Improving pattern tracking with a language-aware tree differencing algorithm

Nicolas Palix, Jean-Rémy Falleri, Julia Lawall

Improving Pattern Tracking with a Language-Aware Tree Differencing Algorithm

Nicolas Palix
Grenoble - Alps University/UJF, LIG-Erods
France
nicolas.palix@imag.fr

Jean-Rémy Falleri
Univ. Bordeaux, LaBRI, UMR 5800
France
falleri@labri.fr

Julia Lawall
Inria/LIP6/UPMC/Sorbonne University
France
Julia.Lawall@lip6.fr

***Abstract*—Tracking code fragments of interest is important in monitoring a software project over multiple versions. Various approaches, including our previous work on Herodotos, exploit the notion of Longest Common Subsequence, as computed by readily available tools such as GNU Diff, to map corresponding code fragments. Nevertheless, the efficient code differencing algorithms are typically line-based or word-based, and thus do not report changes at the level of language constructs. Furthermore, they identify only additions and removals, but not the moving of a block of code from one part of a file to another. Code fragments of interest that fall within the added and removed regions of code have to be manually correlated across versions, which is tedious and error-prone. When studying a very large code base over a long time, the number of manual correlations can become an obstacle to the success of a study.**

In this paper, we investigate the effect of replacing the current line-based algorithm used by Herodotos by tree-matching, as provided by the algorithm of the differencing tool GumTree. In contrast to the line-based approach, the tree-based approach does not generate any manual correlations, but it incurs a high execution time. To address the problem, we propose a hybrid strategy that gives the best of both approaches.

***Keywords*—code tracking, tree-matching, code metrics**

I. Introduction

Recent years have seen the development of numerous tools that scan a code base for fragments of code with particular properties. Such tools include, for example, fault finding tools [1], [2], [3], [4] that identify fragments of code that represent a fault, and clone detection tools [5], [6], [7], [8] that find regions of code that occur at multiple positions across a code base. Fully realizing the benefit of these tools, however, requires not simply applying them to a single version of the software, but applying them to the software repetitively as the software evolves. Doing so will reveal new faults, new clones, etc. But the tool will also generate reports based on code that is preserved from one version to the next. In order to both not bother the user with reports that he has already seen and to be able to track properties of the software over time, it is necessary to distinguish between reports relating to code fragments that have been removed, that have been preserved, and that have been added over time. So that the process will scale to large code bases, over many versions, this *correlation* of reports generated for one version with reports generated for the next must be, as much as possible, automatic.

In this paper, we focus on faults, and assume that a fault to track is represented by a pair of a *keyword*, *i.e.*, some free text obtained from the source code, and a single *position* within the code, *i.e.*, the file name, starting and ending line numbers, and starting and ending column offsets of a term that is critical to the presence of a fault. Based on this assumption, one strategy to determine whether a code fragment is preserved from one version to the next is to check whether the identified position is in the code that has not changed, *i.e.*, the Longest Common Subsequence (LCS), between the two versions. If it is, we conclude that the represented code fragment is preserved from one version to the next. On the other hand, if the code position is not in the LCS, it may be the case that the fault has been removed, or that it remains, perhaps in a slightly different form. For example, one may consider that a null pointer dereference is not removed by the systematic renaming of the variables involved in the dereference expression, even if the variables are deeply intertwined with the dereference expression itself. To make this distinction, a human must intervene.

An off-the-shelf, efficient tool for computing the LCS between two files is GNU Diff [9]. Based on the results of an LCS calculation, GNU Diff produces a line-based *edit script* containing the lines in which at least one character is removed and the lines in which at least one character is added.[1] In previous work [10], [11], the first and third authors have studied the faults in the versions of the Linux kernel of the 2.6 series, from 2003 to 2011. Their work uses the open-source tool *Coccinelle*[2] [1] to find occurrences of faults in each version of the Linux 2.6 C source code. Obtaining an accurate view of the lifetime of these faults requires correlating fault occurrences from one version to the next. For this, they used another open-source tool, *Herodotos*[3] [12], to reconstruct the history of each fault through the different versions. Herodotos uses GNU Diff to identify lines in the LCS. Specifically, given the position of a fault in one version, Herodotos *predicts* the position of the fault in the next version, based on the information obtained using GNU Diff about the lines that were added and removed earlier in the file. If the predicted position is in an unchanged region of the code, and if Coccinelle finds the same kind of fault at that position, Herodotos considers that the fault in the former version is the same as the fault in the latter one. On the other hand, if the predicted position is found in the changed code, Herodotos considers that it is not known whether the fault is preserved across versions, and the user has to perform the correlation manually.

[1] We describe the so-called "unified diff format" of GNU Diff.
[2] https://github.com/coccinelle/coccinelle
[3] https://github.com/coccinelle/herodotos

The algorithm used by GNU Diff is very efficient and reasonably effective. Nevertheless, the results are very coarse-grained. If a simple modification such as adding a comment or renaming a variable occurs on a line that contains a fault, the line is considered to be removed and added. In this case, Herodotos's prediction mechanism fails and no correlation is automatically performed. The same occurs when a block of code is moved, for instance to create a new function. When prediction fails, Herodotos proposes to the user a set of possible correlations between all of the faults that disappear from a given file in one version, and all of the faults that appear in the same file in the next version. This two-phase process has allowed large studies to be carried out [10], [11], [12], [13]. However, it does not scale well and the manual phase is time-consuming and error-prone.

The core of the problem is the line-based nature of GNU Diff and the limited set of kinds of changes identified, considering only adds and removes, but not moves and updates. There exists another class of approaches to compute modifications, based on abstract syntax trees (ASTs) [14]. An AST-based approach produces results at the level of individual nodes in the AST, which are generally more fine-grained than results based on lines. One such tool is the open-source *GumTree* tool[4] [15]. GumTree specifically takes into account additions, deletions, updates and moves of individual tree nodes, and has the goal of producing results that are easier for users to understand than those of GNU Diff.

In this paper, we present how we have integrated the tree-based comparison performed by GumTree with the correlation algorithm of Herodotos. In doing so, we have improved the rate of success of the automatic correlation of positions across multiple versions of C source code. We evaluate the benefits of this approach for tracking software faults in a large software project, Linux 2.6, with over 8 MLOC. While in our previous study we had to manually correlate over 800 pairs of faults, this new approach fully automates the process in practice. This fully automated correlation, however, comes at the price of a higher computing time: from 30 to 123 times more CPU time depending on the complexity of the changes in faulty files. To address this issue, we propose a hybrid approach, combining GNU Diff and GumTree, that, while still fully automating the correlation process, has an overhead of about 3 times in terms of CPU time.

The contributions of our work are as follows:

- We extend GumTree to treat C code.
- We propose an approach to position prediction in C code based on a tree-based comparison.
- We develop an associated toolchain that removes all manual correlations in practice, but at the cost of a high computing time.
- We design a hybrid strategy that reduces the computing time, while still achieving a fully automated correlation process. We observe speedups ranging from 6 to 11 times compared to a pure tree-matching based strategy.

The rest of this paper is organized as follows. Section II presents the tools used in our approach, including the current state of GumTree and Herodotos. Section III presents the improvements of the approach, and the integration of the tools that we have used or developed. Section IV evaluates the benefits of our new approach in terms of the number of manual correlations to perform and the time taken for the automatic ones. Finally, Section V presents related work and Section VI concludes.

[4]https://github.com/GumTreeDiff/gumtree

II. Background

In this section, we present the tools used by our approach: GumTree, to produce a tree-based description of code modifications, and Herodotos, to correlate fault reports and thus reconstruct the fault history.

A. Overview of GumTree

GumTree, developed in previous work [15], is a differencing algorithm that works on abstract syntax trees (ASTs) extracted from source code. While classical text diff algorithms, such as GNU Diff, consider source code as a sequence of text lines and present differences as sequences of added or removed lines, GumTree considers source code as an AST and presents differences as a sequence of added, removed, updated or moved nodes. Since differences are computed at the tree level, they are naturally aligned on the code structure and easier to subsequently process automatically. Additionally, since differences are expressed with a richer set of edit operations on nodes, including update and move as well as add and remove, they are usually easier to understand and closer to the initial developer's intent than the text-based ones.

A detailed description of the algorithm, including an example, is available in the original article on GumTree [15]. Here, we give only an overview. The GumTree differencing algorithm works in two steps: first it establishes mappings between the nodes of two abstract syntax trees (AST) and then it deduces a *edit script*. The edit script contains the edit operations that, when applied to the first tree, yield the second tree. The GumTree algorithm focuses on the first step: computing the mappings between two ASTs. The output of this algorithm can then be used by the optimal and quadratic algorithm of Chawathe *et al.* [16] to compute the edit script.

GumTree's mapping algorithm is inspired by the way developers manually look at changes between two files. First, they search for the biggest unmodified chunks of code. These chunks are usually located in code containers (functions, classes, modules, etc.). Then, developers deduce from the unmodified chunks how the code containers can be mapped to each other. Finally, when two code containers are mapped together, developers look at the modified code they contain to see if there are additional correspondances, such as renamings. Following this model, GumTree's algorithm to compute the mappings between two ASTs is composed of two successive phases:

1) A greedy top-down algorithm to find isomorphic subtrees of decreasing height. Mappings are established between the nodes of these isomorphic subtrees. These mappings are called *anchor mappings*.
2) A bottom-up algorithm in which two nodes are considered to match (called a *container mapping*) if their descendants (their children, their children's children, *etc.*) include a large percentage of common anchor mappings. When two nodes match, the GumTree algorithm finally applies a

```
1 #include <stddef.h>
2
3 struct point {
4         int x;
5         int y;
6 };
7 struct point *a = NULL;
8
9 int fct(void) {
10         struct point *p = NULL;
11
12         return p->x;
13 }
```

Fig. 1. C program in its initial version

very expensive and accurate tree differencing algorithm to search for additional mappings (called *recovery mappings*) among their descendants. This algorithm is applied only if the number of descendants of the nodes is below a certain threshold.

The GumTree algorithm has been validated on many real differencing scenarios. Its evaluation shows that the GumTree algorithm produces more comprehensible results for human developers than GNU Diff, and produces shorter edit scripts than other tree-based diff tools.

B. Overview of Herodotos and our Code-Tracking Application

Herodotos is a tool for tracking positions of interest in a source code file across multiple versions of the software, even when code nearby in the same file has been modified. We use Herodotos to track positions that denote software faults. By tracking faults, Herodotos reconstructs fault histories, independently of the code modifications occurring before or after the fault position.

In the context of our Linux 2.6 experiments [10], [11], we first used the program matching tool Coccinelle [1] to automatically find potential faults in the Linux kernels. Coccinelle fault patterns were applied to every studied version of a project, producing a list of fault reports for each version. Each report contains some text describing the fault and the position of the fault, comprising the file name, the starting and ending line numbers, and the starting and ending columns of a term that is key to the presence of the fault. For example, for the code in Figure 1, the Coccinelle null pointer dereference fault-finding rule would generate a report for line 12, columns 8-9, corresponding to the reference to `p`,[5] as `p` is dereferenced at this point even though its value is null. If the patch shown in Figure 2 were then applied to this code, resulting in the code shown in Figure 3, the report would indicate a dereference of the pointer on line 13 from column 9 to 10. We then used Herodotos to correlate the fault reports between versions [12].

To illustrate the reasoning captured by Herodotos, we again consider the source code shown in Figure 1, the patch on this code, shown in Figure 2, and the result of applying this patch, shown in Figure 3. A GNU Diff patch is composed of *hunks*, which describe a local change. Each hunk starts with a `@@` line giving the edited positions in the source and destination files,

[5]Following Linux conventions, indentation is implemented using tabs, not spaces, when possible.

```
1 --- ver1/code.c 2015-01-24 14:39:54 +0100
2 +++ ver2/code.c 2015-01-24 14:40:04 +0100
3 @@ -1 +1 @@
4 -#include <stddef.h>
5 +#include <stdlib.h>
6 @@ -4,2 +4 @@
7 -        int x;
8 -        int y;
9 +        int x, y;
10 @@ -12 +11,5 @@
11 -        return p->x;
12 +        a = malloc(sizeof(struct point));
13 +        if(a) {
14 +                return p->x;
15 +        } else
16 +                return -1;
```

Fig. 2. Patch to the C program

```
1 #include <stdlib.h>
2
3 struct point {
4         int x, y;
5 };
6 struct point *a = NULL;
7
8 int fct(void) {
9         struct point *p = NULL;
10
11         a = malloc(sizeof(struct point));
12         if(a) {
13                 return p->x;
14         } else
15                 return -1;
16 }
```

Fig. 3. C program in its modified version

with a summary of the edited lines. These lines are interpreted by Herodotos. The remaining lines of a hunk describe the modification; Herodotos does not use these lines.

By interpreting the line 3, Herodotos determines that the first change specification removes and adds the same number of lines of code, and thus does not affect the line numbering of subsequent code lines. Similarly, because the second hunk (lines 6 to 9) removes one more line than it adds, any code in the original source code between the second and third hunks is pulled back by one line. Finally, the pointer dereference reported by the Coccinelle rule is part of the third hunk. Herodotos is thus unable to predict the position of the dereference in the next version. In that case, Herodotos will look for the faults of the same kind in the next version. If any are found, Herodotos will propose the set of pairs of the uncorrelated fault in the original source file with each uncorrelated fault in the patched source file as possible correlations. In the subsequent manual correlation phase, the user will have to review each proposed pair of faults, and annotate at most one of them as being correct. If there is no fault of the same kind in the next version of the file, the current fault is assumed to be fixed.

The above approach is quadratic in the number of faults of a given kind in a file, and thus when there is a high number of such faults, this approach produces a high number of correlation propositions. We thus decided to limit the set of proposed correlations to the set of reports for which the positions describe

two pieces of code of the same length, in the common case where the starting and ending line numbers are the same. Here, the length is being used as an approximation of the content, because the content is not always available in our fault reports. This extension handles easily the cases where a faulty position is moved elsewhere in the file, without renaming.

III. Our Approach

To improve the precision of Herodotos, we propose to replace its use of the line-based GNU Diff with the tree-based code differences generated by GumTree. While GNU Diff works on a pair of raw source code files, GumTree expects a pair of ASTs. Thus, our first challenge in integrating GumTree is to produce a parser for C files that generates ASTs in the format that GumTree expects. We then describe some extensions to GumTree and Herodotos that were required to allow them to work together. Finally, as the resulting tool is much slower than the GNU Diff-based version of Herodotos, we propose a number of performance improvements.

A. Cgum, a Parser for C Code

The GumTree algorithm is language independent, and requires only two ASTs, which can be furnished in a number of formats, including XML. GumTree requires that each node in the AST be associated with a node type that is represented as an integer, and that each node contain the position of the associated code in the original source code file. GumTree also allows a node to optionally contain a string value; for instance, an identifier node would likely contain the name of the identifier that it represents, such as `p`. To use GumTree on a previously unsupported language, we must therefore write a parser that is capable of producing such ASTs. For Linux kernel code, we needed to create such a parser for C code.

To create such a GumTree AST for Linux kernel code, we reuse the parser of Coccinelle [17], extending this parser to a tool called *cgum*.[6] The parser of Coccinelle has been specifically designed to not require expansion of preprocessor macros, to handle the peculiarities of Linux kernel code, and to be robust in the face of parse errors. In particular, if a parse error is encountered, the Coccinelle parser skips to the next top-level code unit and continues, allowing it to parse a high percentage of the code. In practice, the Coccinelle parser parses completely 89.56% of the Linux 3.0 files, and overall parses around 99.08% of the Linux 3.0 code tokens. For the set of files relevant to the **Lock**, **NullRef** and **Null** fault types studied in Section IV, *i.e.* the set of files with at least one fault, independently of the Linux version, the Coccinelle parser parses completely 84.85% of the files, and overall parses around 99.52% of the code tokens.

Figure 4 shows an example of the *cgum* XML output. This XML document is composed of nested `tree` tags, which correspond to the nodes of the AST. The nesting of these tags represent the parent-child relationship in the AST. To be able to relate a node to the position of the corresponding code in the source code file, each node has a `pos` (position) attribute. This position is given, for conciseness, in Figure 4 as a string of four integers, representing the beginning line and column and the ending line and column of the corresponding code.

[6]https://github.com/GumTreeDiff/cgum

```
<tree type="231400" typeLabel="RecordPtAccess"
  pos="12 8 12 12">
  <tree type="230100" typeLabel="Ident"
   pos="12 8 12 9" label="p"/>
  [...]
</tree>
```

Fig. 4. Excerpt of the *cgum* output

```
<tree type="231400" typeLabel="RecordPtAccess"
  pos="12 8 12 12" next_pos="13 9 13 13">
  <tree type="230100" typeLabel="Ident"
   pos="12 8 12 9" label="p"
   next_pos="13 9 13 10"/>
  [...]
</tree>
```

Fig. 5. The annotated *cgum* output produced by GumTree

The extension to create cgum amounts to developing a recursive pretty printer for the Coccinelle C AST that generates the GumTree XML notation. This extension represents around 1,300 lines of OCaml code.[7] For position information and for the optional string values contained in the nodes, we reuse information already collected by the Coccinelle parser. For the node-type numbers, we exploit the union tag numbers available in the internal data representation of OCaml, as made available by the OCaml `Obj` interface [18]. The data set used for the experiment represents 19,456 C files that use 939MB of disk space. Once expanded by cgum into an XML representation, this represents 34GB of data. The complete XML representation, however, is neither stored on disk nor entirely in memory as each GumTree process treats only one pair of files at a time, and Herodotos launches only one GumTree process per CPU. There is typically at most around 46MB of memory used per CPU to store the XML representation of the C files at any given time.

B. Extensions to GumTree and Herodotos

The original GumTree implementation takes as input two source code files, and produces as output an edit script. However, in our approach we are interested in being able to track the position of a particular AST node, *i.e.*, to know its position after the edit, rather than the edit operation itself. To make this information readily apparent, we introduce a new kind of output for the GumTree implementation, called an *annotated source tree*, which exposes positions instead of edits. An annotated source tree amounts to the XML generated by cgum for the first source code file, with an optional additional attribute, `next_pos`, in the `tree` element. If present, this attribute gives the position of the node in the second source code file, in the same format as `pos`. If this attribute is not present, it means that GumTree considers that the node has been deleted in the next version. The annotated source tree associated with the AST shown in Figure 4 is given in Figure 5.

To integrate GumTree with Herodotos, we first use cgum to produce a GumTree AST of versions n and $n+1$. These two ASTs are then used by GumTree to find the mappings

[7]Computed with David Wheeler's SLOCCount.

between the nodes of these ASTs. Based on this information, GumTree produces an annotated source tree for version n. For each fault report, Herodotos then traverses the annotated source tree of version n to find a node whose position is the same as the position of the fault. From this node, Herodotos tries to obtain the position of the fault in version $n+1$. Two cases can arise, depending on whether the node contains a `next_pos` attribute. If the attribute is present, GumTree has considered that the code is either unchanged or moved. In both cases the attribute indicates to Herodotos the location of the fault in the next version. If the attribute is not present, GumTree has considered that the associated code has been removed. In this case, Herodotos considers that the fault has been corrected. Herodotos also considers that the fault has been corrected if there is a `next_pos` attribute, but no fault is reported at the new position.

As an example, for the fault reported in Section II-B at position `"12 8 12 9"` and the subtree shown in Figure 5, Herodotos descends through the `RecordPtAccess` node, whose position `"12 8 12 12"` contains the desired position, and stops at the `Ident` node at position `"12 8 12 9"`, shown on lines 3-5. This node has a `next_pos` attribute (line 5), which is added by GumTree when two GumTree ASTs nodes are paired between the compared versions. From this information, Herodotos determines that the dereferenced pointer is now at line 13 between columns 9 and 10. The Coccinelle null pointer dereference fault-finding rule does generate a fault report for this position, so Herodotos considers that the fault is preserved from the former version to the latter one.

The extension to enable the use of GumTree by Herodotos amounts to changing 15 files and adding 4 files. The changes, including the additions of the new files, amount to 1,569 added lines of code and 857 removed lines of OCaml code. Regarding GumTree, one file is changed with the addition of 53 lines of Java code. We refer to the resulting implementation as *HeroGum*, and for clarity we refer to the original GNU Diff based version as *HeroDiff*.

C. Performance Improvements

Using GumTree rather than GNU Diff in Herodotos improves the precision, as described in Section IV, but introduces a lot of extra computation, for parsing the C code, creating the XML representation, and performing the tree differencing. While cgum takes about an hour and a quarter on a single processor (noted CPU time hereafter) to run on all the faulty files considered in this study, performing the tree matching introduces a high computing cost. Indeed, as shown in Section IV, Table I, the CPU time with GumTree is about 56 hours, while the CPU time with GNU Diff is about an hour. To mitigate this problem, we parallelize the computation of the GNU Diff and GumTree edit scripts, and cache the results, implying that regardless of the number of faults in a pair of files, GNU Diff and GumTree are applied to that pair of files at most once. Furthermore, as parsing the resulting XML is costly, we parse the result of GumTree only once, before caching, and cache the result as a serialized OCaml object. The cache is maintained in an in-memory file system for further efficiency.

While the use of the cache greatly improves performance, there is still the cost of running GumTree to create the initial cache entry. To reduce this cost, we propose a third extension that uses both GNU Diff and GumTree. This hybrid strategy first runs the correlation based on HeroDiff and collects each correlation failure that requires a manual correlation. Herodotos then uses GumTree as above in parallel to create a cache for the set of affected files. Finally, the remaining correlations are performed with HeroGum. We refer to this hybrid strategy as *HeroDiffGum*.

With the hybrid strategy, the overhead incurred by GumTree is paid only when the GNU Diff approach fails. On the other hand, in these cases we now have both the cost of GNU Diff and the cost of GumTree. In practice, as shown in Section IV and reported in previous work [12], HeroDiff fails in less than 1% of the correlations. Still, due to the very large number of correlations, 1% of the correlations represents a large amount of manual work for all of the fault types. As the hybrid strategy uses HeroDiff to handle more than 99% of the cases and GNU Diff is much faster than GumTree, there is still an overall performance benefit.

IV. Evaluation

We first quantitatively evaluate our new approach in terms of the number of correlations that can be done automatically and in terms of performance. We then present qualitatively the benefit of the approach, and finally discuss the few differences observed between the results produced by the various approaches.

To evaluate the performance of the three algorithms, we consider a subset of the fault reports studied during our previous work, specifically the reports generated by the **Lock**, **NullRef** and **Null** patterns [10], [11]. The **Lock** pattern identifies positions where an acquired lock is not released, and positions where a lock is double-acquired. The **NullRef** pattern identifies positions where a dereference of a pointer is followed by a null test on the pointer. The **Null** pattern identifies positions where a potentially null pointer returned from a function is not checked. We choose these patterns because of the high number of manual correlations we had to do for these cases in previous studies [10], [11].

A. Performance

Table I compares three strategies: (1) the original one, HeroDiff, based on GNU Diff and used in our previous work [10], [11], (2) a new one, HeroGum, based purely on GumTree, and (3) a new hybrid strategy, HeroDiffGum, first trying GNU Diff and then falling back on GumTree when the correlation does not succeed. The timings are given in hours:minutes:seconds format in Table I and are the average and the standard deviation computed on three runs. All the runs have been performed on a 64-core 2.1GHz machine based on AMD Opteron 6272 processors. The CPU time reports the cumulative time used by the 64 cores, while the real time reports the time observed by the user.

The cache is always used but we report separately the case where it is empty before running the experiment, and the case where it is filled by a previous run of the experiment. In a real usage, the cache is empty at the very beginning and thus the filled case represents an optimal performance that is unlikely to be achieved in practice. Nevertheless, a file that contains a

TABLE I. COMPARISON OF THE THREE STRATEGIES PROVIDED BY HERODOTOS.

Pattern	Strategy	Manual correlations to review / kept	Automatic correlations after phase 1 / phase 2		Initial state for the edit script cache	CPU time (hh:mm:ss ± ss)	CPU time slowdown versus HeroDiff	Real time (hh:mm:ss ± ss)	Real time slowdown versus HeroDiff	Real time speedup versus HeroGum
Lock	HeroDiff	28 / 23	4,110		Empty	5:41 ± 3	1	0:28 ± 0.2	1	
					Filled	0:08 ± 0.01		0:03 ± 0.00	1	
4,154	HeroGum	0	4,154		Empty	11:43:08 ± 24	123.62	12:00 ± 3	25.6	1
					Filled	3:18 ± 3		0:38 ± 0.8	13.03	1
reports	Hybrid	0	4,110	4,154	Empty	18:19 ± 2	3.22	1:37 ± 1	3.46	7.39
					Filled	0:07 ± 0.3		0:05 ± 0.03	1.59	8.19
NullRef	HeroDiff	200 / 134	14,603		Empty	34:14 ± 23	1	2:05 ± 0.9	1	
					Filled	0:29 ± 0.2		0:16 ± 0.1	1	
14,881	HeroGum	0	14,881		Empty	17:47:29 ± 2:58	30.29	42:35 ± 12	20.46	1
					Filled	16:01 ± 2:27		2:40 ± 8	10.09	1
reports	Hybrid	0	14,603	14,881	Empty	1:45:13 ± 46	2.99	3:36 ± 0.1	1.73	11.85
					Filled	0:30 ± 0.2		0:22 ± 0.3	1.38	7.30
Null	HeroDiff	329 / 139	11,786		Empty	20:34 ± 11	1	1:19 ± 0.05	1	
					Filled	0:22 ± 0.2		0:10 ± 0.2	1	
11,968	HeroGum	0	11,968		Empty	26:36:23 ± 50	77.64	27:19 ± 3	20.68	1
					Filled	9:01 ± 1:05		1:37 ± 2	9.49	1
reports	Hybrid	0	11,786	11,968	Empty	1:04:29 ± 17	3.14	2:20 ± 0.5	1.77	11.68
					Filled	0:20 ± 0.3		0:15 ± 0.2	1.46	6.52

fault of one type may contain faults of other types as well, and thus each run need not start with a completely empty cache.

In the worst case, in the case of **Lock**, HeroGum takes about 26 times longer than HeroDiff, when each of them runs its cache filling phase in parallel. However, in the result of HeroGum, there is no manual correlation to perform. As each manual correlation may take from a few seconds to a few minutes and is error-prone, HeroGum is an efficient and practical alternative to automate the process.

Compared to HeroGum, the hybrid strategy HeroDiffGum is up to 11 times faster. But it still takes up to 3.46 times longer to perform the correlation than HeroDiff with an empty cache, and 1.6 times longer with a filled cache. Nevertheless, the results are obtained in less than 8 minutes for the full set of **Lock**, **NullRef** and **Null** faults with an empty cache, and, as with HeroGum, there is no manual correlation to perform.

Finally, we consider the space usage. As noted in Section III-C, to further reduce the computing time and the space consumption, we cache the edit scripts generated by GumTree in the serialized form of an optimized pre-parsed OCaml object that can be used directly by the correlation algorithm of Herodotos. For the selected patterns, 18,804 edit scripts are generated for the 19,456 files that have faults. Fewer edit scripts are generated than files, because no script is generated for the faults identified in the last considered Linux version. For our dataset, the cache stores 5.8GB of data. We keep this cache in an in-memory filesystem rather than on disk, for efficiency, as a dataset this size can be easily handled by current hardware.

B. Improvements in the Correlations when using GumTree

We have studied in detail the **Lock** reports where HeroDiff proposes correlations and one of the propositions turns out to be correct. We regard these cases as failures of HeroDiff. The reasons for these failures are summarized in Table II.

TABLE II. HERODIFF **Lock** CORRELATION FAILURES

Reason	Instances
changed nesting level (indentation)	6
code moved	4
confusion due to other nearby changes	2

A common reason why HeroDiff fails on **Lock** reports is due to whitespace changes, typically because some code is moved under or out of a conditional. Such a case is illustrated by our example patch of Figure 2. A solution would seem to be to simply ignore whitespace. Nevertheless, the positions reported by Coccinelle and tracked by Herodotos include both line and column information. Thus, whitespace must be taken into account in order to compute the right position prediction and keep tracking the fault. In contrast, whitespace changes are easily handled by HeroGum, as GumTree identifies the code as being moved.

The second most common reason why HeroDiff fails on **Lock** reports is due to blocks of code that are moved, typically complete functions. Indeed, it is common to *e.g.*, identify the need for a new functionality or implementation strategy and reorganize the contents of a file as a result. When a function's position changes by a large number of lines, GNU Diff typically considers that the function is completely removed at its original position and added at its new position. These changes will include the tracked positions, so HeroDiff will fail to automatically find the fault at its new position, and falls back to the manual correlation. In contrast, HeroGum will mark the fault and its surrounding nodes as being moved to a new container node, and will thus be able to continue to track the code fragment.

In the remaining two cases, there are many changes in the surrounding code, and GNU Diff gets disoriented. In one case,[8] most of the body of a function, except the lock code, is factored out into a separate function that is inserted before the original one. GNU Diff aligns the original function with the factored-out one, and thus the lock calls are not aligned, resulting in them being considered to be removed in one version and added in the next. GumTree is able to make the correlation, probably because the lock is only locked at two places in the file, and it matches up the calls in the order in which they appear. In the

[8] `drivers/gpu/drm/radeon/radeon_ring.c`, versions 2.6.34 and 2.6.35, function radeon_ring_lock

```
--- linux-2.6.14/sound/mips/au1x00.c
+++ linux-2.6.15/sound/mips/au1x00.c
@@ -477 +475 @@
-        spin_lock(au1000->ac97_lock);
+        spin_lock(&au1000->ac97_lock);
```

Fig. 6. Locking function argument renaming

other case,[9] GNU Diff is disoriented by a block of declarations that are moved from the function containing the fault to the previous function in the file. GNU Diff aligns on this block of declarations, causing the fault code to be considered to be first removed and then added. GumTree can detect the move of the declarations independently from the other code found in the function, and correlates the lock calls.

Not all failures of the correlation process manifest themselves as correlations that have to be carried out manually. There are also false negatives. HeroDiff has a number of heuristics that allow it to consider that fault reports match or do not match even when the affected code is part of a changed region, by using features such as a relevant keyword found in the fault report (the locking operation in the case of a **Lock** fault) or the length of the term at the position marking the fault (Section II-B). These heuristics can cause a correlation not to take place when it should, which results in a fault silently being considered to be fixed in one version and introduced in the next one. We have seen three such cases for the **Lock** faults, where the path for accessing the lock changes. One case is illustrated in Figure 6, where a field is changed from a pointer to an inlined structure, leading to the addition of an `&` operator. This changes the length of the tracked argument, and, as described at the end of Section II-B, Herodotos does not propose any correlations. In the two other cases, both in the same file, an intermediate substructure is added, adding the need for an extra field dereference.[10] GumTree is able to overlook these minor changes, based on the notion of container, which focuses on larger regions of commonality in the surrounding code.

C. Degradations in the Correlations when using GumTree

GumTree intrinsically eliminates all of the manual correlations, as it either proposes a unique mapping for each node, or indicates that the node (and thus the fault) has been removed, and thus there are no choices left for the user. This, however, does not mean that all of the choices made by GumTree are correct.

The main source of difficulty is GumTree's strategy for detecting moved fragments of code. While we have seen that this detection can allow GumTree to resolve some cases on which HeroDiff has to resort to manual intervention, such as a move of a complete function, the strategy can also be too aggressive and make incorrect choices. For the **Lock** faults, we have observed 7 cases for which GumTree makes such a link that is incorrect. Typically, this arises when a new function is added or removed that is similar to another function that remains in the file. GumTree may choose the wrong instance as the counterpart of a fault. In another case, however, both calls to a function in a file were removed and one was added, and a fault was correlated from one of the old calls to the unique new one, even though there was no real connection, besides the name of the called function, between them. Many of these cases did not require a correlation for HeroDiff, because GNU Diff does not detect code moves and thus is highly constrained by the linear structure of the two files. Finally, for the Lock fault, in 3 cases, GumTree does not make a correlation, presumably finding the code to be too different between one version and the next. In these cases, GumTree starts a new fault report, thus slightly inflating the overall number of fault reports. We have noted, however, that HeroDiff is also susceptible to this error.

We now discuss some other typical cases from the **NullRef** and **Null** reports for which the correlation results are different when using HeroGum than when using HeroDiff.

1) Large Hunks: In Linux 2.6.38, at line 1,008 of the file `drivers/staging/easycap/easycap_ioctl.c`, a null test of the variable `file` is performed after a dereference. This test is removed, along with 38 other lines around it, and replaced by another test, within a sequence of 5 added lines. Moreover, the new test is about `file` being different from null, rather than identical to null, as done before. GumTree fails to correctly associate the Linux 2.6.38 test with the Linux 2.6.39 test. Indeed, the test in the version 2.6.38 is considered removed. In the case of GNU Diff, as the test is part of the removed lines, Herodotos falls back to the manual algorithm to find a fault of the same kind in the next version of the file.

We observe the same problem in a rewrite of more than 350 lines in the file `fs/xfs/xfs_alloc.c` from Linux 2.6.34 to Linux 2.6.35. A **Null** report is issued at line 2,571 of the Linux 2.6.34 code. Both GNU Diff and GumTree fail to propose the correct new position, 2,700. HeroDiff again ultimately succeeds due to the step of manual intervention.

2) Variable/Function Renaming: In Linux 2.6.36, in the file `drivers/staging/hv/channel_mgmt.c`, a large number of functions and variables were renamed to conform to the Linux coding standards. There are thus small changes everywhere, but always within tokens; there is no change to the code structure. This issue is illustrated by the test of `msgInfo` in the hunk illustrated Figure 7. Here, even if the change renames only a single variable, and the hunk is concise (only 3 lines are added and removed), neither HeroGum nor HeroDiff automatically correlates the reports. In the case of HeroGum, the thresholds used by default by GumTree cause it to fail to identify these renamings. As a result, it produces a lot of removed and added blocks in the edit script, rather than updated code. As Herodotos only propagates changes when code is updated or moved, Herodotos stops the correlation and the fault is assumed to be removed. HeroGum thus reports that the corresponding fault in the next version is a new one. HeroDiff again reports the need for a manual correlation, and the user can make the correct choice.

To overcome the problem raised by GumTree where renaming causes nodes to be annotated as removed instead of updated, we increase the threshold on the maximum number of descendants in the third step of the GumTree algorithm (see Section II-A) for which the expensive but accurate tree-differencing algorithm is launched. This change increases the

[9] `mm/slab.c`, versions 2.6.23 and 2.6.24, function cpuup_callback.

[10] `net/ipv4/igmp.c`, versions 2.6.8 and 2.6.9, functions `igmp_mc_get_next` and `igmp_mc_get_next`

```
1 --- linux-2.6.36/drivers/staging/hv/channel_mgmt.c
2 +++ linux-2.6.37/drivers/staging/hv/channel_mgmt.c
3 @@ -809,3 +819,3 @@
4 -       if (msgInfo) {
5 -               kfree(msgInfo->WaitEvent);
6 -               kfree(msgInfo);
7 +       if (msginfo) {
8 +               kfree(msginfo->WaitEvent);
9 +               kfree(msginfo);
```

Fig. 7. Variable renaming case

number of renamings detected by GumTree, allowing HeroGum to correctly correlate the faults.

3) Selection of the Recorded Positions: For the **NullRef** pattern (dereference of a pointer followed by a null test on the pointer), a fault is characterized by two positions: the pointer dereference and the pointer test. The latter is highlighted in the **NullRef** fault report. This poses problems if a new test on the same value is introduced in the control flow between these two positions; the fault remains but the reported position changes. As the original test is preserved, both approaches successfully compute the new position of the original test, but the fault is no longer reported here; it is now reported at the new test.

As an example of this issue, we consider the **NullRef** fault appearing in the hunk on Linux 2.6.31 shown in Figure 8. The `page->index` dereference at line 13 appears before the `page` test of line 26. However, after updating the code, Coccinelle reports the fault with the dereference of line 8 and the `page` test of line 15. As the line 26 is unchanged, HeroDiff and HeroGum fail to properly track the fault. HeroDiff again falls back on manual correlation, but HeroGum, which has no fallback mode, considers the fault to be removed from Linux 2.6.31 and a new one to be created in Linux 2.6.32. Overall, in this case, the issue is related to the choice of position to be tracked, as Herodotos tracks only one position per fault. The selection of the position to be tracked is thus crucial to the success of the correlation process for faults that involve multiple code fragments.

4) Code Refactoring: As shown in Figure 9, from Linux 2.6.28 to Linux 2.6.29, the file `fs/ext4/mballoc.c` has been rewritten so that the `kmem_cache_alloc` function is no longer called in the function `ext4_mb_free_metadata` (line 4), but rather in the function `ext4_mb_free_blocks` (line 23). This latter function then calls the former (line 31). As the allocation may fail, calling the `kmem_cache_alloc` function may return a `NULL` pointer which must be tested. However, the pointer returned in line 4 in the old version is directly used the next line. Similarly, the returned pointer of the new call, on line 23, is not tested.

The `kmem_cache_alloc` call is part of a hunk, so GNU Diff fails to provide useful information to HeroDiff which then falls back on the manual process to track the **Null** fault. In our previous work, it seems that the relation between the old and new code was not obvious and the proposed correlation was erroneously tagged. On the contrary, GumTree successfully tracks this refactoring, and the new position reported allows HeroGum to properly track the fault.

Similarly, we observed a **Null** fault involving calls to the `kvm_mmu_get_page` function that may return `NULL`

```
1 --- linux-2.6.31/fs/fscache/page.c
2 +++ linux-2.6.32/fs/fscache/page.c
3 @@ -549,23 +704,35 @@
4      goto superseded;
5    page = results[0];
6    _debug("gang %d [%lx]", n, page->index);
7 -       if (page->index > op->store_limit)
8 +       if (page->index > op->store_limit) {
9 +         fscache_stat(...);
10          goto superseded;
11 +       }
12
13 -       radix_tree_tag_clear(..., page->index,
14 -         FSCACHE_COOKIE_PENDING_TAG);
15 +       if (page) {
16 +         radix_tree_tag_set(..., page->index,
17 +           FSCACHE_COOKIE_STORING_TAG);
18 +         radix_tree_tag_clear(..., page->index,
19 +           FSCACHE_COOKIE_PENDING_TAG);
20 +       }
21
22 +       spin_unlock(&cookie->stores_lock);
23        spin_unlock(&object->lock);
24 -       spin_unlock(&cookie->lock);
25
26    if (page) {       ...
```

Fig. 8. Introduction of a new test for null pointers

```
1 --- linux-2.6.28/fs/ext4/mballoc.c
2 +++ linux-2.6.29/fs/ext4/mballoc.c
3 @@ -4419,5 +4713,0 @@ ext4_mb_free_metadata
4 -  new_entry = kmem_cache_alloc(..., GFP_NOFS);
5 -  new_entry->start_blk = block;
6 -  new_entry->group  = group;
7 -  new_entry->count = count;
8 -  new_entry->t_tid = handle->...->t_tid;
9 @@ -4610,4 +4890,19 @@ ext4_mb_free_blocks
10 -  if (metadata) {
11 -    /* blocks being freed are metadata. ...
12 -     * be used until this transaction is ... */
13 -    ext4_mb_free_metadata(...);
14 +  err = ext4_mb_load_buddy(sb, blk_group, &e4b);
15 +  if (err)
16 +    goto error_return;
17 +  if (metadata && ext4_handle_valid(handle)) {
18 +    struct ext4_free_data *new_entry;
19 +    /*
20 +     * blocks being freed are metadata. ...
21 +     * be used until this transaction is ...
22 +     */
23 +    new_entry = kmem_cache_alloc(..., GFP_NOFS);
24 +    new_entry->start_blk = bit;
25 +    new_entry->group  = blk_group;
26 +    new_entry->count = count;
27 +    new_entry->t_tid = handle->...->t_tid;
28 +    ext4_lock_group(sb, blk_group);
29 +    mb_clear_bits(..., bitmap_bh->b_data,
30 +      bit, count);
31 +    ext4_mb_free_metadata(...);  ...
```

Fig. 9. Code refactoring, displacing an untested call to `kmem_cache_alloc`

pointers. At line 2,247 of the file `arch/x86/kvm/mmu.c` in Linux 2.6.36, this function is called from the function `mmu_alloc_roots` (64 lines) without a subsequent test of the returned value. The function `mmu_alloc_roots` was refactored in Linux 2.6.37 to untangle two allocating modes. The new version calls conditionally either the function `mmu_alloc_direct_roots` (34 lines) or the function

`mmu_alloc_shadow_roots` (94 lines), according to the allocating mode to use. The refactoring involves untangling a simple and short version for the *direct* case; the remaining code belongs to the *shadow* case. Additionally, new features are added in the *shadow* case. With HeroDiff, the manual process is used and the correlation was tagged properly by the user. With HeroGum, both functions are considered mainly new by GumTree; the nodes are marked as added. Thus, HeroGum fails as GumTree does not correctly represent the refactoring.

In conclusion, neither HeroDiff nor HeroGum is uniformly better for corner cases, and the better one depends on the case considered. However, HeroGum outperforms HeroDiff in terms of correct and automatic correlations in the general case.

V. Related Work

We first present the prior work on the topic of AST differencing, and then consider the main work that has been done on tracking code patterns.

A. AST-Differencing

The best-known differencing algorithm that works on ASTs is ChangeDistiller [19]. This algorithm is largely inspired by the algorithm of Chawathe et al. [16] that computes differences on trees representing hierarchical text documents, such as LaTeX documents, but is tuned to work better on ASTs. However, ChangeDistiller is based on the assumption that leaf nodes contain a significant amount of text. Therefore, it uses simplified ASTs where the leaves are complete code statements, rather than raw ASTs. Therefore, ChangeDistiller will not compute differences at a finer granularity than statements, unlike GumTree. Since fault reports are often located in a precise part of a statement, GumTree seems more suited to track their positions.

Several other AST-differencing approaches have been proposed. The DiffTS algorithm [20] works on raw ASTs. The developers of DiffTS showed that it efficiently produces short edit scripts. However the results of this algorithm have not been validated by humans. The VDiff algorithm [21] generates edit scripts from Verilog HDL files. It is somewhat similar to the first phase of GumTree, but it additionally uses the lexical similarity of the code. However, the generated edit scripts are specific to the VHDL language. Finally, the JSync [22] algorithm is also able to compute edit scripts that include move actions. However it relies on a classical text differencing algorithm applied to the text generated from the traversal of the ASTs as a first step, which limits its ability to find moved nodes. Additionally, it focuses on producing information about clones rather than producing edit scripts.

B. Tracking Code Patterns

Duala-Ekolo *et al.* [23] have introduced a technique to track the location of code clones across several versions of a file. Their approach is based on a so-called *Clone Region Descriptor* (CRD), which can be seen as a URL for source code. A CRD contains information about the file, class, method and blocks in which a code fragment is located. Using this information, the fragment can be tracked across versions as long as its associated block is still present in the file. However, a CRD cannot track a code fragment at a finer granularity than a whole block of code, such as a single expression, and is not robust to changes in the nesting level of the containing block. On the contrary, our approach is able to track code patterns at any granularity.

Canfora *et al.* [24] have introduced an approach that tracks code lines across two versions of a file. Their approach is more advanced than that of GNU Diff as it is able to detect added, deleted, changed and moved lines. The approach first applies a classical GNU Diff algorithm to detect unchanged code lines, and then searches for additional mappings between the lines contained in the hunks detected by GNU Diff. Similarly, Reiss [25] studies the effectiveness of several methods to track code lines across two versions of a file. He finds that the most efficient method, called W_BESTI_LINE, is to combine a line similarity measure with a similarity measure of the context (the four surrounding lines). Finally, Asaduzzaman *et al.* [26] improve the efficiency of the W_BESTI_LINE strategy by using GNU Diff and simhashes, *i.e.*, hash values for which the distance denotes the degree of similarity of the objects being hashed, to avoid comparing every possible pair of lines. They also add heuristics that are capable of detecting lines that have been split. However, all these approaches work at the granularity of text lines. On the contrary, our approach is able to track code patterns at any granularity.

Logozzo *et al.* [27] have developed Verification Modulo Versions (VMV), which is integrated into the tool `cchecker` for C# programs. The goal of VMV is to reduce the number of alarms reported by static verifiers. It does so by leveraging the program history to eliminate false positives in the reports of new warnings. The VMV algorithm computes conditions on functions and compares them between versions. There is no correlation based on LCS or tree-matching. There is thus the strong assumption that functions are not renamed. Our main target for Herodotos is to track faults, but we keep the fault finding tool (Coccinelle) separate from the fault tracking one (Herodotos). Our fault finding process thus does not use previous fault finding results, but it does not make any assumption about code changes. Finally, their evaluation focuses on the reduction in false positives, without considering the running time or the rate of success of the correlation process.

Göde and Koschke [28] propose an incremental clone detector that leverages knowledge about clones in one version to help find clones in the next version. This approach is reported to be quicker than the traditional one of searching for clones in each version independently, "if the changes do not exceed a certain fraction of the source code." Their approach propagates the clones directly for the unchanged files, and then only searches for clones in the remaining files. Our approach is more fine-grained, in that it tracks positions across unchanged regions within files that may contain other modifications.

VI. Conclusion

Using software metrics for taking development decisions is part of the modern software engineering methodologies. Some of the interesting metrics involve tracking code fragments over several software versions. In previous studies, we have used Herodotos and the LCS-based and line-oriented GNU Diff algorithm to help accomplish this task. While this has allowed us to efficiently track code fragments over a decade in an

almost automatic way, the remaining correlations have to be done manually. This manual process is a daunting task which is tedious and error-prone. As the size of the code base increases, either because more versions are studied, or because each version gets larger, the situation gets worse.

In this study, we propose the HeroGum code tracking algorithm, which leverages the tree-based matching provided by the GumTree tool. Our evaluation shows that HeroGum fully automates the correlation process, but significantly increases the computing time and introduces some miscorrelations.

To alleviate the performance problem, we propose the HeroDiffGum hybrid strategy that first runs HeroDiff and then runs HeroGum only for the correlations that have failed. In doing so, our solution allows a fully automated correlation of the code fragments, while keeping the computing time reasonable. While HeroGum takes almost an hour and a half with 64 CPU cores for the reports of the three fault types considered in our evaluation, the hybrid strategy, HeroDiffGum, completes the work in about eight minutes.

Addressing the miscorrelations of HeroGum remains future work. We plan to consider how a level of confidence can be introduced into GumTree, to highlight cases where it has had to choose a matching code fragment from several similar possibilities. In the case of lower confidence correlations, HeroGum could report the possibilities to the user for manual correlation, as does HeroDiff, but where the list of possibilities would be more refined, based on code commonalities. We furthermore note that the number of errors observed with HeroGum in the studied **Lock** case is lower than the number of correlations manually proposed to the user with HeroDiff for the same set of reports.

As future work, we plan to compare the HeroGum and HeroDiffGum strategies with other tree-based algorithms. It would also be interesting to characterize the amount of information needed in the AST for tracking different kinds of faults. This would reduce the amount of text generated by cgum, and should reduce the amount of work performed during the tree-matching phase of GumTree. Alternatively, we could consider how to specialize the output of GumTree to the minimum information required to track positions, rather than preserving the entire AST. This would reduce the memory usage (for the cache in the in-memory file system) and the position-searching process of Herodotos. Finally, it would be interesting to generalize our code tracking approach to be able to track multiple positions representing more complex code artefacts.

REFERENCES

[1] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in Linux device drivers," in *EuroSys 2008*. Glasgow, Scotland: ACM, Mar. 2008, pp. 247–260.

[2] "Static source code analysis, static analysis, software quality tools by Coverity Inc." http://www.coverity.com/, 2008.

[3] Y. Xie and A. Aiken, "Saturn: A SAT-based tool for bug detection," in *Computer Aided Verification*. Springer, 2005, pp. 139–143.

[4] Z. Li and Y. Zhou, "PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code," in *ESEC/FSE*, Lisbon, Portugal, 2005, pp. 306–315.

[5] J. R. Cordy and C. K. Roy, "The NiCad clone detector," in *Program Comprehension (ICPC), IEEE 19th International Conference on*. IEEE, Jun. 2011, pp. 219–220.

[6] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *Software Engineering, IEEE Transactions on*, vol. 28, no. 7, pp. 654–670, 2002.

[7] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code," *Software Engineering, IEEE Transactions on*, vol. 32, no. 3, pp. 176–192, 2006.

[8] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *ICSE*, Minneapolis, MN, USA, 2007, pp. 96–105.

[9] Free Software Foundation Inc., "GNU diffutils," http://www.gnu.org/software/diffutils/.

[10] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller, "Faults in Linux: Ten years later," in *Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2011)*. Newport Beach, CA, USA: ACM, Mar. 2011, pp. 305–318.

[11] N. Palix, G. Thomas, S. Saha, C. Calvès, G. Muller, and J. Lawall, "Faults in Linux 2.6," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, pp. 4:1–4:40, 2014.

[12] N. Palix, J. Lawall, and G. Muller, "Tracking code patterns over multiple software versions with Herodotos," in *AOSD*. Rennes and Saint Malo, France: ACM, Mar. 2010, pp. 169–180.

[13] S. Nadi, C. Dietrich, R. Tartler, R. C. Holt, and D. Lohmann, "Linux variability anomalies: What causes them and how do they get fixed?" in *MSR*, San Francisco, CA, USA, 2013, pp. 111–120.

[14] M. Kim and D. Notkin, "Program element matching for multi-version program analyses," in *MSR*, Shanghai, China, 2006, pp. 58–64.

[15] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Montperrus, "Fine-grained and accurate source code differencing," in *ASE*, Vasteras, Sweden, 2014, pp. 313–324.

[16] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," in *Proceedings of the 1996 International Conference on Management of Data*. ACM Press, 1996, pp. 493–504.

[17] Y. Padioleau, "Parsing C/C++ code without pre-processing," in *International Conference on Compiler Construction (CC'09)*, York, UK, Mar. 2009, pp. 109–125.

[18] OCaml developers, "Module Obj," http://caml.inria.fr/pub/docs/manual-ocaml/libref/Obj.html, Nov. 2014.

[19] B. Fluri, M. Würsch, M. Pinzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Trans. Software Eng.*, vol. 33, no. 11, pp. 725–743, 2007.

[20] M. Hashimoto and A. Mori, "Diff/TS: a tool for fine-grained structural change analysis," in *WCRE*. IEEE, 2008, pp. 279–288.

[21] A. Duley, C. Spandikow, and M. Kim, "Vdiff: a program differencing algorithm for Verilog hardware description language," *Automated Software Engineering*, vol. 19, no. 4, pp. 459–490, 2012.

[22] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Clone management for evolving software," *IEEE Trans. Software Eng.*, vol. 38, no. 5, pp. 1008–1026, 2012.

[23] E. Duala-Ekoko and M. P. Robillard, "Clone region descriptors: Representing and tracking duplication in source code," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 1, p. 3, 2010.

[24] G. Canfora, L. Cerulo, and M. Di Penta, "Tracking your changes: A language-independent approach," *IEEE Software*, vol. 26, no. 1, pp. 50–57, Jan. 2009.

[25] S. Reiss, "Tracking source locations," in *ICSE*, May 2008, pp. 11–20.

[26] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and M. Di Penta, "LHDiff: A language-independent hybrid approach for tracking source code lines," in *ICSM*, Eindhoven, The Netherlands, 2013, pp. 230–239.

[27] F. Logozzo, S. K. Lahiri, M. Fähndrich, and S. Blackshear, "Verification modulo versions: Towards usable verification," in *PLDI*, Edinburgh, UK, 2014, pp. 294–304.

[28] N. Gode and R. Koschke, "Incremental clone detection," in *CSMR*. Genova, Italy: IEEE, Mar. 2009, pp. 219–228.