# A Notation for Non-Linear Program Edits[†]

Martin Erwig
Oregon State University
erwig@eecs.oregonstate.edu

Karl Smeltzer
Oregon State University
smeltzek@eecs.oregonstate.edu

Keying Xu
Oregon State University
xuke@eecs.oregonstate.edu

***Abstract*—We present a visual notation to support the understanding and reasoning about program edits. The graph-based representation directly supports a number of editing operations beyond those offered by a typical, linear program-edit model and makes obvious otherwise hidden states the code can reach through selectively undoing or redoing changes.**

## I. Introduction

Many programming tools such as editors and version control systems (VCSs) model program edits as a linear series of changes, a view which simplifies common operations such as undo and redo and makes tool support easier to implement. However, such a model is intrinsically limited in its inability to express a number of desirable editing operations. Most notable among these is the notion of a *selective undo* [1]. Undo is a widely studied editing operation with a number of proposed models [2], [3], of which selective undo is one. Unlike traditional linear undo, selective undo allows users to undo previous actions without first undoing later ones. This encourages the development a model of program edits which provides support for non-linear, selective undo in a systematic way, including the handling of dependencies among edits. Other operations which are complicated through the use of a linear model are omitted here for space.

In addition to modeling individual operations, the analysis of source code edit histories in gerenal has proved fruitful in a number of areas [4], [5], [6], [7], which helps to justify further study of program edit models. Much of this work assumes that a program edit is analogous to a commit in a version control system. More recent work, however, has found this granularity to be too coarse and imprecise [8], [9]. This suggests the need for a model with arbitrary granularity, in which edits can be anything from individual keystrokes to rewrites of large code sections.

To model program edit histories in support of analysis and reasoning, as well as to support non-linear editing operations, we propose a compositional notation for program edits which handles branching and sharing of program parts explicitly and systematically, called *program edit graphs* (*PEGs*). PEGs make the following primary contributions:

- Explicitly reveal all program variants.
- Show dependencies among edits
- Serve as a semantic basis for selective undo.
- Support partial undo operations.

[†]This work is supported by the National Science Foundation under the grants CCF-1219165 and IIS-1314384.

## II. Program Edit Scenario

Consider the following editing scenario, which serves as a running example for future discussion. A programmer has written a simple function definition.

```
int f(int a){int b; return a+b}
```

The programmer then renames the function parameter from *a* to *c* and obtains the following code. We refer to this change as edit *A*.

```
int f(int c){int b; return c+b}
```

Next she elects to rename the local variable from *b* to *c*. However, since *c* already exists as the function parameter, she must rename that parameter again, say to *d*, in order to avoid shadowing it. This change, which we refer to as edit *B*, results in the following final program.

```
int f(int d){int c; return d+c}
```

Including the initial program, these two edits appear to have produced three program variants: one by applying no edits, one by applying only edit *A*, and one by applying *A* and then *B*. However, the edits have also produced a fourth, less obvious variant, which can be obtained by applying edit *B* without first applying *A*. This is functionally equivalent to a (partially) selective undo of edit *A*.

In order to reveal variants such as this, which are made inaccessible by a linear editing model, and to facilitate reasoning about them, we introduce the notion of a *program edit graph* (*PEG*). PEGs are annotated graphs in which the nodes represent program states or variants and directed edges represent the corresponding program edits. The example here uses single letter edit names for simplicity.

## III. Program Edit Graphs

Figure 1 shows two PEG representations for the preceding editing example. The factored representation, shown on the right, is used in the construction of the expanded version, and is discussed in Section IV. The expanded representation shows the original program as the source node, and the black path shows the program edits *A* and *B* leading to the latest variant, a sink. In green, it shows edits that deviate from the standard, linear path. Traversing an edge in a forward direction amounts to applying an edit while a backward direction indicates an undo.

This PEG also makes it clear that a linear undo history (represented by the black path) would not allow the *A* edit to

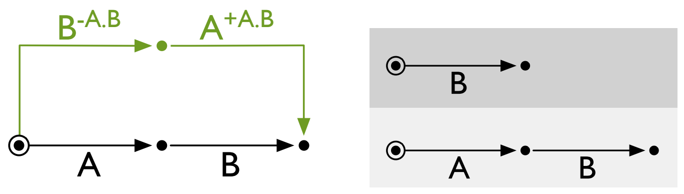


Fig. 1. Expanded (left) and factored (right) PEG for the edits of function f.

be undone without first undoing $B$ since both edges would need to be traversed backwards. However, the expanded PEG shows that a (partially) selective undo of $A$ is possible without first undoing $B$, by traversing the edge $A^{+A.B}$ backwards. Despite only three variants being obvious from the initial edit story, the PEG representation clearly shows this hidden fourth variant.

The superscript notation indicates the dependencies between edits, and therefore also the partial nature of certain operations. In the original edit history, part of $B$ was dependent on $A$. Thus the edge label $B^{-A.B}$ represents only the independent part of edit $B$, without the part that depends on edit $A$. Conversely, the label $A^{+A.B}$ indicates that traversing that corresponding edge applies edit $A$ as well as that part of edit $B$ which is dependent on $A$. This means that, were the programmer to selectively undo $A$, it would necessitate undoing the part of $B$ that is dependent on $A$ as well. This is called a *partially* selective undo.

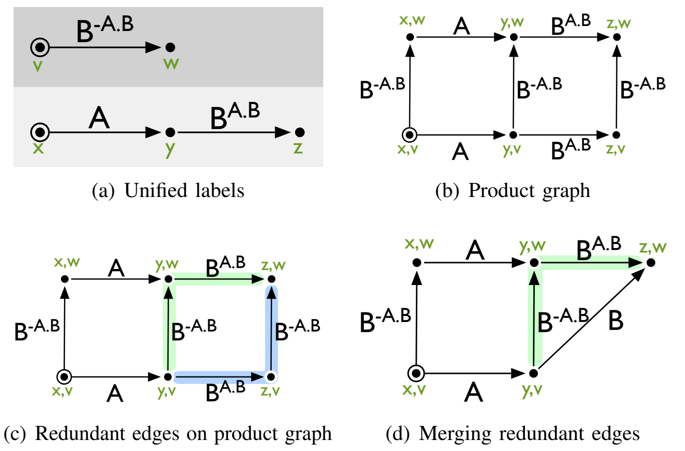
This PEG also illustrates a valuable property, namely that edits are commutable. For any two paths that start at the root of a PEG and end in a common target node represent the same edits. This commutability critically provides the semantic basis for selective undo since it allows the traversal of multiple paths representing the same program edits but with different sequences.

Finally, PEGs identify partial undo operations. Any partially selective undo must necessarily traverse an edge in the PEG which is annotated with a superscript. This provides programmers more information about which edits have been triggered or are lacking, and generally offers more options for navigating the edit space.

## IV. PEG Construction

PEGs can be constructed algorithmically in three main steps, as illustrated in Figure 2. First the factored representation is generated. Each factor corresponds to an independent, top-level edit. Any dependencies between edits results in them being chained together in the corresponding graph, while branching edits introduced through deletions or undos are captured by branching graphs. Figure 2(a) shows the factors for the preceding edit scenario. The top factor shows the independent part of the $B$ edit, which involved renaming the local variable. The bottom factor shows two edits. First the $A$ edit, which is independent, is applied, followed by the remaining portion of the $B$ edit which is dependent on $A$.

The second step is to produce a Cartesian product graph of all the individual factors. This will produce a node for every possible combination of nodes in the factors, with edges between nodes only if there is a corresponding edge in one of the factors. A product graph is shown in Figure 2(b).



(a) Unified labels (b) Product graph (c) Redundant edges on product graph (d) Merging redundant edges

Fig. 2. Generating a PEG by creating and simplifying a product graph of factors. Node labels are for clarity.

The product graph will frequently include redundancies such as paths along which multiple edges have the same label. Because of this, the graph must be simplified by merging edges, as illustrated in Figures 2(c) and 2(d). Once all redundancies have been removed, the result is the final, expanded PEG.

## V. Conclusions

We have demonstrated program edit graphs as a notation to formally capture program edits and operations. PEGs reveal non-obvious program variants and support a general form of selective undo. The commutable edits and explicit edit dependency information provide additional tools for programmers to reason about edits. Finally, because PEGs can be constructed algorithmically, they could be integrated into existing programming tools.

## References

[1] T. Berlage, "A selective undo mechanism for graphical user interfaces based on command objects," *ACM Trans. Comput.-Hum. Interact.*, vol. 1, no. 3, pp. 269–294, 1994.

[2] A. G. Cass and C. S. Fernandes, "Using task models for cascading selective undo," in *Task Models and Diagrams for Users Interface Design*, 2007, pp. 186–201.

[3] B. Shao, D. Li, and N. Gu, "An algorithm for selective undo of any operation in collaborative applications," in *ACM Int. Conf. on Supporting Group Work*, 2010, pp. 131–140.

[4] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *European Conf. on Object-Oriented Programming*, 2006, pp. 404–428.

[5] T. Graves, A. Karr, J. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. on Software Engineering*, vol. 26, no. 7, pp. 653–661, 2000.

[6] B. Livshits and T. Zimmermann, "Dynamine: Finding common error patterns by mining software revision histories," in *ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, 2005, pp. 296–305.

[7] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Trans. on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.

[8] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig, "Is it dangerous to use version control histories to study source code evolution?" in *European Conf. on Object-Oriented Programming*, 2012, pp. 79–103.

[9] Y. Yoon and B. A. Myers, "Capturing and analyzing low-level events from the code editor," in *ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, 2011, pp. 25–30.