examples.) The problem is conceptually contained within the `if_stmt` from the previous version of the program structure. The isolation algorithm discovers this fact and 'reverts' this statement to its previous form. The erroneous insertion is left as an unincorporated textual modification, and presented to the user as an error.

Isolation is not limited to purely 'local' problems; Figure 7 contains an example that would result in an extensive secondary repair in conventional batch recovery. In our approach, the accidental deletion that merges the two function definitions is recovered by the isolation process. The use of the previous structure allows the right side of the first definition's structure and the left side of the second definition's subtree to be restored. (The actual node chosen for isolation in this case will be the lowest common ancestor in the sequence containing these function definitions; sequence representation is discussed in Section V-D.)

The paths to unincorporated modifications defined by `nested_error` attributes are terminated immediately below the root of the isolated subtree. This allows subsequent analyses to avoid re-inspecting the isolated errors unnecessarily.

C. Computing Isolation Regions

Figure 8 contains the routines to locate and apply isolation. Isolation begins by removing any default reductions from the parse stack using `right_breakdown`; the associated nodes are ignored in the subsequent search for an isolation node. The search for an isolation candidate then proceeds by comparing the current and previous versions of the program's structure in the region of the error. (Note that, it is always possible to isolate *some* subtree, since the ultra root persists across all versions of the tree.) Having found a suitable candidate, we can cut back the parse stack to the beginning of the isolated subtree, shift the isolated subtree's root node, advance the lookahead pointer to the following subtree in the previous version, and re-start the analysis in the resulting configuration.

The search for an isolation candidate proceeds in two dimensions: each entry on the (current) parse stack that is not a new node is considered, and for each of these nodes its set of ancestors in the previous ver-

```
SetOfBool paths_to_ignore;

Find a node that exists in the previous version of the parse tree that
covers the damaged region.
recover () {
  SetOfBool paths_to_ignore = ∅;
  right_breakdown();
  int sp = 0;
  for (NODE *n = stack.node(); !n→is_bos; {
    sp++;
    if (n→created_in(gvid())) continue;
    int offset = stack.offset(stack.length - 1);
    if (valid_iso_subtree(n, offset, stack.state()))
      return isolate((NODE*)n, sp, 1);
    If the root of this subtree is new, keep looking down the parse stack.
    Otherwise, try searching his ancestors.
    int cut_point;
    for (NODE *ancestor = last_edited→parent(n);
         ancestor != ultra_root &&
         stack.get_cut(ancestor, cut_point);
         ancestor = last_edited→parent(ancestor))
      if (ancestor ∉ paths_to_ignore &&
          valid_iso_subtree(
            ancestor, stack.offset(cut_point),
            stack.state(cut_point), cut_point))
        return refine(ancestor, sp, cut_point);
      else {add ancestor to paths_to_ignore; n = ancestor;}
    state = stack.state(); stack.pop(); n = stack.node();
  }
  return ultra_root; Isolate the entire tree.
}

Remove any subtrees on top of parse stack with null yield, then
break down right edge of topmost subtree.
right_breakdown () {
  NODE *node;
  do { Replace node with its children.
    node = parse_stack→pop();
    add node to paths_to_ignore;
    Does nothing when child is a terminal symbol.
    foreach child of node do shift(child);
  } while (is_nonterminal(node));
  shift(node); Leave final terminal symbol on top of stack.
}

Shift a node onto the parse stack and update the current parse state.
void shift (NODE *node) {
  parse_stack→push(parse_state, node);
  parse_state =
    parse_table→GOTO(parse_state, node→symbol);
}

Get offset of first character not before yield of index^th entry.
int Stack::offset (int index) {
  for (int i = 0, offset = 0; i < index; i++)
    if (i == index) return offset;
    else offset = offset + entry[i].node→text_length;
}

Compute stack entry corresponding to leading edge of node's subtree in
the previous version. Returns false if no entry is so aligned.
bool Stack::get_cut (NODE *node, int &cut_point) {
  int old_offset = last_edited→offset(node);
  for (int cut_point = 0, offset = 0;
       cut_point < length; cut_point++)
    if (offset > old_offset) return false;
    else if (current_offset == old_offset) return true;
    else
      offset = offset + entry[cut_point].node→text_length;
  return false;
}

Isolate. . .finish!
isolate (NODE *node, int sp, int cut_point) {
  stack.unpop(sp - 1);
  refine(node);
  parse_state = stack.state(sp - 1 + cut_point);
  stack.pop(sp - 1 + cut_point);
  shift(node);
}
```

Fig. 8. Isolating syntax errors. The `valid_iso_subtree` test is shown in Figure 9. The operation of the `refine` routine is presented in Figure 13, after the refinement techniques have been presented.

```
bool valid_iso_subtree (NODE *node, int left_offset,
                        int state, int cut_point) {
  if (node ∈ isolation_rejects) return false;
  add node to isolation_rejects;
  The starting offset of the subtree must be the same in both the
  previous and current versions.
  The ending offset must meet or exceed the detection point.
  int left_offset = new_offset;
  if (left_offset != last_edited→offset(node))
    return false;
  if (left_offset > detection_offset) return false;
  if (left_offset + node→text_extent(last_edited→gvid()) <
      detection_offset)
    return false;
  [Lexical tests —see Section VII-C]
  Now see if the parser is willing to accept this isolation, as
  determined by the shiftability of its root symbol in the current
  stack configuration.
  stack.pop(cut_point);
  action = next_action(node, state);
  stack.unpop(cut_point);
  return action == SHIFT;
}
```

Fig. 9. Procedure to test whether a given subtree is a valid isolation candidate. The tests include textual alignment with respect to the previous version of the subtree, lexical consistency checks, and an LR(0) (shift) test for the symbol labeling the root node of the subtree.

sion is considered.[7] Each candidate is tested with `valid_iso_subtree`, to determine whether an isolation rooted there would cover the damaged area and be acceptable to both the lexer and parser.

The choice of an isolation region must simultaneously maintain *all* analysis invariants. The primary lexical restriction is that the isolated region contains the same text (range of offsets) in both the previous and current versions—otherwise characters could be lost or appear multiple times. Additional lexical invariants may need to be imposed depending on the expressive power of the lexical description language. (Section VII-C describes the impact of several lexical description features on the isolation conditions.)

The syntactic condition for a successful isolation requires that the subtree from the previous version of the program 'align' with the current parse stack—the left edge of the isolated subtree must correspond to the left edge of some subtree on the parse stack. This allows the isolated subtree to replace partial analysis results by popping one or more entries off the stack and pushing the root node from the isolated subtree. Such a push must make sense with respect to the parse table: shifting the symbol (left-hand side) of the production labeling the root node of the isolated subtree must be a legal move in that configuration.

Passing the alignment and shift tests does not guarantee that the parser will be able to continue parsing successfully when the recovery is complete; the *right-context* of the isolated subtree may not be legal. Although it would be possible to check this as part of the isolation conditions, a simpler technique is to allow the parser to detect the problem and re-invoke recovery: a

[7] Different search strategies could be employed; for instance, nodes deeper in the stack may sometimes be preferable to ancestors high in the previous tree.

```
Discard changes and record errors in the subtree rooted at node;
discard_changes_and_mark_errors (NODE *node) {
  node→discard_changes();
  if (node→has_changes(reference_version, local)
    if (!node→local_errors) {
      node→local_errors = true;
      node→compute_presentation(reference_version);
    }
  if (∃ child of node s.t.
       (child→local_errors || child→nested_errors)))
    node→nested_errors = true;
  else node→nested_errors = false;
}
```

Fig. 10. Dicarding partial analysis results and marking unincorporated modifications ('error') local to a single node. The structure and content of the subtree rooted at node are reverted to their state in the previous version of the program. Any user modifications (textual or structural) within this subtree are marked as unincoporated errors, and nested error attributes are set to record the path between node and the location of each such error. The presentation of errors is discussed in Section VI.

larger isolation region will then be selected, since all previously isolated nodes are rejected as candidates.

Once chosen, any partial analysis results applied an isolation region can be removed by a recursive application of the algorithm in Figure 10; this will revert each modified subtree to its state in the previous version of the program (where the structure is known to be correct). Any user changes since the reference version are marked as unincorporated errors. By construction, modifications (valid or invalid) outside the isolated region are unaffected. (In Section V we develop methods to incorporate legal modifications *within* the isolated region.)

The general search for an isolation subtree considers only interior nodes. Since errors are sometimes contained within the textual modification(s) applied to a single token, the recovery can also consider *token-level isolation* prior to the algorithm described above. Reasonable choices of tokens to examine include the top-of-stack and lookahead symbols, as well as tokens close to them in the previous version. If a token-level isolation succeeds, the algorithm in Figure 10 is applied to it, the configuration is reset based on the location of the token relative to the error, and analysis is re-started.

D. Handling Lexical Errors

Although the previous sections have focused on recovery from *syntactic* errors, the mechanisms are also applicable to *lexical* problems. Errors at the lexical level can be discovered by including explicit rules in the lexical description to match invalid sequences; when one of these patterns is recognized, it creates a special error token. A simpler mechanism, which can be used either alone or in concert with explicit error patterns, is *implicit* detection: Instead of modifying the description, problems are discovered when characters cannot be legally recognized as belonging to any pattern; each contiguous sequence of unmatched characters produces an instance of a special *unmatched* token class. The parser will be unable to shift an unmatched-text token, since it is neither a legal whitespace token nor does it represent a terminal symbol in the grammar. This will result in the detection of a parse error; the subsequent recovery will then treat the erroneous textual changes using the mechanisms already discussed.

Explicit error tokens can either induce this same behavior or persist as whitespace tokens—the latter behavior is occasionally useful when the error is so common or idiosyncratic that recognizing a superset of the actual language is preferable to a normal error presentation. Regardless of the policy chosen, no special effort is required to recognize, recover from, or present lexical problems. (However, recovery must respect lexical invariants as well as syntactic restrictions; Section VII-C describes the impact of various features of the lexical description language on the recovery process.)

V. Incorporating Modifications Within Isolated Regions

If every isolated region contained only errors and no legal modifications, then the algorithm in Figure 10 would constitute a sufficient recovery. However, an isolated subtree will in general contain several legal modifications, especially when the isolated subtree is large (as can happen with errors in bracketing constructs or lengthy sequences). In this section we consider *refinement* techniques that can integrate some, and in many cases all, of the correct modifications within an isolated subtree.

A. Retaining Partial Analysis Results

Refinement employs two different techniques, depending on the location of the modifications relative to the detection point that triggered the recovery. The first technique attempts to *retain* modified subtrees that have already been analyzed. When the recovery routine is invoked, the incremental lexer and parser have already seen any material to the left of the detection point. In general, several legal modifications will already have been incorporated into this material (represented by new or modified subtrees on the parse stack). Subject to certain restrictions, these subtrees can be retained instead of discarded. This will allow *nested* isolation regions to persist in those subtrees, and avoids the redundant work of re-marking errors within them. Figure 11 illustrates a simple example.

The two-pass retention algorithm is shown in Figure 12. In the first pass, the previous structure of the portion of the isolated region to the left of the detection point is examined; any subtrees reconstructed in their entirety and that meet alignment[8] and lexical invariant restrictions can be retained in their *new* form instead of being discarded. The second pass is used to actually carry out this transformation, discarding re-

[8]Note that permitting the incorporation of valid modifications within the isolated region implies that the mapping between lexemes and characters may change, even though the character yield of the isolated region *as a whole* is unchanged from the previous to the current version.

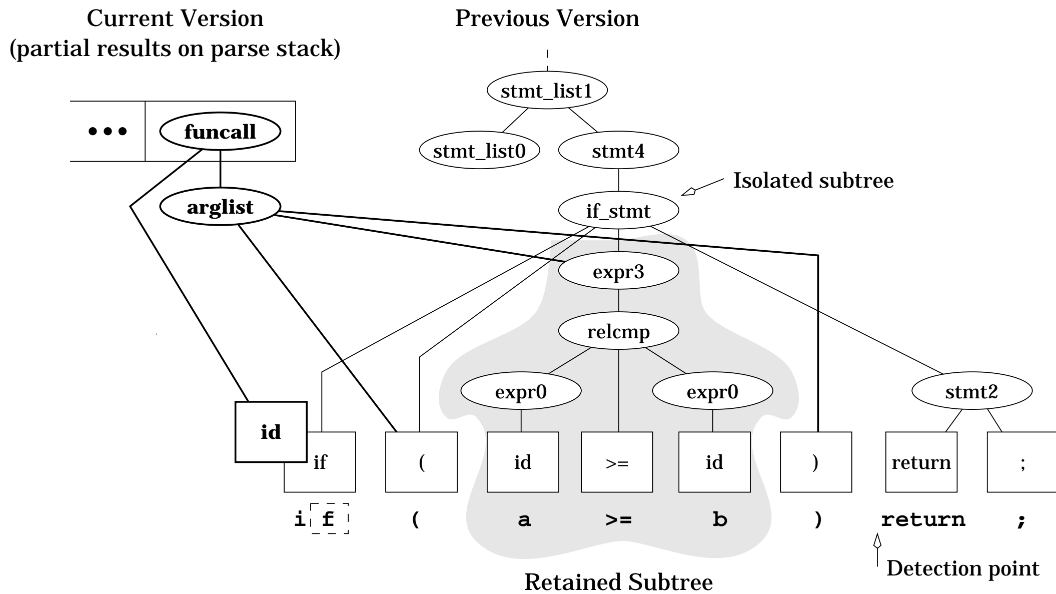

Fig. 11. Retaining partial analysis results. The mistaken change of the keyword `if` to the identifier `i` causes the beginning of the statement to be re-interpreted as a function call. When the `return` keyword is reached, the parser detects the problem. The test expression (shaded) represents a subtree shared by *both* versions: no inspection of this subtree is required if it contains no edits. More interestingly, if it *is* changed to any other legal expression, the recovery can retain the analyzed result—the keyword error does not preclude the incorporation of correct changes to the test expression.

sults that cannot be retained and marking the unincorporated modifications using the algorithm of Figure 10. In general this creates a 'canopy' of structure from the previous version with aribitrarily large subtrees from the current analysis embedded within it. A two-pass algorithm is necessarily to avoid corrupting the new version until all the decisions about retaining portions of it have been made. Unmodified subtrees from the previous version that occur in the new structure are not inspected further by either pass.

B. Out-of-Context Analysis

Retaining pre-parsed subtrees permits the incorporation of legal modifications within an isolated region to the *left* of the detection point. Modifications may also exist to the right of the detection point, in which case the lexer and parser have not processed the affected subtrees. In Figure 11, suppose that, in addition to the error in the keyword `if`, the user replaces the return statement with a different statement. Without the techniques described below, such a modification would go unincorporated.

Even though no analysis has been performed on modified subtrees to the right of the detection point, it is not correct to simply restart analysis within the isolated region. (Error recovery guarantees that a legal analysis configuration has been restored only at its conclusion.) Instead, we perform an *out-of-context analysis*, which attempts to analyze each (maximal) subtree containing user modifications *independent* of its surrounding context.

As with partial analysis retention, we place sufficient conditions on out-of-context analysis to ensure that any

Given the root of a subtree from the previous version of the tree, itemize the retainable subtrees within it.

```
find_retainable_subtrees (NODE *node) {
  if (exists_in_new_tree(node) &&
      (!node→changes(reference_version, nested) ||
       same_text_pos(node))) {
    add node to retainable;
    return;
  }
  foreach child of node in the previous version do
    find_retainable_subtrees(node);
}
```

Retain retainable subtrees and discard remaining structure rooted at the argument node.

```
retain_or_discard_subtrees (NODE *node, NODE *parent) {
  if (node∈retainable) {
    node→set_parent(parent);
    remove node from retainable;
    return;
  }
  discard_changes_and_mark_errors(node);
  foreach child of node do keep_or_discard_subtrees(node);
}
```

Fig. 12. Computing partial analysis retention. The top function is the first pass, which computes the set of retainable subtrees. The bottom function is the second pass, which retains the legal subtrees and discards any partial results for the remaining nodes. `same_text_pos` determines whether a subtree's yield occupies the same character offset range as in the previous version of the program.

incorporated changes do not interfere with the isolation itself or with the handling of adjacent subtrees. Unlike retention, however, the sufficiency tests for out-of-context analysis are distributed: Prior to analysis we verify that the target subtree contains at least one modification, has a non-null yield, and is not followed by a terminal requiring analysis. During the subtree's analysis we check whether the lexer was able to synchronize with the previous contents before hitting the rightmost

edge—otherwise the lexical analysis would 'bleed' into the next subtree to the right. Finally, at the conclusion of the subtree's analysis we must ensure that the symbol of the production labeling its (possibly changed) root is the same as in the previous version of the program.[9] When all of these conditions are met, the out-of-context analysis succeeds, and the analyzed results are integrated into the current version of the program.

The algorithms for incremental lexing and parsing during out-of-context analysis are the same as for normal analysis. To simplify the handling of out-of-context analysis, we can build a temporary set of sentinel node that allow the subtree to appear as the entire program. Out-of-context parsing requires augmenting the parse table to allow any symbol to serve as the start symbol [31]. This requires a distinguished terminal for each symbol to avoid conflicts; the temporary `bos` token can be used to represent this 'starting terminal' in order to place the parser in the correct state to process the subtree in an out-of-context manner.

The error recovery routines themselves are available during an out-of-context analysis: errors detected while the subtree is being analyzed are processed by reentering the recovery routine. This permits nested isolation and refinement to occur in modified subtrees to the right of the (outer) detection point just as they can exist in retained analysis results to the left. The failure of any sufficiency checks applied during or immediately after the out-of-context analysis of a subtree result in a nested recovery that isolates the subtree being processed. (In general, partial analysis results will be valid and will be retained within the subtree, even though the out-of-context analysis as a whole did not succeed.)

C. Refinement Algorithm

Figure 13 contains the top-level routine to initiate a refinement of the recovery within an isolated region. This routine divides the isolated structure into three groups: subtrees to the left of the detection point, subtrees to the right of it, and subtrees that span the detection point. The latter are treated in the same fashion as unsuccessful candidates for retained or out-of-context analysis: any pending local changes are discarded, local changes by the user become errors, and the node's (previous) children are recursively investigated if the node indicated nested changes.

The correctness of these refinement techniques can be easily established as a left-to-right inductive proof on the subtrees within the isolation region; unmodified subtrees remain unchanged, discarded changes revert to previously correct structure, and any subtrees chosen for analysis retention or out-of-context analysis possess (by construction) sufficient conditions to ensure that their handling is independent of the surrounding material.

[9] Unlike Degano [30], we apply this restriction only as a mechanism for improving error recovery; this restriction does *not* apply to the user's editing model.

Isolate the argument and recursively recover the subtree that it roots.

```
refine (NODE *node) {
  int offset = last_edited→offset(node);
  pass1(node, offset);
  node→discard();
  node→local_errors = node→nested_errors = false;
  pass2(node, offset);
}

pass1 (NODE *node, int offset) const {
  foreach child of node in the previous version do {
    if (offset + child→text_extent() <= detection_offset)
      find_retainable_subtrees(child);
    else pass1(child, offset);
    offset += child→text_extent();
  }
}

pass2 (NODE *node, int offset) {
  foreach child of node in the current version do {
    if (offset > detection_offset)
      attempt_out_of_context_analysis(child);
    else if (offset + child→text_extent()
             <= detection_offset)
      retain_or_discard_subtrees(child, node);
    else {
      discard_changes_and_mark_errors(node);
      pass2(node, offset);
    }
    offset += child→text_extent();
  }
}
```

Fig. 13. Refining an isolated region. The `refine` routine performs two passes over the isolated subtree. The first pass is read-only and computes the set of retainable subtrees. The second pass reverts any unretainable material to the left of the detection point, invokes out-of-context analysis on any candidate subtrees to the right of the detection point, and discards changes on material that spans the detection point.

There are several implicit trade-offs in the computation of candidate nodes for isolation, retention, and out-of-context analysis. For isolation, increased time spent searching for a tighter isolation region may provide little practical benefit even if it succeeds, especially since the refinement techniques are so powerful. For the refinement tests, the independence constraints we impose can result in the failure to incorporate some legal changes. These restrictions could be relaxed—for instance, allowing several subtrees to be *jointly* retained or analyzed out of context, where considered singly they would fail. However, in addition to a more complicated correctness proof, looser constraints imply the need for more complex verification checks, limited backtracking, or both; any potential benefits must thus be weighed against the increased computation required and the fact that refinement as presented is already extremely effective.

D. Running Time

Optimal methods for incremental lexing and sentential-form parsing require time $O(t + s \lg N)$, for t new terminal symbols and s modification sites in a tree with N nodes. This result assumes that lengthy sequences are identified in the grammar and represented as balanced trees in the resulting program structure; see [20] for a discussion of the issues. The presence of history-sensitive error recovery does not affect the running time

of either algorithm, since no additional work is required until a recovery is actually invoked.

Under the same assumptions regarding the representation of lengthy sequences, the error recovery routines presented here require a worst case running time of $O(t + s(\lg N)^2)$. The additional $\lg N$ factor is inherent in the approach: intuitively, it represents the need to compare the current and previous structure of the tree in validating isolation and retention candidates. (However, the worst case behavior does not appear to occur in practice; in trials with our prototype using several languages, error recovery executes as fast as the analysis of correct structure.)

VI. Presenting Errors

The previous sections have concentrated on detecting, isolating, and refining the program representation in the presence of errors. While an effective treatment of errors is important to the analysis algorithms and other tools in the ISDE, ultimately it is the comprehensible presentation of errors to the user that determines the effectiveness of a recovery.

Batch error recovery based on a correcting strategy typically uses information about the repair to construct an *error message* to explain the correction (and hopefully the error itself) to the user. A history-based approach provides a simpler and more effective method for communicating with the user: included among the unincorporated edits is the cause of the problem; in practice the isolation and refinement strategies are often effective in producing a set of unincorporated errors that includes all *and only* the actual errors.

In a history-based error recovery, the obvious way to present the recovery result to the user is to visually indicate the changes that were not successfully incorporated.[10] Since this interface has the user's own changes as its vocabulary and naturally correlates actual changes with the displayed indication of the problem, it subsumes and improves upon the conventional technique of generating explanatory messages.

Figures 1 and 2 suggest one way in which invalid textual insertions and deletions can be presented, using the difference between the current and the previous (correct) contents of the tokens in the affected region.[11] More elaborate presentations of the accrued changes, involving color, side-by-side comparisons, etc. can be provided using available information about the unincorporated material. The presentation attributes are computed when unsuccessful analysis results are discarded from a node; in Figure 10, the call

node→compute_presentation(reference_version)

indicates the computation of presentation attributes for any user changes local to `node`. The specific textual or structural changes to `node` can be extracted from its local history log [23].

VII. Extensions

In this section we summarize a number of extensions to the basic history-sensitive error recovery technique.

A. Structural Editing

If the user is permitted to perform arbitrary structural editing, then the program structure is no longer guaranteed to be valid, even after an analysis is performed. However, unincorporated structural edits represent all and only the points where the structure is not correct—the structure remains 'piecewise' correct. Isolation, refinement, and error marking can all be extended to handle persistent invalid structure represented in this manner. (The presentation of structural errors to the user is simplified by the presentation of a parallel structural view alongside the textual presentation of the program.)

There is a distinction between the fixed arity model of nodes previously presented and a model where, in the case of a structural error, the number of children assigned to a given symbol is permitted to vary. The flexibility of the latter model is required to represent correcting strategies in large text insertions (Section VII-F). We can simulate the effect of a varying number with fixed arity nodes by using the sequence representation described in Section V-D; in the case of errors this is used for uniformity in the structural representation, *not* for performance reasons. (The number of children of a given node is still assumed to be effectively bounded in the case of a structural error.) In all cases, nodes are typed by their role in the grammar; both missing information and malformed (variable arity) structural errors can be handled as completing productions for the appropriate grammar symbol.

B. Whitespace

Explicit whitespace material can be integrated into the persistent program structure of an ISDE through grammatical transformations or extensions to the incremental parser [32]. Either approach can be used with the recovery techniques described here, which require only minor modifications to enable the 'parsing' of whitespace material during recovery.

The exception to this is the handling of leading whitespace during out-of-context analysis. In the top-level program representation, any explicit whitespace material that precedes the first terminal symbol can be represented as following the `bos` pseudo-token. In an out-of-context analysis, the same phenomenon may occur; however in this case the `bos` token is temporary. At the conclusion of a successful out-of-context analysis, any leading whitespace must be re-integrated with the preceding subtree; this may require merging two white-

[10] In our experience, users do not benefit from a visual presentation of the isolation regions.

[11] The user may not correct the error by the next analysis, and may update the location containing the error without correcting the problem. Thus in composing the presentation, the recovery should combine new modifications with the existing display until the error is finally corrected (or the region is removed from the program).

space sequences. (Alternatively, the absence of newly-formed leading whitespace can be introduced as one of the conditions for a successful out-of-context analysis.)

C. Generalized Incremental Lexing

Certain features of the lexical description language, such as arbitrary lookahead, multiple start states, and atomic sequences [19], can complicate error recovery. These features introduce additional dependencies between tokens, restricting the conditions in which an isolation or refinement candidate is acceptable. (Actual inter-token dependencies are usually trivial—no reduction in the power of isolation or refinement due to lexical invariants occurs in practice.) Here we discuss one such feature, atomic sequences, to illustrate the effect on the recovery algorithms.

In choosing both isolation candidates and subtrees for analysis retention, the leftmost and rightmost tokens must be in singular sequences (i.e., not part of any non-trivial atomic token sequence). This condition must hold in *both* the current and previous version of the program. In testing a subtree for out-of-context analysis, this condition is checked in the previous structure only; during the out-of-context analysis an attempt by the incremental lexer to construct an atomic sequence spanning the right edge of the subtree will result in an error cause a recursive recovery.

D. Non-deterministic Parsing

Non-deterministic incremental parsing [21] can also be used with history-sensitive error recovery. The primary change needed to support IGLR parsing is an extension of the isolation boundary test to ensure that each non-deterministic region is treated as an atomic unit. Partial analysis retention and out-of-context parsing have similar restrictions. (This has no practical effect on the efficacy of the recovery, due to the small size of these regions in actual programs.)

E. Severity Levels

The isolation and refinement methods described earlier treat all unincorporated modifications identically. However, in some cases the recovery can distinguish between modifications *known* to be errors and modifications which it cannot prove correct *or* incorrect. When a change remains unincorporated because the sufficient conditions for partial analysis retention or out-of-context analysis were not met, any unincorporated changes in the affected subtree may be valid, but analysis limitations will cause them to be treated as errors. The recovery process can expose this additional information by assigning an integer *severity level* to each unincorporated change. The interpretation of these levels will be heuristic, but an appropriate choice in their translation to presentation characteristics can assist the user in distinguishing actual problems from incomplete or insufficient analysis results.

F. Recovery for Large Insertions

Insertions of large text strings mimic batch parsing, since no previous history for such material exists. Error recovery within such a region is limited to batch techniques.[12] Either correcting or non-correcting methods may be applied. (In the case of a correcting strategy, persistent 'error' nodes must be introduced into the representation to model deviations from valid syntax; see Section VII-A above.) Recovery methods applied to large insertions must operate only within the boundaries of the inserted material; in particular, 'stack cutting' and right context acquisition must treat material outside the inserted region as read-only.

VIII. Conclusion

This paper presents a non-correcting approach to the detection and presentation of syntactic errors that is suitable for use in an interactive and incremental software developement environment. Unlike previous techniques, it uses the contents of the development log to correlate the modifications actually made by the user to the errors detected in the program. Unlike batch non-correcting strategies, it precisely identifies the location of errors, including errors involving closing syntax. History-sensitive error recovery can be incorporated easily into existing algorithms for incremental lexing and parsing. The approach is itself incremental, requires no language-dependent information or user interaction, and provides a more accurate and informative report than any previous approach to error recovery.

References

[1] Pierpaolo Degano and Corrado Priami, "Comparison of syntactic error handling in LR parsers", *Software—Practice & Experience*, vol. 25, no. 6, pp. 657–679, Jun. 1995.
[2] Robert Paul Corbett, *Static Semantics and Compiler Error Recovery*, PhD thesis, University of California, Berkeley, 1985, Available as technical report UCB/CSD 85/251.
[3] Michael G. Burke and Gerald A. Fisher, "A practical method for LR and LL syntactic error diagnosis and recovery", *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 2, pp. 164–197, Apr. 1987.
[4] U. Bianchi, P. Degano, S. Mannucci, S. Martini, B. Mojana, C. Priami, and E. Salvator, "Generating the analytic component parts of syntax-directed editors with efficient error recovery", *J. Syst. Softw.*, vol. 23, no. 1, pp. 65–79, Oct. 1993.
[5] D. Notkin, R. J. Ellison, B. J. Staudt, G. E. Kaiser, E. Kant, A. N. Habermann, V. Ambriola, and C. Montangero, "Special issue on the GANDALF project", *J. Syst. Softw.*, vol. 5, no. 2, May 1985.
[6] Michael L. Van De Vanter, Susan L. Graham, and Robert A. Ballance, "Coherent user interfaces for language-based editing systems", *Intl. J. Man-Machine Studies*, vol. 37, pp. 431–466, 1992.
[7] Fahimeh Jalili and Jean H. Gallier, "Building friendly parsers", in *9th ACM Symp. Principles of Prog. Lang.*, Albuquerque, N.Mex., 1982, pp. 196–206, ACM Press.
[8] Rolf Bahlke and Gregor Snelting, "The PSG system: From formal language definitions to interactive programming environments", *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 4, pp. 547–576, Oct. 1986.
[9] E. Steegmans, J. Lewi, and I. Van Horebeek, "Generation of interactive parsers with error handling", *IEEE Trans. Softw. Eng.*, vol. 18, no. 5, pp. 357–367, May 1992.

[12] Although analysis near the right edge of the region may be more powerful than in a conventional recovery, given that non-trivial subtrees that are available in the input stream.

[10] Ian Jacobs and Laurence Rideau-Gallot, "A Centaur tutorial", Tech. Rep. 140, INRIA, Jul. 1992.
[11] Graham Ross, "Integral C – A practical environment for C programming", in *Proceedings of the Second ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*. 1986, pp. 42–48, ACM Press.
[12] Robert A. Ballance, Susan L. Graham, and Michael L. Van De Vanter, "The Pan language-based editing system", *ACM Trans. Softw. Eng. and Meth.*, vol. 1, no. 1, pp. 95–127, Jan. 1992.
[13] H. Richter, "Noncorrecting syntax error recovery", *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 3, pp. 478–489, Jul. 1985.
[14] Gordon V. Cormack, "An LR substring parser for noncorrecting syntax error recovery", in *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*. Jun. 1989, vol. 24(7), pp. 161–169, ACM Press.
[15] Jan Rekers and Wilco Koorn, "Substring parsing for arbitrary context-free grammars", *SIGPLAN Not.*, vol. 26, no. 5, pp. 59–66, May 1991.
[16] Joseph Bates and Alon Lavie, "Recognizing substrings of LR(k) languages in linear time", *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 1051–1077, May 1994.
[17] Mark N. Wegman, "Parsing for structural editors", in *Proc. of 21st Annual IEEE Symposium on Foundations of Computer Science*, Syracuse, N.Y., Oct. 1980, pp. 320–327, IEEE Press.
[18] Bernd Fischer, Carsten Hammer, and Werner Struckmann, "ALADIN: A scanner generator for incremental programming environments", *Software—Practice & Experience*, vol. 22, no. 11, pp. 1011–1025, 1992.
[19] Tim A. Wagner and Susan L. Graham, "General incremental lexing", 1997, In preparation.
[20] Tim A. Wagner and Susan L. Graham, "Efficient and flexible incremental parsing", 1996, Submitted to *ACM Trans. Program. Lang. Syst.*
[21] Tim A. Wagner and Susan L. Graham, "Incremental analysis of real programming languages", in *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*. Jun. 1997, vol. ??(??) of SIGPLAN Not., pp. ??–??, ACM Press.
[22] Tim A. Wagner and Susan L. Graham, "Integrating incremental analysis with version management", in *5th European Softw. Eng. Conf.*, Berlin, Sep. 1995, number 989 in LNCS, pp. 205–218, Springer-Verlag.
[23] Tim A. Wagner and Susan L. Graham, "Efficient self-versioning documents", in *CompCon '97*, San Jose, Calif., Feb. 1997, pp. 62–67, IEEE Computer Society Press.
[24] Susan L. Graham, "Language and document support in software development environments", in *Proc. DARPA '92 Software Technology Conf.*, Los Angeles, Calif., Apr. 1992.
[25] Brian M. Dennis, Roy Goldman, Susan L. Graham, Michael A. Harrison, William Maddox, Vance Maverick, Ethan V. Munson, and Tim A. Wagner, "A document architecture for integrated software development", Unpublished, 1995.
[26] Robert A. Ballance, Jacob Butcher, and Susan L. Graham, "Grammatical abstraction and incremental syntax analysis in a language-based editor", in *Proceedings of the ACM SIGPLAN '88 Symposium on Compiler Construction*, Atlanta, Ga., Jun. 1988, pp. 185–198, ACM Press.
[27] Görel Hedin, *Incremental Semantic Analysis*, PhD thesis, Department of Computer Science, Lund University, Mar. 1992.
[28] J. M. Larchevêque, "Optimal incremental parsing", *ACM Trans. Program. Lang. Syst.*, vol. 17, no. 1, pp. 1–15, 1995.
[29] Carlo Ghezzi and Dino Mandrioli, "Augmenting parsers to support incrementality", *Journal of the ACM*, vol. 27, no. 3, pp. 564–579, Jul. 1980.
[30] Pierpaolo Degano, Stefano Mannucci, and Bruno Mojana, "Efficient incremental LR parsing for syntax-directed editors", *ACM Trans. Program. Lang. Syst.*, vol. 10, no. 3, pp. 345–373, Jul. 1988.
[31] Luigi Petrone, "Reusing batch parsers as incremental parsers", in *Proc. 15th Conf. Foundations Softw. Tech. and Theor. Comput. Sci.*, Berlin, Dec. 1995, number 1026 in LNCS, pp. 111–123, Springer-Verlag.
[32] Tim A. Wagner and Susan L. Graham, "Modeling explicit whitespace in an incremental SDE", 1997, Submitted to Software—Practice & Experience.
[33] Tim A. Wagner, *Practical Algorithms for Incremental Software Development Environments*, PhD thesis, University of California, Berkeley, 1997, Available as technical report UCB/CSD 97/???

Tim A. Wagner received his BSE in Computer Science from Princeton University in 1989 and an MS degree from the University of California, Berkeley, where he is currently completing his PhD in Computer Science. He has been a member of the *Pan* and *Ensemble* research groups while at Berkeley. His research interests include incremental language analysis for software development environments as well as compiler optimizations and systems-level support for Internet-based execution. He is a member of Tau Beta Pi and a student member of IEEE and ACM.
E-mail: `twagner@cs.berkeley.edu`.
URL: `http://http.cs.berkeley.edu/~twagner`

Susan L. Graham is a professor in the Computer Science Department at the University of California at Berkeley. And some more bio text here, it has to go on for some time in order for the badly-written macro that they used to format this region to actually lay out the text without splitting individual words letter by letter. In fact it's quite amazing how their macro performs. Or perhaps one should say doesn't perform, given the results. Now we add one more sentence to force the email line below the picture, so that is looks more normal.
E-mail: `graham@cs.berkeley.edu`
URL: `http://http.cs.berkeley.edu/~graham`