FLEXEME: Untangling Commits Using Lexical Flows

Profir-Petru Pârţachi
profir-petru.partachi.16@ucl.ac.uk
University College London
London, United Kingdom

Santanu Kumar Dash
s.dash@surrey.ac.uk
University of Surrey
Guildford, Surrey, United Kingdom

Miltiadis Allamanis
miallama@microsoft.com
Microsoft Research
Cambridge, Cambridgeshire, United Kingdom

Earl T. Barr
e.barr@ucl.ac.uk
University College London
London, United Kingdom

ABSTRACT

Today, most developers bundle changes into commits that they submit to a shared code repository. Tangled commits intermix distinct concerns, such as a bug fix and a new feature. They cause issues for developers, reviewers, and researchers alike: they restrict the usability of tools such as git bisect, make patch comprehension more difficult, and force researchers who mine software repositories to contend with noise. We present a novel data structure, the δ-NFG, a multiversion Program Dependency Graph augmented with name flows. A δ-NFG directly and simultaneously encodes different program versions, thereby capturing commits, and annotates data flow edges with the names/lexemes that flow across them. Our technique, FLEXEME, builds a δ-NFG from commits, then applies Agglomerative Clustering using Graph Similarity to that δ-NFG to untangle its commits. At the untangling task on a C# corpus, our implementation, HEDDLE, improves the state-of-the-art on accuracy by 0.14, achieving 0.81, in a fraction of the time: HEDDLE is 32 times faster than the previous state-of-the-art.

CCS CONCEPTS

• **Software and its engineering** → **Software version control**; • **Mathematics of computing** → *Graph algorithms*; • **Computing methodologies** → *Kernel methods*; • **General and reference** → **General conference proceedings**.

KEYWORDS

graph kernels, clustering, commint untangling

ACM Reference Format:
Profir-Petru Pârţachi, Santanu Kumar Dash, Miltiadis Allamanis, and Earl T. Barr. 2020. FLEXEME: Untangling Commits Using Lexical Flows. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3409693

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3409693

1 INTRODUCTION

Separation of concerns is fundamental to managing complexity. Ideally, a commit to code repositories obeys this principle and focuses on a single concern. However, in practice, many commits tangle concerns [2, 6]. Time pressure is one reason. Another is that the boundaries between concerns are often unclear. Murphy-Hill *et al.* [21] found that refactoring tasks are often committed together with code for other tasks and that even multiple bug fixes are committed together.

Tangled commits introduce multiple problems. They make searching for fault inducing commits imprecise. Tao *et al.* [27] found that tangled changesets (commits) hamper comprehension and that developers need untangling (changeset decomposition) tools. Barnett *et al.* [2] confirmed this need. Herzig *et al.* [6, 7] studied the bias tangled commits introduce to classification and regression tasks that use version histories. They found that up to 15% of bug fixes in Java systems consisted of multiple fixes in a single commit. They also found that using a tangled version history significantly impacts regression model accuracy. In short, tangled commits harm developer productivity two ways: directly, when a developer must search a version history and indirectly by slowing the creation of tools that exploit version histories.

Version histories permit a multiversion view of code, one in which multiple versions of the code co-exist simultaneously. Le *et al.* built on principles described by Kim and Notkin [12] for multiversion analysis: they constructed a multiversion intraprocedural control flow graph and used it to determine whether a commit fixes all *n* versions [16]. This task is important when multiple versions are active, as in software product lines, and the patch fixes a vulnerability. Inspired by Le *et al.*, we define a δ-PDG, a multiversion program dependence graph (PDG), a graph that combines a program's data and control graphs [5].

We hypothesise that identifiers differentiate concerns. We harvest names that are used together in a program's execution, as in this statement «`takehome:=tax * salary`» to augment our δ-PDG and produce a δ-NFG. A desirable property of our δ-NFG is its modularity. It allows projecting any combination of data, control, or lexeme. Consequently, we could effortlessly reproduce Barnett *et al.*'s and Herzig *et al.*'s methods [2, 6] to explore the design space in tooling for concern separation in commits.

We introduce FLEXEME, a novel approach to concern separation that uses the δ-NFG. We group edits into concerns using the graph similarity of their neighbourhoods. We base this on the intuition that nodes are defined by the company they keep (their neighbourhoods) and cluster them accordingly. Thus, we reduce the concern separation problem to a graph clustering task. For clustering, we start by considering each edit in a commit as an separate concern, then use graph similarity to agglomeratively cluster them. We compute this similarity using the Weisfeiler-Lehman Graph Kernel [25].

We realised Flexeme in a tool we call Heddle[1]. Developers can run Heddle to detect tangled commits prior to pushing them or reviewers can use it to ask developers to untangle commits before branch promotion. We show that heddle improves the state-of-the-art accuracy by 0.14, and run-time by 32 times or 3′10′′. We also demonstrate the utility and expressivity of our δ-PDG construct by adapting Herzig *et al.*'s confidence voters (CV) to use the δ-PDG rather than diff-regions; the resulting novel combination, which we call δ-PDG+CV improves the performance and lowers the run-time of Herzig *et al.*'s unmodified approach.

In summary, we present

(1) Two novel data-structures, the δ-PDG, a multiversion program dependence graph, and δ-NFG, which augments a δ-PDG with lexemes;
(2) The Flexeme approach for untangling commits that builds a δ-NFG from a version history then uses graph similarity and agglomerative clustering to segment it; and
(3) Heddle, a tool that realises Flexeme and advances the state of art in commit untangling in both accuracy and run-time.

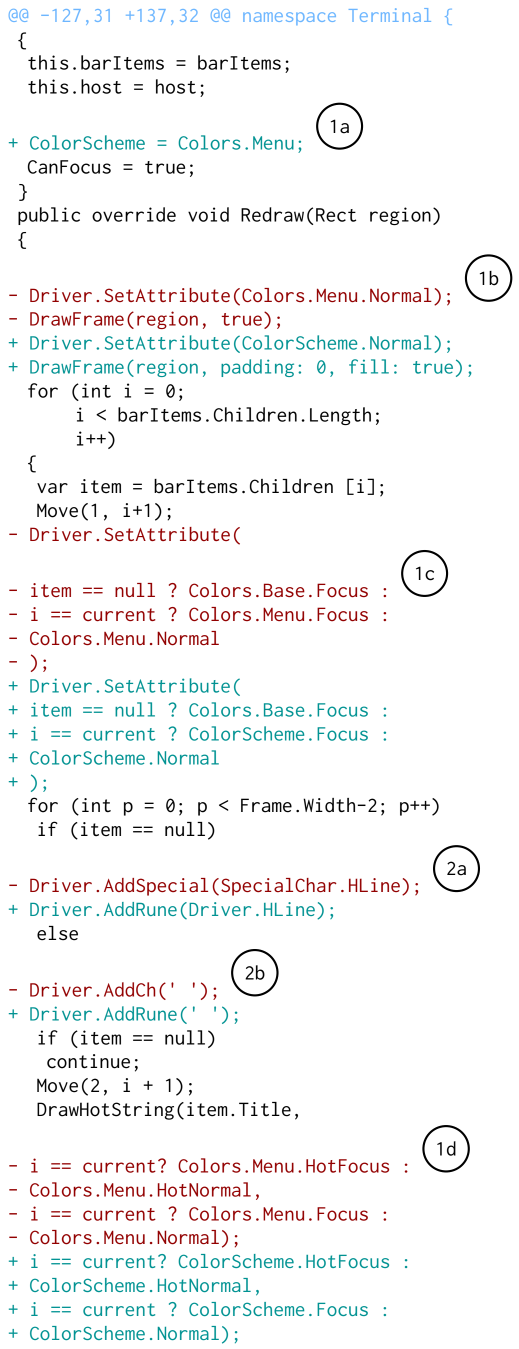
All of the tooling and artefacts needed to reproduce this work are available at https://pppi.github.io/Flexeme/.

2 EXAMPLE

Localising a bug with git bisect to determine a bug inducing commit, or reviewing a changeset during code review in presence of tangled commits can make the task unnecessarily difficult or even impossible [27]. Further, as Herzig et al. [6] found, tangled commits have a statistically significant impact on the performance of regressions methods used for defect prediction. Barnett et al. [2], determined that developer productivity benefits from tools that can propose changeset decompositions. In light of this, it is natural to ask how do different code entities co-occur within a concern?

The code entities of a concern tend to be in close proximity with each other in both the control- and data-flow graphs. Barnett et al. [2] exploited this by using def-use chains, which offer a short-range view of these connections. However, as shown in Figure 1, connectivity through the data-flow graphs on its own is insufficient to demarcate concerns. Indeed, this observation is reflected in the relatively lower accuracy rates on concern separation reported in Barnett et al. [2] compared to Herzig et al. [6], the concerns 1 and 2 consisting of the hunks 1a-1d and 2a-2b, respectively, could be conflated as they are method invocations of the same `Driver` class. The conflation might occur even though hunks 1b, 1c and 1d are connected via the use of the `Colors.Menu` and `ColorScheme` which provides counterweight to conflating concerns 1 and 2.

Control-flow can help delineate regions or constructs that handle specific types of concerns. For example, a loop could be performing



```
@@ -127,31 +137,32 @@ namespace Terminal {
 {
  this.barItems = barItems;
  this.host = host;

+ ColorScheme = Colors.Menu;
  CanFocus = true;
 }
 public override void Redraw(Rect region)
 {

- Driver.SetAttribute(Colors.Menu.Normal);
- DrawFrame(region, true);
+ Driver.SetAttribute(ColorScheme.Normal);
+ DrawFrame(region, padding: 0, fill: true);
  for (int i = 0;
       i < barItems.Children.Length;
       i++)
  {
   var item = barItems.Children [i];
   Move(1, i+1);
- Driver.SetAttribute(

- item == null ? Colors.Base.Focus :
- i == current ? Colors.Menu.Focus :
- Colors.Menu.Normal
- );
+ Driver.SetAttribute(
+ item == null ? Colors.Base.Focus :
+ i == current ? ColorScheme.Focus :
+ ColorScheme.Normal
+ );
  for (int p = 0; p < Frame.Width-2; p++)
   if (item == null)

- Driver.AddSpecial(SpecialChar.HLine);
+ Driver.AddRune(Driver.HLine);
   else

- Driver.AddCh(' ');
+ Driver.AddRune(' ');
   if (item == null)
    continue;
   Move(2, i + 1);
   DrawHotString(item.Title,

- i == current? Colors.Menu.HotFocus :
- Colors.Menu.HotNormal,
- i == current ? Colors.Menu.Focus :
- Colors.Menu.Normal);
+ i == current? ColorScheme.HotFocus :
+ ColorScheme.HotNormal,
+ i == current ? ColorScheme.Focus :
+ ColorScheme.Normal);
```

Figure 1: A diff with two tangled concerns: (a) the change of the drawing API (all other changes) and (b) the migration from using chars and special chars to runes (the two changes related to `AddRune`). Attempting to disentangle this diff with state-of-the-art tools relying on DU-chains fails because the tangled changes are connected in the def-use chain pertaining to `Driver` and are in close proximity in the file. Using a PDG allows us to additionally exploit control flow information to aid untangling.

[1] A heddle is wire or cords with eyelets that hold warp yarns in a place in a loom. While it does not untangle, a heddle prevents tangles, so we have named our tangle-preventing tool after it.

a specialised computation that forms a single concern on its own. We see an example of such a loop in Figure 1. The for loop captures the process by which a screen line is generated and is strongly related to how on screen characters are handled (1a and 1b). This suggests that using a Program Dependency Graph (PDG), which encapsulates both control- and data-flow, as a basis for performing commit untangling overcomes some of the shortcomings of using the data-flow alone. The PDG provides evidence that 1a and 1b could be a part of the same concern because of control-flow. Additionally, the PDG also tells us that 1c and 1d could be a part of the same concern by virtue of data-flow through `Colors.Menu` and `ColorScheme`. However, it may be observed that the link with 2a-2b is still strong due to flows through `Driver`.

Lexemes in the two concerns in Figure 1 provide strong evidence for their separation. While concern 1 uses `Set*` methods in `Driver`, concern 2 uses `Add*` methods. This evidence is missed by PDGs which discard lexical information. Developers tend to use dissimilar names for different tasks. Dash et al. [3] leveraged this observation to augment data-flow with lexical information to successfully identify type refinements. In our work, we take a similar approach and use lexical information to separate concerns. Our approach to introducing lexemes in our PDG representation is similar to the *name-flows* construct of Dash et al. [3]. While they augmented data-flow with lexemes, we augment PDGs and a description of how we achieve this follows.

3 CONCERNS AS LEXICAL COMMUNITIES

Given consecutive versions of some code, Flexeme constructs their PDGs and overlays them adding name-flows [3] to build a δ-NFG. We feed the δ-NFG to a graph clustering procedure to reconstruct atomic commits. A δ-NFG naturally captures flows that bind concerns together such as data- and control-flows. Additionally, it also captures natural correlation in names that developers choose when addressing a given concern.

Figure 2 overviews how flexeme constructs and uses a δ-NFG. For a sequence of contiguous versions, we generate a PDG first with the help of a compiler. We then combine these PDGs to form a multi-version PDG which we call a δ-PDG. We then decorate the δ-PDG with version specific name-flow information to obtain a δ-NFG. We feed the δ-NFG to agglomerative clustering. We use graph similarity to separate concerns across the original set of contiguous versions. We discuss the details of the δ-NFG construction in Section 3.1, Section 3.2 and Section 3.3. We discuss the graph clustering approach in Section 3.4.

3.1 Multi-Version Name Flow Graphs

We now formally define the PDG, δ-PDG and δ-NFG in order to bootstrap discussion on the δ-NFG construction.

Definition 3.1. *Program Dependency Graph (PDG).* Flexeme's PDG is a directed graph with node set N and edge set E *s.t.* each node $n \in N$ is annotated with either a program statement or a conditional expression; each edge $e \in E$ has an optional annotation representing the name or the data that flows along it, and a kind that describes the relationship type: data or control.

Definition 3.2. *Multi-version Program Dependency Graph (δ-PDG).* A δ-PDG ^{p,q} is the disjoint union of all nodes and edges across all versions in $[p, q]$. δ-PDG ^{p,p} is the PDG at version p.

Definition 3.3. *Name Flow Graph (NFG).* Flexeme's NFG is a standard dataflow graph $G = (N, E)$, augmented with name flows, the raw lexemes of the literals and identifiers in a program text that originate at some node flow across edges and collect in downstream nodes. A name flow labels an edge with those lexemes that flow across it and a node with those that either originate at it or flow into it.

We re-use NFG from Dash *et al.*'s Refinym [3]. To a first approximation, lexemes flow along def-use chains and collect in the variables on the LHS of assignments.

Definition 3.4. *Multi-version Name Flow Graphs (δ-NFG).* A δ-NFG ^{p,q} is the δ-PDG ^{p,q}, augmented with name flows: If an edge exists in both the PDG and the NFG of a version, we augment the PDG edge with the corresponding name flow.

Inspired by Le and Pattison [16]'s multiversion intraprocedural graphs, we are the first to propose and construct δ-PDGs. To construct a δ-PDG, we start from the initial version considered. For each subsequent version, we make use of line-span information in the PDG and UNIX diff on the source-files to determine changed and unchanged nodes, making Flexeme language agnostic. Changed nodes are introduced to the δ-PDG as they appear in the new version. δ-PDGs retain nodes deleted across the versions a δ-PDG spans. Deletion becomes a label. We match unchanged nodes between the nodes of the δ-PDG and the new version. To match, we use string similarity to filter candidates and we use line-span proximity to rank them (Section 3.2). For nodes the new version deletes, we backpropagate the delete label to edges flowing into them. To add edges, we consider all unmatched edges in the new version, then match their source and target either to existing δ-PDG nodes, or, when a either the source or target does not exist in input δ-PDG, match it to a fresh node (Section 3.3). Finally, to obtain a δ-NFG, we endow the δ-PDG with name flow information for each of the versions considered by matching nodes using their line-spans.

3.2 Anchoring Nodes Across Versions

To integrate a fresh PDG G^j, we start with the patch P_{ij}. We view each hunk $h \in P_{ij}$ as a pair of snippets (s_i^-, s_j^+) where version i deletes s_i^- and version j adds s_j^+. Snippets s_i^- and s_j^+ do not exist for hunks that only add or only delete; ϕ represents these patches. Accounting for s_i^- is straightforward: we do not update the nodes in the G^{1-i} that fall into s_i^-. Accounting for s_j^+ is non-trivial; much like patching utilities, we need a notion of context.

For all s_j^+, we introduce fresh nodes in the δ-PDG. However, we cannot anchor these nodes until we identify counterparts for nodes in G^j that were untouched by P_{ij}. Identifying untouched nodes in G^i is straightforward; we can simple check locations in P_{ij} against the location of each node in G^j. All touched nodes from G^j are treated as new added nodes and need to be introduced in the δ-PDG. Identifying the counterpart in G^i of an untouched node in G^j requires a notion of node equivalence.

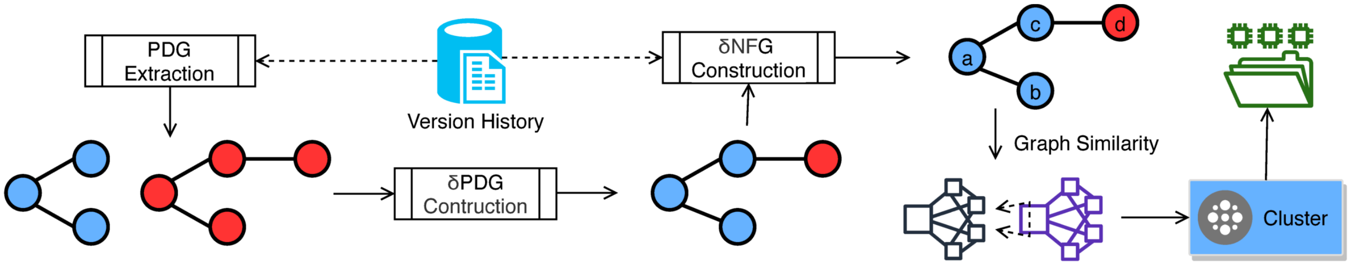


Figure 2: Overview of FLEXEME's δ-NFG construction and concern separation.

Definition 3.5. *Node Equivalence.* Given a node v_j in a G^j and a candidate node v_i in $G^{1,i}$,

$$v_j \equiv v_i \iff \forall (p,q) \in R(A[v_j], A[v_i]).F(p,q) \geq T$$

Here, $A[k]$ returns all nodes adjacent to the node k in a graph. R is a variant of the *Stable Roommates Problem* [9] where nodes drawn from $A[v_i]$ are matched with nodes drawn from $A[v_j]$. Each node in $A[v_j]$ has an affinity for nodes in $A[v_i]$ proportional to the similarity of lexemes at the nodes. The operator R considers these affinities and tries to match each node with the one it has the highest affinity for. By construction, R always returns a one-to-one match even though two different nodes a and b may have a strong affinity to a third node c. In such a case, either a or b will be matched with c but not both; the one that is not matched to its highest preference (defined in terms of affinity) is matched to the next node from its preference list. We further require that lexical similarity, computed by the function F, is above the threshold T. This lets us control the level of fuzziness while matching nodes. In this work, we have chosen F to be string edit distance and set $T = 1.0$ thus requiring exact matches.

3.3 Integrating Nodes Across Versions

Once counterpart nodes are identified using a notion of node equivalence, the next task is to store the location information for the untouched nodes in G^j at theirs counterpart and mark the location with version j. Finally we add all the nodes P_{ij} adds to the δ-PDG and create edges between them and the counterparts of their parents and children.

We demonstrate the δ-PDG construction in Figure 3. We show the PDG for two versions of an application — 1 and 2. Since version 1 is our initial version in this example, it is also our δ-PDG G^1. We have omitted the data-flows in the PDG for brevity. Each node in the PDG contains a list of location-version tuple; the location information has been supressed for simplicity. The patch above the two PDGs is the *diff* of the two versions. We have two snippets in the patch: s_1^- for the snippet that is to be deleted in 1 and s_2^+ the snippet that is to be added in 2. Accounting for the deleted line comes for free as we store locations for snippets across versions. All we need to do is to check the location information for s_2^- against the span of the nodes in the G^1. For s_2^+, we need to search for equivalent nodes across the two versions. We perform fuzzy matching on lexemes in nodes shortlisted using location information as detailed above. In the case of s_2^+, we identify the call expressions `Move` and `Driver.AddCh(...)` as parents and children, respectively, for s_2^+. Merging s_2^+ into the δ-PDG shown on the right is then a straightforward task of drawing edges between nodes and their parents/children. Once we obtain this δ-PDG representation, we perform the untangling task in a reconstructive manner.

3.4 Identifying Concerns

We start by assuming each change is atomic, and iteratively merge changes that are similar enough. At a high-level, we expect similar nodes to have similar "neighbourhoods". To measure this, we build the k-hop neighbourhood[2], for each node. We then cluster by similarity of these neighbourhoods. To compute graph similarity, we use the Weisfeiler-Lehman graph kernel [25] which builds on top of the Weisfeiler-Lehman graph isomorphism test [30]. For a pair of graphs, the test iteratively generates multi-set labels. When two graphs are isomorphic, then all the sets are identical.

Formally, let the initial vertex labelling function of the graph be $l_0 : V(G) \to \mathcal{L}$, where $\mathcal{L}$ is the space of all node labels. At step i, let $l_i(v) = \{\{l_{i-1}(v')\} \mid v' \in N(v)\}$, where $N(v)$ is the set of neighbours of vertex v. At each iteration i, this process labels the node v with a set comprising the labels of all of v's neighbours. This set becomes the new label of that node. Since isomorphism testing can diverge, we bound it to n iterations and obtain the sequence $\langle G_0, G_1, \ldots, G_n \rangle$.

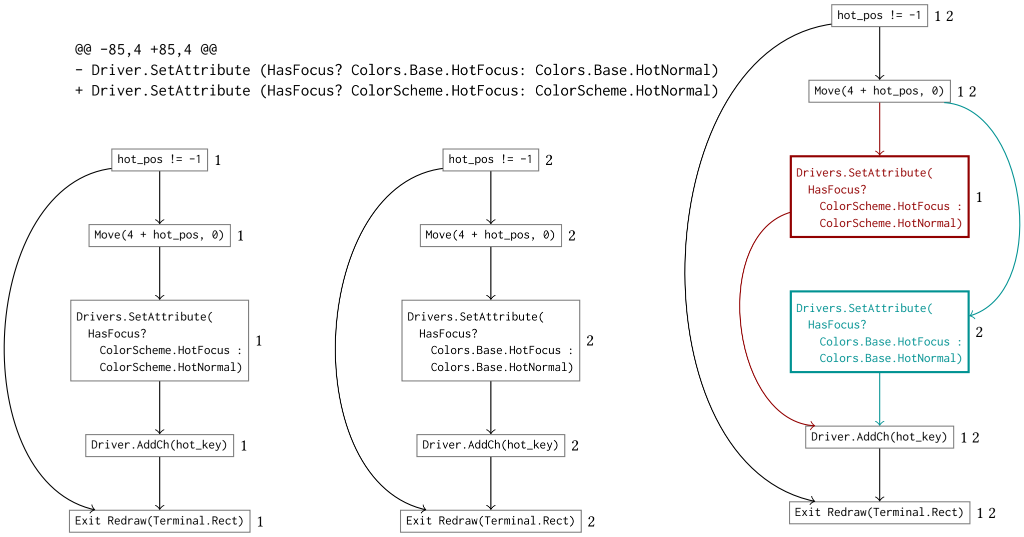
A positive semi-definite kernel on the non-empty set X is a symmetric function

$$\begin{aligned} &k : X \times X \to \mathbb{R} \\ &s.t. \sum_{i=1}^{n} \sum_{j=1}^{n} c_i c_j k(x_i, x_j) \geq 0, \\ &\forall n \in \mathbb{N}, x_1, \ldots, x_n \in X, c_1, \ldots, c_n \in \mathbb{R}. \end{aligned}$$

This function can take the arguments in either order and, for any parametrization by real constants, has non-negative weighted sum over all inputs. A graph kernel is a positive semi-definite kernel on a set of graphs. When a kernel takes a set of graphs ($\mathcal{G}$) as input, we define the $K(\mathcal{G})_{ij} = k(G_i, G_j), G_i, G_j \in \mathcal{G}$ to compute the matrix of pairwise kernel values.

Let $\mathcal{G}$ be the set of graphs over which we wish to compute graph based similarity, and let $K : \mathcal{G} \to \mathbb{R}^{|\mathcal{G}|} \times \mathbb{R}^{|\mathcal{G}|}$ be a graph kernel. Then the WL Graph Kernel becomes: $K_{WL}(\mathcal{G}) = K(\mathcal{G}_0) + K(\mathcal{G}_1) + \ldots + K(\mathcal{G}_n)$;, where $\langle \mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_n \rangle$ is obtained by applying n steps of the isomorphism test to each graph in $\mathcal{G}$

[2] In our experiments we consider the $k = 1$-hop neighbourhoods.



```
@@ -85,4 +85,4 @@
- Driver.SetAttribute (HasFocus? Colors.Base.HotFocus: Colors.Base.HotNormal)
+ Driver.SetAttribute (HasFocus? ColorScheme.HotFocus: ColorScheme.HotNormal)
```

hot_pos != -1 1
Move(4 + hot_pos, 0) 1
Drivers.SetAttribute(HasFocus? ColorScheme.HotFocus : ColorScheme.HotNormal) 1
Driver.AddCh(hot_key) 1
Exit Redraw(Terminal.Rect) 1
hot_pos != -1 2
Move(4 + hot_pos, 0) 2
Drivers.SetAttribute(HasFocus? Colors.Base.HotFocus : Colors.Base.HotNormal) 2
Driver.AddCh(hot_key) 2
Exit Redraw(Terminal.Rect) 2
hot_pos != -1 1 2
Move(4 + hot_pos, 0) 1 2
Drivers.SetAttribute(HasFocus? ColorScheme.HotFocus : ColorScheme.HotNormal) 1
Drivers.SetAttribute(HasFocus? Colors.Base.HotFocus : Colors.Base.HotNormal) 2
Driver.AddCh(hot_key) 1 2
Exit Redraw(Terminal.Rect) 1 2

Figure 3: Construction of a δ-PDG from two versions — 1 and 2 — of a program. Version 2 is obtained from version 1 by application of the patch shown in the program. Each node is annotated with the version number it is present in.

The WL Graph Kernel K is a meta-kernel that extends an underlying kernel. We want a Subtree WL Graph Kernel that counts the number of identical rooted subtrees for each node in the graph of the same depth as the iteration. In our case, this maps to identical downstream behaviour from each node in terms of each of the flows considered. To achieve this behaviour, we set K to be the Vertex Label Histogram Kernel and encode outgoing flow types in the label function. This kernel is defined as follows: Let Ψ be a function that embeds the graph into a vector space, often called a feature map in literature, and let $\langle \cdot, \cdot \rangle$ be the inner product, then

$$\Psi(G) = \mathbf{f};$$

$$\mathbf{f}_i = |\{v \mid v \in V(G), l_i \in \mathcal{L}, lv(v) = l_i\}|\,;$$

$$K(\mathcal{G})_{ij} = \langle \Psi(G_i), \Psi(G_j) \rangle;$$

For clustering, we opt for agglomerative clustering, like Herzig et al. [6], Herzig and Zeller [7]. With node-neighbourhood pairwise similarity information, we can build an affinity, *i.e.* a pair-wise distance, matrix for clustering trivially by simply inverting the value, *i.e.* 1 − similarity. Section 4.2 details the implementation.

4 HEDDLE

Heddle realises Flexeme. Heddle first constructs a δ-PDG for each input file, and combines them into a δ-NFG. Heddle then decomposes the δ-NFG into a forest, and uses graph kernels to compute distance matrices which we input into agglomerative clustering. We close by describing how a project could adopt Heddle.

4.1 δ-PDG Construction

We implement both name flow extraction and PDG extraction in Roslyn [20], the open-source compiler for C# and Visual Basic from Microsoft. We store the PDG in GraphViz Dot format. We then implement the PDG merging procedure as described above over the dot files. This allows us to reuse the merging procedure; it is language agnostic. One need only provide PDGs (and optionally name flows) as 'dot' files. To enable the merging process, we store the origin line-span[3] and method membership information in the nodes along with the usual data associated with such graphs, *i.e.* expression information and edge kind. To obtain textual diffs needed for δ-NFG construction, we make use of the UNIX 'diff' tool.

By default, Heddle constructs one δ-NFGs per file. To mitigate the problems cross-file dependencies cause and reduce the cost of untangling, Heddle merges all graphs associated with a commit, into a single structure. This merge uses node equivalence (Definition 3.5) when operating on files that share a namespace. A key difference in the same-version, cross-file setting is that we need to copy over both types of changed nodes and add the edges similar to the added nodes scenario described in Section 3.

To simplify our code, our implementation of PDG extraction does not consider goto statements in the Control Flow Graph; this does not matter much in practice, as goto statements are very rare in our corpus.

4.2 Graph Node Clustering

We use GraKeL [26] for the Weisfeiler-Lehman (WL) Graph Kernel [25] implementation and leave the number of iterations of the isomorphism tests at the library's default of 10. We set the underlying kernel to Vertex Histogram Graph Kernel to obtain the same behaviour as the Subtree WL Graph Kernel.

For agglomerative clustering, we use SciPy [10]. We precompute the affinity matrix by using the WL kernel similarity. We call the clustering method with the linkage parameter set to "complete", which mimics the behaviour described in Herzig *et al.* [7], *i.e.* when two groups are merged, the maximal distance from any member of the group to any other group is kept as the new distance. We stop merging groups when they are less than 0.5 similar to any other group instead of providing oracle information to the method. This

[3] We consider code snippets at line granularity.

models the fact that, in practice, developers do not know how many concerns a commit contains.

Code for both δ-PDG construction and node clustering is available online[4].

4.3 Deployability

Heddle takes ten seconds on average to untangle a commit (Section 6.2), and 45s on average to construct and merge the PDG for a commit into a δ-PDG. This is beyond the one second limit suggested by Nielsen [22] for processes that allow a user to feel like operating directly on data, which suggests that our tools should be onboarded into a process that is out-of-band with regards to developer attention. An example of such a process is the build automation within continuous deployment. Heddle could be added as an additional pass at the end of the build process providing an untangle report for the code reviewers, ready to be inspected when the review process starts. Further, on-boarding Heddle in such a manner makes it independent of the workflow and tooling choices made by the developers; the system would only need to be deployed on the build servers. The report provided would allow reviewers to better focus on the different parts of the patch and aid patch comprehension. There is an initial onboarding cost requiring generating NFGs for all source files in the code-base. However, our construction is incremental and for any fresh patch that needs integration into the δ-NFG, we only need to consider the files touched by the patch.

5 EXPERIMENTAL DESIGN

In this section, we discuss how we constructed a dataset, measure untangling performance and our reproduction of two baseline methods.

5.1 Corpus Construction

To construct our corpus, we reuse Herzig et al. [6] methodology who artificially tangle atomic commits. Therefore, we consider commits that:

1. Have been committed by the same developer within 14 days of each other with no other commit by the same developer in between them.
2. Change namespaces whose names have a large prefix match.
3. Contain files that are frequently changed together.
4. Do not contain certain keywords (such as 'fix', 'bug', 'feature', 'implement') multiple times.

The first criterion mimics the process by which a developer forgets to commit their working directory before picking up a new task. The next criterion is an adaptation of Herzig *et al.*'s 'Change close packages' criterion to the C# environment. The third considers files that are coupled in the version history, thus creating a tangled commit not too dissimilar from commits that naturally occurred. The intuition being that if commit A touches file f_A and commit B touches file f_B, *s.t.* f_A and f_B are frequently changed before(coupling) [31], then A and B should be tangled. The final criterion is a heuristic to ensure that we do not consider tangling commits that we are certain are not atomic. We add this condition to mitigate the problem of tangling actually tangled commits which would cause an issue when computing ground truth. Overall, this artificially created corpus mimics some of the tangled commits we expect developers to make; specifically, it captures the intuition of a developer committing multiple consecutive work units as a single patch. Section 7 discusses the threat this poses to Heddle's validity.

Table 1: Project statistics. The last revision indicates the commit at which we performed the 'git clone'.

Project	LOC	# of Commits	Last revision
Commandline	11602	1556	67f77e1
CommonMark	14613	418	f3d5453
Hangfire	40263	2889	175207c
Humanizer	56357	1647	604ebcc
Lean	242974	7086	71bc0fa
Nancy	79192	5497	dbdbe94
Newtonsoft.Json	71704	299	4f8832a
Ninject	13656	784	6a7ed2b
RestSharp	16233	1440	b52b9be

Following the above procedure, we obtain a shortlist of chains of SHAs of varying length for nine C# systems; we show project statistics in Table 1. These SHAs refer to atomic commits. We sanity check that they are atomic by uniformly sampling 30 commits from our corpus and examining each commit for up to five minutes. We found 27/30 to be atomic, 2 of the tangled commits refactor comments (which are invisible and therefore atomic to Heddle), and 1 tangled commit due to merging content from a different versioning system (SVN) in a single commit. From this study, we extracted two heuristics that we used to filter out non-atomic commits. Specifically, we excluded all merge commits and those that generate δ-PDGs that have no changed nodes.

We attempt to create tangled commits by selecting the SHAs in the tail of these chains and git cherry-picking them onto the head. We then mark the originating commit in the tangled diff using the individual atomic diffs as not all selections are successful. Some of the successful selections may not have changes from all tangled commits as later commits may shadow them. Therefore, we perform a final pass to learn the actual number of surviving concerns. In the end, we built two sets of tangled commits: those that tangle 2 and those that tangle 3. This models the most common numbers of tangled concerns in the wild [6, Figure 7].

Table 2 shows the final statistics for our corpus, where the number of concerns is the count of surviving concerns at the end of the selection process. We report all successfully generated data-points and detail, in Section 6, the subsets on which we compared any two methods when at least one of them did not run on the full corpus due to time-outs. We do not treat time-outs as a zero accuracy result, but drop them from consideration. We remark that the primary source of time-outs is the computational cost of running our reproduction of Herzig *et al.*.

5.2 Experimental Setup

Our experiments assess how well our method recovers the original commits compared to the baseline methods proposed by Barnett *et al.* [2] and Herzig *et al.* [6]. Additionally, we measure the runtime cost of the different methods. For this, all methods are run

[4] https://pppi.github.io/Flexeme/

Table 2: Successfully tangled commits.

Project	Concerns		
	2	3	Overall
Commandline	308	32	340
CommonMark	52	0	52
Hangfire	229	87	316
Humanizer	85	4	89
Lean	154	24	178
Nancy	284	67	351
Newtonsoft.Json	84	7	91
Ninject	82	0	82
RestSharp	95	18	113
Overall	1373	239	1612

in isolation on the same high-end laptop (i7-8750H @ 3.20 GHz, 16 GB RAM @ 2666 MHz) and we compute accuracy for all methods as follows:

$$A = \frac{\#\text{Correctly labeled nodes}}{\#\text{Nodes in graph}}. \quad (1)$$

Both baselines, as well as Heddle, may recover an arbitrary permutation of the ground truth labels. To avoid artificially penalising them, we first use the Hungarian Algorithm [15] to find the permutation that maximises accuracy. Consider the ground truth '[01122]', should a tool output '[20011]', a naïve approach would award it 0.0 accuracy, while a trivial permutation of the labelling function reveals that this is indeed 1.0 accuracy. We report this maximal accuracy for each method.

For the purpose of timing, we perform one burn-in run of commit segmentation followed by 10 repeats that are used to compute the runtime cost. We then obtain the speed-up factor as a non-parametric pairwise comparison between the methods. Notably, we do not include the cost of the static analysis required for each method, rather only the cost of segmentation. This is due to deriving the required program representations for each method as a projection from the δ-NFG.

5.3 Reproducing Barnett et al. and Herzig et al.

In order to use Barnett *et al.*'s method as a baseline, we had to re-implement it, because its source is not public. Their method rests on def-use chains. They retain all def-use chains that intersect a diff hunk in a commit. To obtain this, we first recover the dataflow graph and then we separate the flows by 'kill' statements, such as assignments. In a δ-PDG, this becomes a def-use chain projected onto a diff hunk. Two chains are equivalent if (a) they are both changed and are both uses of the same definition or (b) they are a changed use of a changed definition. Under this partition, they divide the parts into trivial and nontrivial. All diff-regions in a trivial part fall within a single method. To avoid overwhelming developers with chains that are highly likely to be atomic, they do not show trivial parts; implicitly, they are assuming that developers can see and avoid method-granular tangling. In contrast, we, like Herzig *et al.*, consider method-granular tangling, so our re-implementation does not distinguish trivial and non-trivial parts.

To reproduce Herzig *et al.*'s method, we reconstruct the call graph by collapsing into hypernodes by method membership, we recover the dataflow from the δ-PDG, and we additionally generate an occurrence matrix specifying the files changed by a commit, as well as file sizes in terms of number of lines. Finally, we compute a diff-region granular corpus for all the successful tangles, as the Herzig *et al.* algorithm works at a diff-region granularity. Using this information, we construct a distance matrix for each tangled commit. This distance matrix is populated by the sum of the distances from each individual voter. All confidence voters are identical to the original paper with one exception. We replaced package distance by namespace distance; we do, however, compute it in the same manner. At evaluation time, we also provide the number of concerns to be untangled. This is known by construction, as in the original paper. We perform agglomerative clustering on the resultant matrix using complete linkage, *i.e.* taking the maximum distance over all diff-regions within a cluster.

We also create a version of Herzig *et al.*'s confidence voters method that operates directly on δ-PDGs, which we call δ-PDG+CV. The last stage here is not dissimilar to flexeme, with the remark that we still provide Herzig *et al.*'s approach with oracle access to the number of concerns while flexeme requires only a similarity threshold. We implemented the voters so that only file distance and change coupling require auxiliary information. This is pre-computed from the git history of the project under analysis. Every other voter — call graph distance, data dependency and namespace distance — are computed on demand only for the nodes that we consider for merging.

For both baselines, as well as Heddle, we measure only the time taken to untangle and not the construction of auxiliary structures. We exclude the construction time as we derive the DU-chains, call-graphs and dataflow-graphs from our δ-PDG.

6 RESULTS

In this section, we compare Heddle against our two baselines, in terms of accuracy and runtime. To implement our baselines, we reproduced the methodology and tooling from Herzig *et al.* [6] and Barnett *et al.* [2]. We show that Heddle outperforms, in both accuracy and run-time, our reproduction of Herzig *et al.*'s method. We report comparisons between tools only on the subset of datapoints on which both tools run to completion.

6.1 Untangling Accuracy

When recovering the original partition of the δ-NFG from our artificial tangle of code concerns, Heddle achieves a median accuracy of 0.81 and a high of 0.84 on the project Nancy; it outperforms Herzig *et al.* by 0.14 and trails δ-PDG+CV only by 0.02 while scaling better to big patches. Unlike Herzig *et al.*, Heddle achieves this result without resorting to heuristics or manual feature construction.

Heddle outperforms both baselines in accuracy, the difference being statistically significant at $p < 0.001$ (Wilcoxon pair-wise test), and matches the performance of δ-PDG+CV, the new technique we have built from grafting Herzig *et al.*'s confidence voters on top of our δ-NFG ($p = 0.76$, Wilcoxon pair-wise test). Unlike Heddle, the other approaches consider file-granular features. Specifically,

Table 3: Median performance of untangling commits for each method by project and number of tangled concerns. The performance differences are significant to $p < 0.001$ for all overall results according to a two-tailed Wilcoxon pair-wise test on the common set of data-points, except δ-PDG+CV vs heddle, which is significant to $p < 0.001$. Entries indicated by a '*' signify that there was no relevant data point to report the performance on and those indicated by 'x' indicate time-outs.

Project Name	Barnett *et al.* [2]			Herzig *et al.* [6]			δ-PDG+CV			Heddle (δ-NFG + WL)		
	2	3	Overall	2	3	Overall	2	3	Overall	2	3	Overall
Commandline	0.18	0.21	0.19	0.67	0.48	0.64	0.77	0.84	0.80	**0.82**	**0.92**	**0.82**
CommonMark	0.20	*	0.20	0.65	*	0.65	**0.90**	*	**0.90**	0.70	*	0.70
Hangfire	0.16	0.13	0.15	0.70	0.54	0.64	0.84	**0.88**	**0.87**	**0.86**	0.68	0.79
Humanizer	0.18	0.31	0.18	0.64	0.42	0.62	0.69	x	0.69	**0.83**	**0.57**	**0.81**
Lean	0.19	0.12	0.18	0.69	0.62	0.69	**0.84**	0.71	**0.84**	0.77	**0.82**	0.80
Nancy	0.09	0.08	0.09	0.70	0.56	0.67	**0.86**	0.80	**0.86**	0.81	**0.92**	0.84
Newtonsoft.Json	0.15	0.11	0.15	0.71	0.56	0.71	**0.86**	**0.69**	**0.82**	0.71	0.52	0.71
Ninject	0.14	*	0.14	0.57	*	0.57	**0.94**	*	**0.94**	0.80	*	0.80
RestSharp	0.12	0.14	0.12	0.71	0.69	0.70	0.74	0.53	0.70	**0.82**	**0.89**	**0.82**
Overall	0.14	0.11	0.13	0.69	0.62	0.67	**0.83**	**0.84**	**0.83**	0.81	**0.84**	0.81

Table 4: Median time taken (s) to untangling commits for each method by project and number of tangled concerns up to 3 sig figs. The runtime cost differences are significant to $p < 0.001$ for all overall results according to a two-tailed Wilcoxon pair-wise test, except δ-PDG+CV vs Heddle, where the difference is not statistically significant ($p = 0.51$). Entries indicated by a '*' signify that there was no relevant data point to report the performance on and those indicated by 'x' indicate time-outs.

Project Name	Barnett *et al.* [2]			Herzig *et al.* [6]			δ-PDG+CV			Heddle (δ-NFG + WL)		
	2	3	Overall	2	3	Overall	2	3	Overall	2	3	Overall
Commandline	**0.10**	**8.51**	**0.12**	10.51	8.56	9.26	0.42	153.55	0.59	0.85	182.62	1.04
CommonMark	**2.56**	*	**2.56**	$1.96{\times}10^3$	*	$1.96{\times}10^3$	10.38	*	10.38	14.95	*	14.95
Hangfire	**1.15**	**4.97**	**1.95**	$1.30{\times}10^4$	$5.57{\times}10^4$	$1.72{\times}10^4$	10.61	123.99	13.84	8.06	45.29	11.64
Humanizer	**0.44**	**0.23**	**0.41**	40.62	49.53	44.44	9.24	x	9.24	4.86	2.56	4.58
Lean	**1.00**	**1.59**	**1.28**	345.05	173.58	288.35	19.28	17.08	19.28	18.07	24.07	18.23
Nancy	**2.06**	**5.63**	**2.42**	570.57	$1.29{\times}10^3$	600.16	22.04	20.52	21.96	18.38	85.55	21.78
Newtonsoft.Json	**2.14**	**6.42**	**2.35**	225.62	510.77	230.49	20.04	51.54	20.32	8.01	11.98	8.58
Ninject	**1.25**	*	**1.25**	81.53	*	81.53	4.52	*	4.52	14.99	*	14.99
RestSharp	**0.74**	1.25	**0.78**	46.22	222.98	72.09	7.80	**1.01**	4.99	9.86	26.25	10.11
Overall	**1.02**	**5.02**	**1.41**	81.53	647.99	117.35	7.17	70.35	10.27	7.99	43.29	9.56

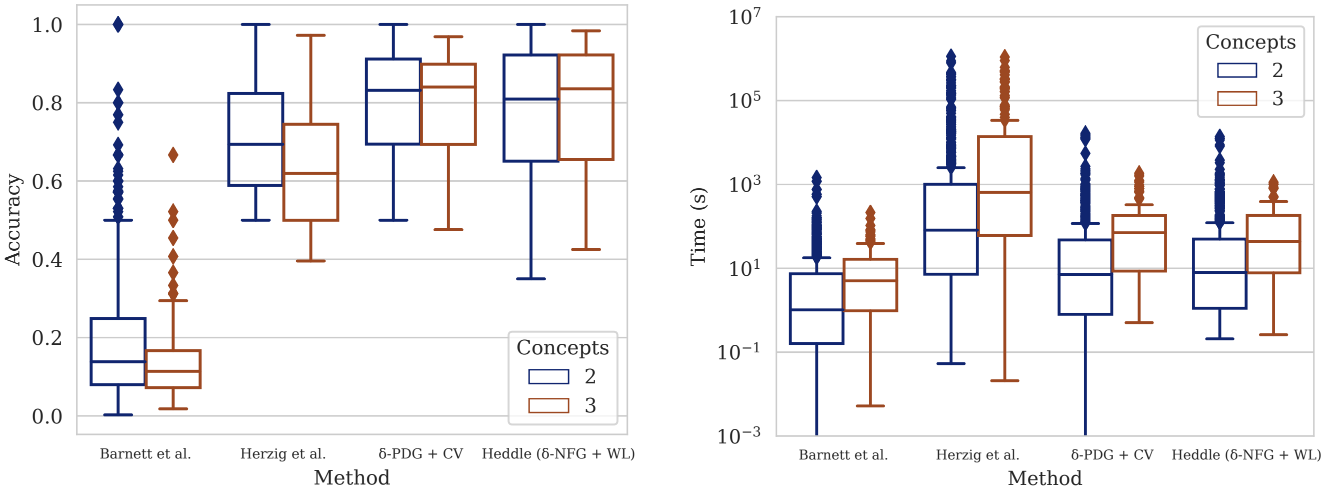
they compute a probability that two files are changed together, and, by proxy, tackle the same concern, from the version history. This allows them to better cluster related changes that span multiple files. Heddle, in such cases, relies only on the existence of call edges between the different files when projected onto a δ-NFG. Further, Herzig *et al.* has oracle access to the number of concerns while heddle has not. Despite the lack of explicit file-level relationships or oracle access, Heddle's accuracy matches confidence voters when applied to δ-NFGs, and outperforms them when they are applied to diff-regions.

Of the four methods we consider, our re-implementation of Barnett *et al.*'s method (Section 5.3) produces the lowest median accuracy — 0.13. We believe that two reasons account for this accuracy. First, we evaluate our re-implementation on trivial parts. Second, Barnett *et al.* speculate that their high FN rate is due to relations, like method calls, that def-use chains miss [2, §VI.A]. We emphasise that Barnett *et al.* performed much better in its native setting. They built their approach for Microsoft developers and the commits they handle on a daily basis. Section 7 details the threats to Heddle's validity this difference in methodology incurs. Section 8.2 further details their approach and evaluation and its differences to Heddle.

When considering accuracy for an increasing number of concerns (See Figure 4), only Barnett *et al.* and Herzig *et al.* report statistically significant performance drops ($p < 0.01$ and $p < 0.001$ according to a Mann-Whitney U test). Barnett *et al.*'s drop is, however, not observable at two decimal points, while Herzig *et al.*'s drop is by 0.07. Both δ-NFGs-based tools report statistically indistinguishable results as the number of concerns increases.

As heddle is not privy to the number of concerns, its behaviour on atomic commits is interesting. When we apply heddle to atomic commits, they are correctly identified as atomic with 0.63 accuracy. This results is when we ask the the yes/no question 'Is this commit atomic?'. We also want to determine how wrong heddle is when creating spurious partitions. For this, we consider the node-level accuracy of heddle. The result is 0.93, suggesting that heddle often mislabels a small number of changed nodes.

Table 3 shows detailed per project results broken down by project and number of concerns for each of the four untangling techniques.



(a) Accuracy

(b) Runtime

Figure 4: Boxplot comparing the accuracy of the baseline and Heddle (Figure 4a) as well as time taken (s) to segment a commit (Figure 4b) for all projects. The drop in accuracy for Herzig *et al.*'s approach as the number of concerns increases is significant to $p < 0.001$ and Barnett *et al.*'s to $p < 0.01$ according to a Mann-Whitney U test. The results of the same test for δ-PDG+CV and Heddle indicate that there is no statistically significant difference ($p = 0.28$ and $p = 0.76$ respectively). All increases in time taken to segment are statistically significant ($p < 0.001$, Mann-Whitney U test).

6.2 Untangling Running Time

Figure 4b shows that Barnett *et al.* [2]'s def-use chain technique is by far the fastest. This result is expected because the algorithm is, at its core, strongly connected components detection over a sparse graph, and is therefore linear in the number of nodes in def-use chains that contain at least one addition. However, as we have previously seen in Table 3, its accuracy is considerably worse.

Heddle is 32 times faster than Herzig *et al.* in a pair-wise ratio test. Where n is the number of diff-regions, the Herzig *et al.* technique requires n^2 shortest path computations, each requiring the solution of n^2 reachability queries over the dataflow graph. Consider the sparse occurrence matrix that encodes which commit touched which file; its dimensions are the number of commits by number of files that ever existed in the repository. Herzig *et al.*'s technique also sums each row of this matrix. Although their technique needs these steps only to populate the distance matrix before agglomerative clustering, these operations are expensive and must be computed for all diff-regions within a patch. The fact that their technique is heavy weight is unsurprising.

When compared to δ-PDG+CV, the performance on graphs tangling only two concerns is comparable; however, Heddle scales better as the number of concerns, and the number of changed nodes increases. We estimate the runtime of both Heddle and δ-PDG+CV using a robust linear model regression and fitting a second order polynomial in the number of changed nodes (n). We find Heddle to scale with $t = 0.3371 - 0.0041n + 0.0015n^2$, $R^2 = 1.00$ and δ-PDG+CV with $t = 0.8794 - 0.0528n + 0.0019n^2$, $R^2 = 0.99$. At 500 nodes changed, which is common in our dataset, this would account for a difference of 68 seconds.

Finally, we compute the pair-wise ratio of runtimes and find that Heddle is, over the median of these ratios, 9 times slower than Barnett *et al.* and 32 times faster than Herzig *et al.* at untangling commits, taking, on average, ten seconds per commit.

7 THREATS TO VALIDITY

Heddle faces the usual threat to its external validity: the degree to which its corpus of commits across a set of projects is representative. The fact that we construct tangled commits exacerbates this threat and introduces the construct validity threat that commits that we assume are atomic, are not, in fact, atomic. To address the latter threat, we validated the atomicity of the commits, from which we built tangled commits, on a small, uniform sample of 30 commits across our corpus. As is conventional, we choose 30 because this is typically when the central limit theorem starts to apply [8]. We did not validate the representativeness of our corpus against a real-world sample of tangled commits. Ground truth in real-world samples can be hard to identify, so we opted to use the methodology from Herzig *et al.* [6] to create an artificial corpus that mimics some tangled commits we expect developers to make; it captures the intuition of a developer committing multiple consecutive work units as a single patch. This decision restricts our results only to the type of tangled commits we mimic, which generalise only in so far as our algorithmically tangled commits generalise. Further, like Barnett *et al.*, we evaluated Heddle only on C# files, so, despite Flexeme's language-agnosticism, Heddle's result may not generalise to other languages.

Our re-implementations of Herzig *et al.*'s and of Barnett *et al.* may contain errors. Section 5.3 details these re-implementations and where they differ from their authors' descriptions of the original implementations. Finally, we published these re-implementations at https://pppi.github.io/Flexeme/, so other researchers can vet our work.

We borrowed Herzig *et al.*'s commit untangling evaluation strategy wholesale, as Section 5.1 and Section 5.2 detail. Thus, we were able to directly compare our work with theirs. Barnett *et al.* opted for a different evaluation strategy (Section 8.2), because obtaining a ground truth for their evaluation is too time-consuming in their setting. Thus, we can neither directly compare Heddle against their approach, nor assess our re-implementation relative to their tool. They also conducted a user study showing both a developer need for such tooling and that their suggestions are useful. Because we did not conduct a user study, our results lack the sanction of developer approval.

8 RELATED WORK

We first discuss the impact of tangled commits both on developers and researchers. We then discuss approaches to untangling such commits followed by a discussion of multiversion representations. We conclude with a discussion of graph kernels.

8.1 Impact of Tangled Commits

Tao *et al.* [27] were amongst the first to highlight the problem of change decomposition in their study on code comprehension; they highlight the need for decomposition when many files are touched, multiple features implemented, or multiple bug fixes committed. The latter is diagnosed by Murphy-Hill *et al.* [21] as a deliberate practice to improve programmer productivity. Tao *et al.* conclude that decomposition is required to aid developer understanding of code changes.

Independently, Herzig *et al.* [6, 7] investigate the impact of tangled commits on classification and regression tasks within software engineering research. The authors manually classify a corpora of real-world changesets as atomic, tangled or unknown, and find that the fraction of tangled commits in a series of version histories ranges from 7% to 20%; they also find that most projects contain a maximum of four tangled concerns per commit, which is consistent with previous findings by Kawrykow and Robillard [11]. They find non-atomic commits significantly impact the accuracy of classification and regression tasks such as fault localisation.

8.2 Untangling Commits into Atomic Patches

It is natural to think of identification of communities in the δ-NFG as a slicing problem [28]. However, boundaries across concerns do not naturally map to a slicing criterion; it is unclear how to seed a slicing algorithm and when to terminate it. This is because concerns are linked with multiple edges which makes their separation difficult to specify with a slicing criterion. In the rest of this section, we discuss the literature around the problem of tangled commits and the theoretical foundations of flexeme.

Research on the impact of both tangled commits and non-essential code changes prompted an investigation into changeset decomposition. Herzig *et al.* [6, 7] apply confidence voters in concert with agglomerative clustering to decompose changesets with promising results, achieving an accuracy of 0.58-0.80 on an artificially constructed dataset that mimics common causes of tangled commits. In contrast, Kirinuki *et al.* [13, 14] compile a database of atomic patterns to aid the identification of tangled commits; they manually classify the resulting decompositions as True, False, or Unclear, and find more than half of the commits are correctly identified as tangled. The authors recognise that employing a database introduces bias into the system and may necessitate moderation via heuristics, such as ignoring changes that are too fine-grained or add dependencies.

Other approaches rely on dependency graphs and use-define chains: Roover *et al.* [23] use a slicing approach to segment commits across a Program Dependency Graph, and correctly classify commits as (un)tangled in over 90% of the cases for the systems studied, excluding some projects where they are hampered by toolchain limitations. They propose, but do not implement, the use of System Dependency Graphs to reduce some of the limitations of their approach, such as being solely intraprocedural. Flexeme tackles interprocedural and cross-file dependencies by merging the δ-PDGs of the files touched by a commit.

Barnett *et al.* [2] implement and evaluate a commit-untangling prototype. This prototype projects commits onto def-use chains, clusters the results, then classifies the clusters as trivial or non-trivial. A cluster is trivial if its def-use chains all fall into the same method. Barnett *et al.* employ a mixed approach to evaluate their prototype. They manually investigated results with few non-trivial clusters (0-1), finding that their approach correctly separated 4 of 6 non-atomic commits, or many non-trivial clusters (> 5), finding that, in all cases, their prototype's sole reliance on def-use chains lead to excessive clustering. For results containing 2–5 clusters, they conducted a user-study. They found that 16 out of the 20 developers surveyed agreed that the presented clusters were correct and complete. This result is strong evidence that their lightweight and elegant approach is useful, especially to the tangled commits that Microsoft developers encounter day-to-day. During the interviews, multiple developers agreed that the changeset analysed did indeed tangle two different tasks, sometimes even confirming that developers had themselves separated the commit in question after review. In addition to validating their prototype, their interviews also found evidence for the need for commit decomposition tools. Because they use def-use chains and ignore trivial clusters, Barnett *et al.*'s approach can miss tangled concerns that Flexeme can discern. Barnett *et al.*'s user study itself shows that this can matter: it reports that some developers disagreed with the classification of some changesets as trivial.

Dias *et al.* [4] take a more developer-centric approach and propose the EpiceaUntangler tool. They instrument the Eclipse IDE and use confidence voters over fine-grained IDE events that are later converted into a similarity score via a Random Forest Regressor. This score is used similarly to Herzig *et al.* [6]'s metrics, *i.e.* to perform agglomerative clustering. They take an instrumentation-based approach to harvest information that would otherwise be lost, such as changes that override earlier ones. This approach also avoids relying on static analysis. They report a high median success rate of 91% when used by developers during a two-week study. While Dias *et al.* sidestep static analysis, they require developers to use an instrumented IDE. Heddle is complementary to EpiceaUntangler: it allows reviewers to propose untanglings of code that may originate from development contexts where instrumentation might not be possible.

8.3 Multiversion Representations of Code

Related work has considered multiversion representations of programs for static analysis. Kim and Notkin [12] investigate the applicability of different techniques for matching elements between different versions of a program. They examine different program representations, such as String, AST, CFG, Binary or a combination of these as well as the tools that work on them on two hypothetical scenarios. They only consider the ability of the tools to match elements across versions and leave the compact representation of a multiversion structure as future work. Some of the conclusions from the matching challenges presented by Kim and Notkin [12] are echoed in Flexeme as well, we make use of the UNIX diff as it is stored within version histories; however, we also make use of line-span hints from the compilers for each version of the application to better facilitate matching nodes within a NFG.

Le *et al.* [16] propose a Multiversion Interprocedural Control Graph (MVICFG) for efficient and scalable multiversion patch verification over systems such as the PuTTY SSH client. Our δ-PDG is a generalisation of this approach to a more expressive data structure, with applications beyond traditional static analysis.

Alexandru *et al.* [1] generalise the Le *et al.* MVICFG construction to arbitrary software artefacts by constructing a framework that creates a multiversion representation of concrete syntax trees for a git project. They adopt a generic ANTLR parser, allowing them to be language agnostic, and achieve scalability by state sharing and storing the multi-revision graph structure in a sparse data structure. They show the usefulness of such a framework by means of 'McCabe's Complexity', which they implement in this framework such that it is language agnostic, does not repeat computations unnecessarily and reuses the data stores in the sparse graph by propagating from child to parent node. Sebastian and Harald [24] propose a compact, multiversion AST that cleverly shares state across versions. Flexeme, in contrast, rests on PDGs and is well-suited for the untangling tasks, as our evaluation demonstrates.

8.4 Semantic Slicing of Version Histories

Features in a system often co-evolve, which tangles the changes made for a one high-level feature with others in a version history. To resurface feature-specific changes, they dynamically slice a target version, then walk backwards in history while they can reverse the intra-version patch without conflict; at each version they reach, they add any commit that contains a hunk that touches the current slice to it. The goal of this semantic slicing of version histories is to find a minimal slice of a version history that captures the evolution of a feature. Li *et al.* [17, 19] first formulated and introduced this problem. Semantic slicing is a form of commit untangling backwards through history. This retrospective framing is why they treat the history as immutable. In this initial solution, Li *et al.* treat commits as atomic so their slices may contain noise introduced by tangled commits. To reduce this noise, Li *et al.*, in more recent work [18], unpack commits into single-file commits into a private, local history. Flexeme, in contrast, is static and online: built from the ground up to rewrite commits as developer make history. As such Flexeme and semantic slicing are complementary: Flexeme would improve the signal to noise ratio of semantic slicing. An interesting direction for future work would be to use Flexeme to preprocess version histories prior to semantically slicing them as with Definer [18].

8.5 Graph Kernels

Real world data is often structured, from social networks, to protein interactions and even source code. Knowing if a graph instance is similar to another is useful if we wish to make predictions on such data by means that employ either similarity or distance. Vishwanathan *et al.* [29] provide a unified framework to study graph kernels, bringing previously defined kernels under a common umbrella and offering a new method to compute graph kernels on unlabelled graphs in a fast manner, reducing the asymptotic cost from $O(n^6)$ to $O(n^3)$. They mainly study the construction of the different graphs and demonstrate the run-time improvement without applying it to a downstream prediction task. Shervashidze *et al.* [25] introduce the Weisfeiler-Lehman Graph kernel, which they evaluate with three underlying kernels — subtree, edge histogram, and shortest path — on several chemical and protein graph datasets. Although code is often represented as a graph structure and the methods presented here are also used by us to compute graph similarity, this literature primarily concerns itself with chemical and social network datasets that have become standardised benchmarks.

9 CONCLUSION

We have presented Flexeme— a new approach to commit untangling. Flexeme's realisation in heddle advances the state-of-the-art: it is 0.14 more accurate (achieving 0.81) and 32 times faster than the previous state-of-the-art. This result rests on a novel data structure, δ-NFG, which augments a multiversion program dependence graph (also introduced in this paper) with name flows. δ-NFG facilitate dual-channel reasoning across versions. Thus, we believe that δ-NFG will be useful on tasks other than commit untangling, such as code refactoring, notably renaming, or code summarisation, as when suggesting docstrings.

ACKNOWLEDGEMENTS

This research was supported by the EPSRC Ref. EP/J017515/1 and EPSRC Ref. EP/P005314/1.

REFERENCES

[1] Carol V Alexandru, Sebastiano Panichella, Sebastian Proksch, and Harald C Gall. 2019. Redundancy-free analysis of multi-revision software artifacts. *Empirical Software Engineering* 24, 1 (2019), 332–380.

[2] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K. Lahiri. 2015. Helping developers help themselves: Automatic decomposition of code review changesets. *Proc. - Int. Conf. Softw. Eng.* 1, August 2014 (2015), 134–144. https://doi.org/10.1109/ICSE.2015.35

[3] Santanu Kumar Dash, Miltiadis Allamanis, and Earl T. Barr. 2018. RefiNym: Using Names to Refine Types. In *Foundations of Software Engineering (FSE)*.

[4] Martin Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stephane Ducasse. 2015. Untangling fine-grained code changes. In *2015 IEEE 22nd Int. Conf. Softw. Anal. Evol. Reengineering, SANER 2015 - Proc.* IEEE, 341–350. https://doi.org/10.1109/SANER.2015.7081844 arXiv:1502.06757

[5] Jeanne Ferrante, Karl J. Ottenstein, and Joe D Warren. 1984. The program dependence graph and its use in optimization. In *Lect. Notes Comput. Sci. (including*

Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics), Vol. 167 LNCS. 125–132. https://doi.org/10.1007/3-540-12925-1_33

[6] Kim Herzig, Sascha Just, and Andreas Zeller. 2016. The impact of tangled code changes on defect prediction models. *Empir. Softw. Eng.* 21, 2 (2016), 303–336. https://doi.org/10.1007/s10664-015-9376-6

[7] Kim Herzig and Andreas Zeller. 2013. The impact of tangled code changes. *IEEE Int. Work. Conf. Min. Softw. Repos.* (2013), 121–130. https://doi.org/10.1109/MSR.2013.6624018

[8] Robert V. Hogg, Elliot A. Tanis, and Dale L. Zimmerman. 2020. *Probability and statistical inference*. Pearson.

[9] Robert W. Irving and David F. Manlove. 2002. The Stable Roommates Problem with Ties. *J. Algorithms* 43, 1 (April 2002), 85–105. https://doi.org/10.1006/jagm.2002.1219

[10] Eric Jones, Travis Oliphant, Pearu Peterson, et al. 2001–. SciPy: Open source scientific tools for Python. http://www.scipy.org/ [Online; accessed 17.08.2018].

[11] David Kawrykow and Martin P. Robillard. 2011. Non-essential changes in version histories. *Proceeding 33rd Int. Conf. Softw. Eng. - ICSE '11* (2011), 351. https://doi.org/10.1145/1985793.1985842

[12] Miryung Kim and David Notkin. 2006. Program element matching for multi-version program analyses. In *Proceedings of the 2006 international workshop on Mining software repositories*. 58–64.

[13] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. 2014. Hey! are you committing tangled changes?. In *Proc. 22nd Int. Conf. Progr. Compr. (ICPC 2014)*. ACM Press, New York, New York, USA, 262–265. https://doi.org/10.1145/2597008.2597798

[14] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. 2017. Splitting commits via past code changes. *Proc. - Asia-Pacific Softw. Eng. Conf. APSEC* (2017), 129–136. https://doi.org/10.1109/APSEC.2016.028

[15] H. W. Kuhn. [n.d.]. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly* 2, 1–2 ([n. d.]), 83–97. https://doi.org/10.1002/nav.3800020109

[16] Wei Le and Shannon D. Pattison. 2014. Patch verification via multiversion interprocedural control flow graphs. *Proc. 36th Int. Conf. Softw. Eng. - ICSE 2014* (2014), 1047–1058. https://doi.org/10.1145/2568225.2568304

[17] Yi Li, Julia Rubin, and Marsha Chechik. 2015. Semantic slicing of software version histories. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*. 686–696.

[18] Yi Li, Chenguang Zhu, Milos Gligoric, Julia Rubin, and Marsha Chechik. 2019. Precise semantic history slicing through dynamic delta refinement. *Autom. Softw. Eng.* 26, 4 (2019), 757–793. https://doi.org/10.1007/s10515-019-00260-8

[19] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. 2018. Semantic Slicing of Software Version Histories. *IEEE Trans. Softw. Eng.* 44, 2 (2018), 182–201. https://doi.org/10.1109/TSE.2017.2664824

[20] Microsoft. [n.d.]. Microsoft Rosyln. https://github.com/dotnet/roslyn. Accessed: 31-05-2018.

[21] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2012. How we refactor, and how we know it. *IEEE Trans. Softw. Eng.* 38, 1 (2012), 5–18. https://doi.org/10.1109/TSE.2011.41 arXiv:arXiv:1011.1669v3

[22] Jakob Nielsen. 1993. Response times: the three important limits. *Usability Engineering* (1993).

[23] De Roover and Ward Muylaert. 2017. Untangling Source Code Changes Using Program Slicing. (2017).

[24] Panichella Sebastian and Proksch Harald. 2018. Redundancy-free Analysis of Multi-revision Software Artifacts Redundancy-free Analysis of Multi-revision Software Artifacts. May (2018).

[25] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. 2011. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research* 12, Sep (2011), 2539–2561.

[26] Giannis Siglidis, Giannis Nikolentzos, Stratis Limnios, Christos Giatsidis, Konstantinos Skianis, and Michalis Vazirgiannis. 2018. GraKeL: A Graph Kernel Library in Python. *arXiv preprint arXiv:1806.02193* (2018).

[27] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. 2012. How do software engineers understand code changes? *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng. - FSE '12* (2012), 1. https://doi.org/10.1145/2393596.2393656

[28] Frank Tip. 1994. *A Survey of Program Slicing Techniques*. Technical Report. Amsterdam, The Netherlands, The Netherlands.

[29] S Vichy N Vishwanathan, Nicol N Schraudolph, Risi Kondor, and Karsten M Borgwardt. 2010. Graph kernels. *Journal of Machine Learning Research* 11, Apr (2010), 1201–1242.

[30] B. Yu. Weisfeiler and A. A. Leman. 1968. Reduction of a graph to a canonical form and an algebra arising during this reduction.

[31] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. 2005. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering* 31, 6 (2005), 429–445.