N-way Diff: Set-based Comparison of Software Variants

Slawomir Duszynski
Fraunhofer Institute for Experimental Software Engineering (IESE) *
Kaiserslautern, Germany
slawomir@duszynski.it *

Vasil L. Tenev
Fraunhofer Institute for Experimental Software Engineering (IESE)
Kaiserslautern, Germany
vasil.tenev@iese.fraunhofer.de

Martin Becker
Fraunhofer Institute for Experimental Software Engineering (IESE)
Kaiserslautern, Germany
martin.becker@iese.fraunhofer.de

***Abstract*—Software is frequently developed in many similar copies, called forks or cloned software variants. During this development, pairwise comparison is routinely used for finding differences between the cloned copies, assessing their similarity, and merging the content. However, analyzing the similarity of a large group of variants using pairwise comparison is a relatively difficult task, as the number of compared pairs grows quadratically with the number of variants. Furthermore, the result of such group of pairwise comparisons is difficult to visualize.**

In this paper, we discuss the problem of N-way comparison of cloned software variants. We represent the N-way comparison result as a model of N intersecting sets. By aggregating the sets along the system decomposition hierarchy, we construct the sets at every level of the system structure (files, folders, and whole systems). We define a generalized approach for set model construction, and instantiate it for an N-way diff on the textual code representation. We propose set-based visualizations for the N-way comparison, which scale for more than ten component variants and MLOC-sized components. We evaluate the approach by applying it to several groups of industrial software system variants and by performing a controlled experiment with a comparison of 5 software forks. In the experiment, the group using set-based comparison solved the tasks in 58% less time and with 92% fewer incorrect answers than the group using pairwise comparison. Finally, we propose a generalization of the approach beyond software, to enable set-based comparison and similarity visualization for hierarchically structured models and data, for example genomes.

Keywords—software comparison, software reuse, similarity, set model, set visualization, software variability, product lines

I. Introduction

A comparison of two source code trees is a routine task in software development. Typically, the two compared folder trees are displayed side by side, with overlay icons representing additions, deletions, or changes inside the content of every folder and file. By navigating the folder tree, the user can explore the differences, down to the atomic content level of the text lines inside a code file. At this lowest level, the text differences between two files are found using the diff [1] algorithm and indicated using text highlighting. The two-way comparison is easy to understand, works for code trees of any realistic size, and enables data exploration by providing both abstract and detailed views on the differences. In a general sense, the same kind of comparison is also available for other artefact types such as software models [2][3], genomes [4], statistical data [5], any many more.

Problem. In many science and engineering applications, the problem of characterizing similarities and differences among many artefact variants arises. This is for example the case for comparing the code of many software variants to assess reuse potential [6][7][8][9], for comparative analysis of many genomes [10][11][12][13], or for comparing the results of multiple medical test methods [14]. If none of the input variants is distinguished (e.g., as a base or reference to which the others should be compared), and there is no meaningful ordering of the variants (e.g., such as time ordering of subsequent code revisions), a comparison of N variants needs to represent all $\frac{N(N-1)}{2}$ pairwise comparison results. Only after all these pairs are compared, it becomes possible to identify common content occurring in all the variants, unique content found in just one variant, or to find groups of variants similar to each other, as well as the variants dissimilar from the rest.

Goal of the paper. In this paper we discuss the problem of N-way comparison, which we previously confronted in our work on software variants and branching [15][16][17][18][19]. We propose a set-based approach to measure and visualize similarity information across N software implementation variants, for any level of abstraction: from complete systems, through components, folders and files, down to individual content elements. The approach can be used in maintenance and reengineering of software forks. It supports finding components highly similar across variants, and identifying groups of variants particularly similar to each other. It enables result exploration from high-level abstraction down to detail across any selection of input variants or their fragments. The approach can accommodate different definitions of similarity, provided by different analysis algorithms. Hence, the focus of this paper is the set-based representation and visualization of similarity information for N-way comparison, and the generic methods to construct this representation for compared assets.

Contribution. We define the N-way comparison result as a set of tuples of equivalent elements (and not pairs, as in the pairwise comparison case), and propose the data structure of hierarchical set models, based on trees and intersecting sets, which represents the N-way comparison results. We also propose set-based visualizations displaying for which variants and elements the content is similar, and how high that similarity is. While we discuss algorithms to construct the comparison result, the main contribution of this paper is thus the N-way comparison framework, consisting of the data structure and the visualizations. The comparison framework can be utilized by various similarity detection algorithms – we describe an instantiation of the approach in the form of an N-way diff.

We generalize the approach to N-way comparison of any hierarchically structured content (e.g., using folders, packages, sections) containing equivalent atomic content elements (text lines, tokens, model elements). So far, we applied the approach mainly to software and text comparisons. However, we hypothesize that it could also be applied to other content types found in software development, data science, or genetics and bioinformatics.

Finally, we evaluate the set-based comparison approach in a controlled experiment, where the group using set-based N-way software comparison solved the analysis tasks in 58% less time and with 92% fewer incorrect answers than the group using pairwise comparison. We also report application experience of the approach to industrial software system variants, illustrating its scalability for many software variants of large sizes. The contributions of this paper are therefore:

* This work was created while the author was with Fraunhofer IESE.

978-1-7281-9914-6/20/$31.00 ©2020 IEEE
DOI 10.1109/VISSOFT51673.2020.00012

- Definition of N-way comparison approach, based on tuples of equivalent elements, the data structure of hierarchical set models, and set-based visualizations.
- Instantiation of the approach and discussion of the algorithms for an N-way diff.
- Generalization of the approach beyond software.
- Evaluation of the approach in a controlled experiment and practical application to industrial software variants.

Paper structure. In Section 2, we discuss the related work on N-way software comparison. In Section 3, we introduce a running example of 5 compared software systems, and present a set-based approach to file content comparison. In Section 4, we address folder and system comparison. In Section 5, we present set-based similarity visualizations. We report the controlled experiment results in Section 6, and the industrial application experiences in Section 7. In Section 8 we discuss the benefits and drawbacks of the proposed set-based N-way comparison approach, and the generalization of the approach beyond software. Section 9 concludes the paper.

II. Related Work

The development of software components or whole systems in many parallel variants or forks is a common phenomenon. In open source, it occurs for Android app development [20], for projects hosted on collaboration platforms such as Github [21], as well as for whole operating systems such as the BSD family [22]. In industrial software development, cloning of software systems is practiced as a form of reuse [6]. At the same time, maintaining parallel variants incurs redundant efforts [6][23]. This motivates a multitude of approaches for analysis and reengineering of parallel software variants, with goals such as finding or tracing features [24][25], supporting maintenance of forks [26][27][28], merging the fork implementations [29], or recovering product line architecture [30].

Focus on the similarity representation. This paper focuses on the exploration of similarity between software variants, both in the system structure dimension (e.g., to find which components are similar to components of other variants), as well as in the variant dimension (e.g., to find which variants are similar to each other). Hence, in this section we focus on the similarity representation aspect in the related papers.

Finding pairwise similar software variants. Yamamoto et al. [8], Yoshimura et al. [9], and Mende et al. [7] use clone detection to measure the share of similar code (in terms of clone coverage) between all pairs of software variants. In the first and third approach, also the similarity between corresponding components pairs across the variants is computed. The similarity metrics are presented in a square matrix (see Fig. 1 left for a schematic depiction). For visualization of such metrics, Kamiya et al. [31] propose to use scatterplots. Cordy [32] extends that idea by proposing live scatterplots, which can be expanded or collapsed to aggregate scatterplot rows and columns based on the system hierarchy, e.g., aggregating files to parent folders, and the folders upwards to the complete system (see Fig. 1 right). On user demand, the scatterplots provide detailed data for each scatterplot cell.

However, this representation which is based only on pairwise similarity hides important information when used for three or more variants, such as for example the size of the common code shared by all analyzed variants. The pairwise similarity matrix is also ambiguous, as the same matrix can be

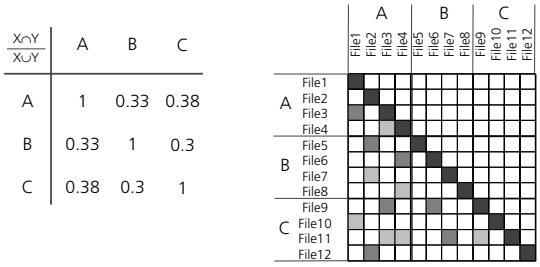

Fig. 1 Example of a pairwise similarity matrix for three systems A, B, C (left), and its visualisation using expandable scatterplots (right). The color intensity in a scatterplot cell corresponds to the value of the simiarity metric.

constructed for different distributions of similarities across three or more variants. Both these points are illustrated in Fig. 2 below, where two groups of three software variants, represented by sets A, B, and C, are analyzed. In the group on the left side of Fig. 2, the complete common code is similar across all three sets, while on the right side of Fig. 2 only one element is similar across three sets. However, for both situations the cardinalities of the three sets A, B, C, as well as the cardinalities of A∩B, A∩C and B∩C are the same, so that the same Jaccard similarity values result, and the same similarity matrix is created (in the middle of Fig. 2). This illustrates that the pairwise similarity does not deliver full information about the analyzed variants. In contrast to that, a set-based representation can provide information about the similarity of any possible variant group, as illustrated later in this paper.

Chen et al. [33] compare tens of thousands of mobile apps, to find software products that are likely cloned, by computing system-level similarity. However, they do not investigate the similarity of internal app components. Hemel and Koschke [34] compare a group of forks to a reference implementation. However, the forks are not compared to each other, so this approach cannot detect component variants that are similar between forks but not similar to the reference. The Diffuse file differencing tool [35] applies the same principle by comparing N file variants to a reference file but not against each other.

Finding reusable components. Koschke et al. [36], Wu et al. [37] and Shatnawi et al. [30] recover the architectures of the variant systems and measure component similarity to identify common and variable components. However, these approaches focus on the membership of the components in variants, and do not report how far the different implementations of the components are similar.

Analysis of N software versions is a problem different from an N-way comparison, because the versions are ordered in time. Hence, the techniques for multi-version analysis (Eick et al. [38], Kagdi et al. [39], Hurter et al. [40], Telea and Auber [41]), focus on comparing the N-1 pairs of consecutive versions: the difference between any other two versions can be treated as a sum of the intermediate changes. Without the time ordering however, all $\frac{N(N-1)}{2}$ possible variant pairs need to be compared.

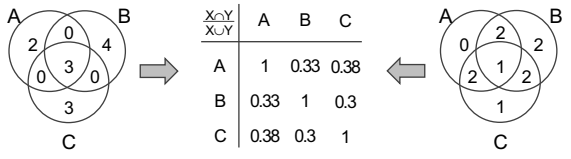

Fig. 2 Similarity of two different groups of systems A, B, C, represented by intersecting sets (left and right) and by a pair-wise similarity matrix (center). While the set models allow for distinguishing the different situations, the pair-wise matrix is less informative as it shows the same values for both cases.

N-way matching and merging. Rubin and Chechik [42], Reuling et al. [43], and Schlie et al. [44] propose algorithms for finding similar elements across N software models. Rubin and Chechik group the matching elements into tuples, while the other approaches provide an equivalent solution by annotating the merged elements with variability information. Ishio et al. [45] define an approach to detect similar folders and files across software systems. Sakaguchi et al. [46] construct a unified directory tree which matches the corresponding directories even if some of them were renamed or moved in the variant systems. We see the finding and matching approaches as complementary to our work. The matched system components and content elements, found using these approaches, can be used to construct a set model and corresponding visualizations in our comparison framework.

Visualization of sets. The approaches for visualization of sets and set intersections, such as these described by Lex et al. [47], Lamy and Tsopra [48], and Alsallakh and Ren [49] are complementary to our work: they can be used to visualize a set model for N variants of software systems or components. Several further such approaches are presented by Alsallakh et al. [50] in their survey of the state of the art of set visualization.

Our previous work. This paper presents a redefinition of our former analysis approach [15]. We now refine and extend the approach, use the set models as the basic data structure, define set-based visualizations, and experimentally evaluate the benefits of set-based similarity representation. More details about the approach are available in a PhD thesis [16], and the tooling aspects are described in a separate paper [19].

III. N-Way Content Comparison with a Set Model

In our approach, we combine two main ideas. First, we represent variant software components as intersecting sets of atomic content elements. The elements similar across any K variants belong to the respective intersection of corresponding K sets. Second, we aggregate the sets along the system structure hierarchy, constructing the sets for container elements (e.g., folders) based on the sets of their constituent parts (e.g., files). This section details the set model construction for files, while the next section concerns the hierarchical aggregation of the file-based models for folders.

System structure. For the purpose of comparison, we assume that the input system variants are structured according to the model in Fig. 3: they form a tree of *Container* elements (e.g., folders, packages), where the leaves of the tree (e.g., files, classes) are *Content Units* as they contain the actual content of the system (e.g., source code). The content itself is then formed by *Content Elements* (e.g., text lines, tokens) which are the base elements for content comparison and size measurement. We define no further assumptions on the nature of *Structure Tree Elements* and the *Content Elements*. For example, we do not define how the *Content Elements* are stored inside a *Content Unit*: they can be stored as an ordered list (e.g., as text lines in a file), as a tree (e.g., as an abstract syntax tree of tokens), or in an unordered form (e.g., as classes in an UML diagram). The depicted structure is generic and can represent system variants in the form of a source code tree, software model, or others.

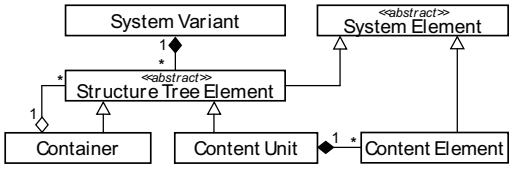

Fig. 3 An UML diagram of the generalized system variant structure.

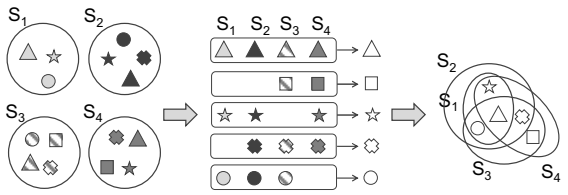

Fig. 4 Four *Content Units*, each from a different system variant, contain similar *Content Elements*. The set model is constructred by forming tuples of similar *Content Elements*, and placing the tuples in the set intersections.

Set model construction is illustrated in Fig. 4 for four example *Content Units*, each belonging to a different system variant. We treat the *Content Units* as sets of their contained *Content Elements*, and use a comparison function to identify tuples of similar *Content Elements* between the variants. Each tuple contains at most one *Content Element* from each variant, and each *Content Element* belongs to exactly one tuple. Then, the tuples are placed in the set intersections of the set model according to the variant membership of the elements they contain: for example, a tuple containing only elements of S_3 and S_4 is placed in the $\overline{S_1}\overline{S_2}S_3S_4$ intersection.

The comparison function is an equivalence function, i.e., it is symmetric, reflexive and transitive. It defines which *Content Elements* are considered similar when the variant *Content Units* are compared. Hence, the equivalence function is a variable part of our approach: it needs to be defined depending on type of the analyzed content (text, model, etc.) and on the required notion of content similarity. In Fig. 4, the similarity is based on the shape of the elements, but not on their fill pattern. Similarly, in the comparison of text lines the equivalence can consider the text content, but ignore the formatting. In Section 4 we introduce a second comparison function, matching tuples of *Structure Tree Elements* across the variants.

Example N-way comparison. We now illustrate the set-based comparison for an example group of five software systems, depicted in Fig. 5. In this case, the system structure consists of folders (as *Containers*) and files (as *Content Units*), and the *Content Elements* are text lines. The example comparison function is an N-way diff, which is based on the output of pairwise diff for all file variant pairs. As diff uses the Longest Common Sequence algorithm [1][51], two text lines are only matched if they are identical and are part of the common sequence (for example, in sequences ABC and CAB, the longest common sequence is AB, and C is not matched). To construct the set model for N variants,

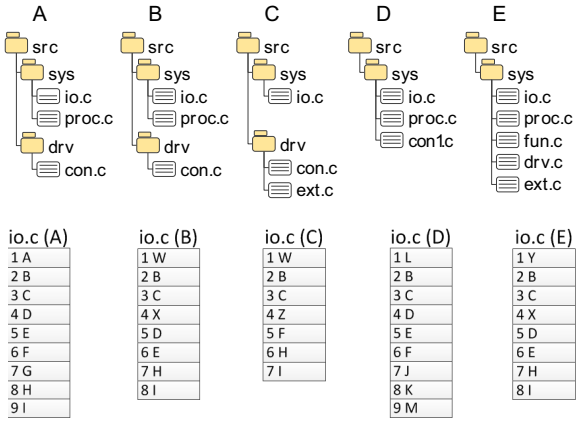

Fig. 5 Five example system variants, which differ slightly in their folder structure and the file content (top). Abstracted content of five variants of the *io.c* file (bottom).

the function creates a tuple of text lines if and only if every pair of lines within the tuple was identified as similar by the pairwise diff. The bottom part of Fig. 5 depicts an abstracted content of the *io.c* file in five variants. Each small rectangle represents a text line, a number within a rectangle is the consecutive line number within that file, and the letters symbolize the line content.

Fig. 6 shows pairwise comparison of the five *io.c* file variants, where ten file pairs are compared. This result is complex to analyze: although the matches and differences for every file pair are shown, it is difficult to identify facts related to a higher number of variants. For example, it requires time to find text lines common across all variants, or unique to just a single variant. In the N-way diff, we aggregate the pairs of matched lines shown in Fig. 6 to tuples: two lines are in the same tuple if and only if a pairwise match between them exists. The resulting set model is depicted in the left part of Fig. 7. Using the set model, it is easy to recognize in which variants the particular lines occur: for example, the lines B and C are common to all five variants, lines H and I occur in all variants except variant D, and line Y is unique to variant E. It is also visible that all the lines of variant B occur in other variants, while the variant D has the highest number of unique lines (4).

Bar diagrams. To visualize the basic similarity metrics for N intersecting sets, we use a bar diagram with N+1 bars, as depicted in the right part of Fig. 7. Each of the N bars corresponds to a different set, and the last bar depicts the union of all N sets. The length of each bar is equal to the size of the respective set. Each bar is divided into three parts, with their lengths equal to the number of set elements falling into three categories: core (elements belonging to all N sets), shared (belonging to 2..N-1 sets), and unique (belonging to just 1 set). The legend for the used colors is provided in Fig. 8. The bar diagram provides a quick overview over the amount of elements falling into each category for each set, and over the sizes of the sets and their union. It is particularly useful for a higher number of sets, when a Venn diagram such as the one in the left part of Fig. 7 becomes difficult to read.

Showing element membership. For each compared element it is known to which sets its corresponding tuple belongs. This information can be visualized in their Content Unit, by coloring the elements according to their set membership as shown in Fig. 8. Hence, this visualization is an N-way analogy to the match and difference information visualized in Fig. 6. Furthermore, the elements belonging to any group or intersection of sets (e.g., shared with other selected set) can be shown in that view using an additional color, as discussed below.

Set theoretic operations on the set model can be performed on user demand to investigate the variant similarity. A visualization of an example operation result, using an additional color in the bar diagram, is shown in Fig. 9. As described by Alsallakh et al. [50], set-typed data is suitable for a large variety of similarity analysis tasks, for example:

- Find or count elements which fulfill a condition on set membership, e.g. elements in set A and in B but not in C.
- Compare set and set intersection cardinalities, e.g., |A| with |B|, or |A∩B| with |A∩C|, and identify the sets and intersections with the largest or smallest cardinality.
- Analyze set inclusion relations, e.g., whether set A is fully included in B, or in B∪C.
- Analysis of element reuse: for a specific element, find or count all sets in which that element occurs.
- Find or count elements in a set with a specific degree of reuse: e.g., elements unique to a set, shared by a given group of sets, shared by at least/exactly K sets, or by all sets.

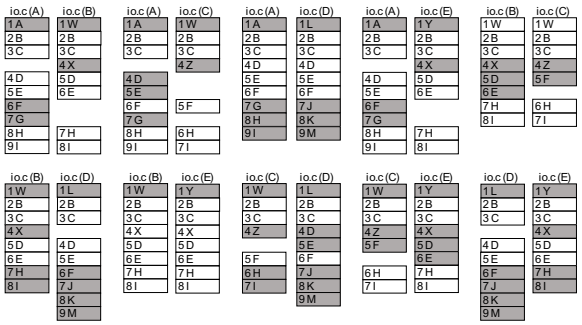

Fig. 6 Pairwise comparison of the five *io.c* file variants. White text lines represent matches, grey text lines represent differences.

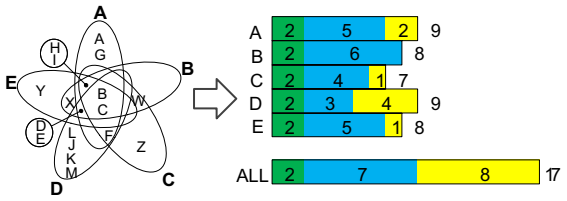

Fig. 7 The set model constructed for five variants of the *io.c* file (left). A bar diagram visualizing the similarity between the *io.c* variants (right).

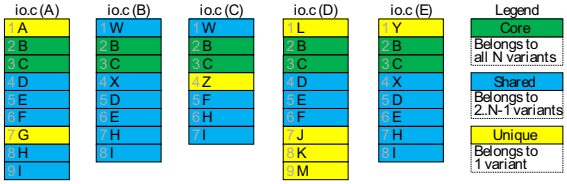

Fig. 8 The content of the five file variants colored according to the set membership of the text lines.

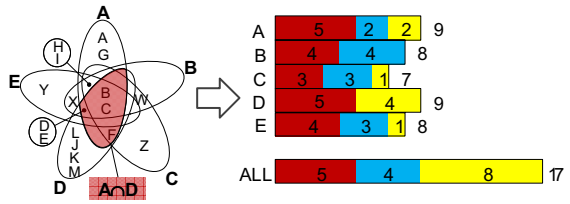

Fig. 9 An example set theoretic operation on the set model (left) and its visualization in the bar diagram using an additional color (right).

- Compare sets based on element reuse, e.g. check if set A contains more unique elements than B, or more elements shared with C, or more elements shared with 1 or K other sets.

In our approach we construct set models for every element of system structure tree, as well as for whole variant systems. The above analysis tasks, applied to these set models, are highly relevant when analyzing software variants with the goal of finding reusable components or finding similar variant groups. With the set model, performing these analysis tasks is much easier than in the case of pairwise comparison.

Non-transitive similarity. In the above case of N-way diff, a set model for N variants is constructed from $\frac{N(N-1)}{2}$ pairwise comparison results. To create a tuple, the match graph between the element pairs is used. If the graph is complete, all its elements form the tuple. However, the match graph can also be incomplete, that is, the similarity relation built from the matches can be non-transitive. Fig. 10 depicts an example case of incomplete match graph (left), as well as an incomplete match graph containing two elements from the same file variant (right).

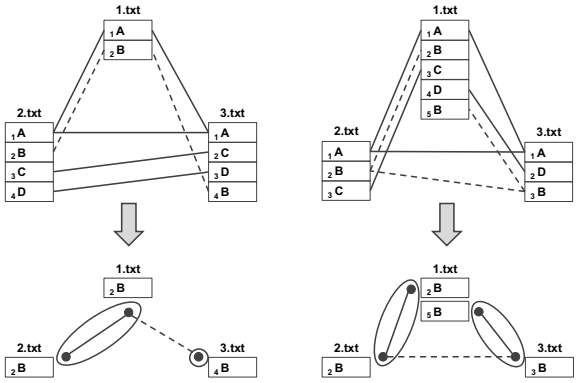

Fig. 10 Top: two examples of pairwise diff comparisons which result in transitive (full lines) and non-transitive (dashed lines) match graphs. Bottom: cliques found in the non-transitive graphs are used to form tuples.

To build element tuples from a non-transitive match graph, we find cliques (complete subgraphs) in the graph, and form the tuples from these cliques, as illustrated in the bottom part of Fig. 10. With this approach, each two elements in every tuple are pairwise similar, and the match edges which do not belong to any clique are ignored. The process of finding the cliques can be optimized towards different, partially conflicting criteria: finding the largest cliques, finding the least amount of cliques, or minimizing the amount of original graph edges not included in any clique. Algorithms for each of these criteria exist [16]. In our implementation, we iteratively use the largest clique algorithm.

While a strongly non-transitive similarity cannot be fully represented with a set model, as some similarity matches need to be ignored, we found that this problem is not significant in the practice when building the set model based on pairwise diff. We analyzed 9 groups of variant systems, each having from 4 to 15 variants. For each group, we needed to ignore at most 0.75% of the pairwise text line matches to obtain fully transitive similarity and build the set model [16].

IV. Comparing Structure Trees and System Variants

The previous section details the construction of a set model for a group of *Content Units*, such as files, on an example of an N-way diff. In this section we continue using that example to illustrate the set-based comparison for *Containers* (folders) and complete system variants, structured hierarchically according to the model in Fig. 3.

Finding similar files. As the system variants were cloned (forked) from each other, a large part of their tree structure could remain identical. However, some *Content Units* or *Containers* could be renamed or moved in the structure tree. Hence, a comparison of a group of system variants starts with the search for corresponding *Content Units* and *Containers*, based on their similarity, across the system variants. As in the case of content comparison, we model the similarity of structure trees by forming tuples of tree elements: each such tuple contains at most one *Content Unit* or *Container* from each variant tree, and each *Unit* or *Container* belongs to exactly one tuple. Hence, in the case of N-way diff a tuple represents a group of matched files or folders. Naturally, we only match *Units* to *Units* and *Containers* to *Containers*. In result, a unified structure tree is constructed from the variant trees, as illustrated on the example in Fig. 11. The unified structure tree contains a *Unit* or *Container* for each tree location and element name which occurs in any of the compared trees. The tree elements which have different names or locations, but are matched to each other, are modeled as hard links analogously to hard links in a file system. This means that such hard link elements in the unified tree lead to the same tuple of matched tree elements and to the same set model, as shown by the color coding of the elements in Fig. 11. In the opposite case, when two variant elements having the same name and system-relative location are not matched, the unified structure tree contains two different, not linked elements, with a suffix string added to their name for unambiguous identification.

The search for similar files and folders can be performed using many algorithms. As in the case of equivalence function for content elements, the file search algorithm is a variable part of our approach, which can be exchanged depending on the required notion of tree and file similarity. The simplest file matching algorithm is based on the equality of element names and system-relative paths, hence matching elements located in the same locations of the variant trees. In our experience, this algorithm is sufficient for many industrial software variant groups, where the move or rename operations were rare.

To search for renamed or moved files, we extend the existing algorithms for DNA aligning. We define an algorithm for pairwise aligning of file trees, based on textual file similarity. Then, we iteratively use the Gusfield's Center Star method [52] to construct a multiple tree alignment. We document the multiple tree alignment algorithm in detail in [17]. Further algorithms, such as those discussed in the related work section, can be likewise applied. We require however that the algorithm returns tuples of matched elements, to enable the unified structure tree construction described above.

Hierarchical set aggregation. The set models of *Content Elements* are first constructed for the tuples of matched *Content Units*, as described in Section 3. Afterwards, we

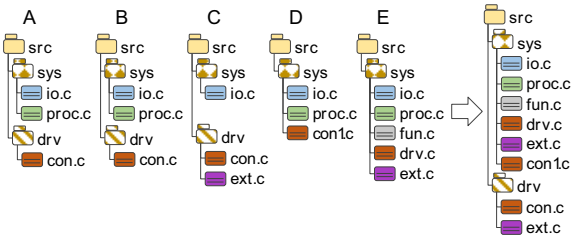

Fig. 11 The unified structure tree (right) for example systems (left). Each tuple of matched files or folders is indicated by a different color – for example, the files *con.c*, *con1.c* and *drv.c* are matched in the same tuple. The unified tree contains hard links between matched elements having different name or location.

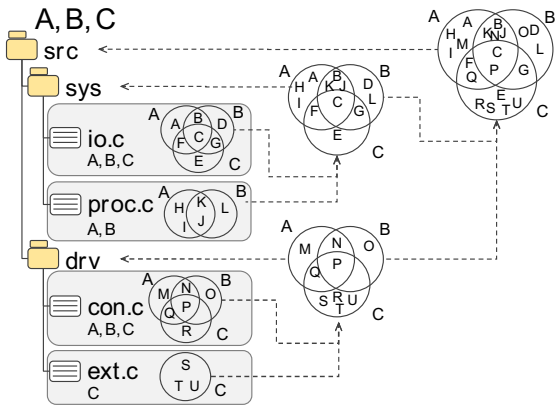

Fig. 12 Hierarchical aggregation of set models, using the unified structure tree for variants A, B, C of the example system variant group.

create the set models for *Containers*: such a model contains all *Content Elements*, which are contained in the *Content Units* belonging to the *Container* (as direct children, or through child *Containers*). This aggregation method is illustrated in Fig. 12 for the files and folders of the example system variant group: the set model for the *sys* folder contains the text lines of the *io.c* and *proc.c* files, the model for the *drv* folder is built using the models of *con.c* and *ext.c* files, and the model for the root folder contains all text lines of every variant of every file. As the content similarity between folder variants, as well as between whole system variants, is expressed using set models, the same analysis tasks, the same set theoretic operations and the same set-based visualizations can be applied likewise to files, folders, and system variants.

Inclusive and exclusive aggregation. The set aggregation principle described above is straightforward for unified structure trees which are created by the equal name and location algorithm. In such a name-based unified structure tree, the containment relation between the tuples forms a tree: for any tuple of *Containers* $T_{CONTAINER}$ and any tuple of *Content Units* T_{UNIT}, either every member of T_{UNIT} has a parent in $T_{CONTAINER}$, or none of them has. However, that property does not hold for unified trees containing hard links: as illustrated in Fig. 13, the folder tuple of *sys* contains the tuple of *con.c* and *con1.c* files, but for the *conc.c* file variants the parent folder is *drv*, belonging to a different folder tuple. In this case, we can use an inclusive or exclusive aggregation method to construct the set models for folders, as illustrated in Fig. 13:

- In **the inclusive method**, a set model of a $T_{CONTAINER}$ tuple is built from the set models of all T_{UNIT} tuples in which at least one member *Content Unit* is a child of any of the $T_{CONTAINER}$ members.
- In **the exclusive method**, a set model of a $T_{CONTAINER}$ tuple is built from the set models of all T_{UNIT} tuples in which all member *Content Units* are children of any of the $T_{CONTAINER}$ members.

Hence, the inclusive aggregation method includes the set models of T_{UNIT} tuples where some of the member *Content Units* are not children of $T_{CONTAINER}$ members, while the exclusive method excludes such tuples from the aggregation.

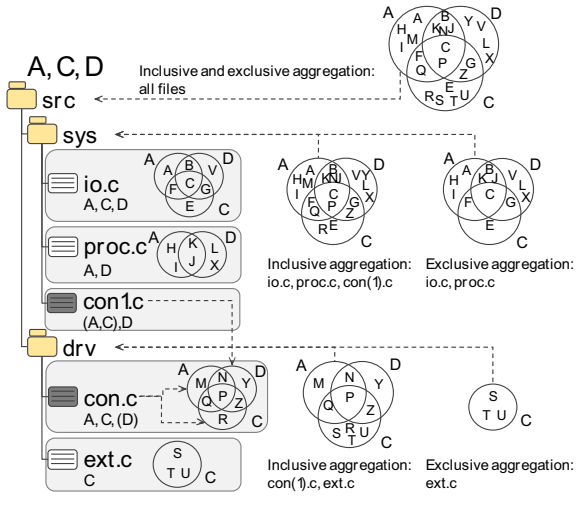

Fig. 13 File *con1.c* in variant D is matched to the file *con.c* in variants A and C. The respective structure tree nodes are treated as a hard link, connected to the same set model (left). The set models for folders can be aggregated using the inclusive or exclusive approach (right).

Naturally, the set model of a given T_{UNIT} tuple is aggregated just once, even if the tuple contains many hard links to that model. Note that both aggregation methods produce the same result for the root folders of the system variants, as these folders are parents of all files in their trees.

V. Set-Based Similarity Visualization

In this section we present N-way comparison visualizations based on the unified structure tree and the set models. Since the set models are available for *Content Units*, *Containers*, and complete system variants, the same visualizations can be used for all these elements. The visualized data is an N-way diff analysis of six systems from the BSD Unix family, i.e. 386BSD 0.1, FreeBSD 2.0.5, NetBSD 1.1, NetBSD 1.2, NetBSD 1.3, and OpenBSD 2.0. For each visualization we indicate the analysis task it supports by underlining the task description.

Unified structure tree view. To navigate through the unified structure tree, we use a system structure view in a UML-like representation (Fig. 14). Initially, the view shows the children elements of the root folder. The folders can be further expanded to show their contained elements. Each tree element shown in the view displays its name, the number of variants it belongs to (in the bottom left corner), number of lines in the set union of its associated set model (bottom right corner) and a graphic showing the proportion of core, shared and unique code in the set union. On user demand, also the result of a set theoretic operation, e.g. counting set elements fulfilling a specific membership condition, can be visualized on every element using an additional color in the bar diagram. By selecting any element in this view, the user can display further visualizations of the similarity information, listed below.

Tree map structural diagram is an alternative way of system structure visualization (Fig. 15). A tree map [53] displays all elements of a system tree or a component subtree, visualized as rectangles and nested according to the tree hierarchy. The rectangles have areas proportional to the size of the respective

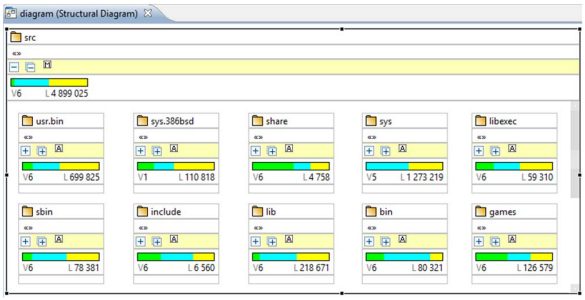

Fig. 14 A screenshot of the navigable visualization of the unified structure tree for the group of six BSD Unix systems.

Fig. 15 A screenshot of the tree map diagram of the unified structure tree. The intensity of red color in an element corresponds to the proportion of the containted code that belongs to at least 4 variant systems.

set model, and their color indicates the value of a user-selected metric, such as the proportion of code fulfilling a set membership condition. Hence, a tree map supports identification of elements with an interesting combination of sizes and metric values, while also indicating whether these elements are located nearby in the system structure.

Bar diagrams, such as presented in Fig. 7 in Section 3, are available for every unified structure tree element. The bar diagram can display the result of arbitrary set theoretic operation, as shown in Fig. 9. Hence, it is used to count and compare the cardinalities of sets and any selection of set intersections. For convenience, the sizes of displayed bar areas are also provided in a table. In Fig. 16, we present the bar diagram for the root folder of six example BSD Unix systems.

Tree map set diagram is a size-preserving visualization of all existing set intersections. Hence, it provides another way to count and compare set intersection cardinalities. In this diagram, we combine the ideas of a Venn diagram [54] and of a tree map [53]. Fig. 17 shows a Venn diagram of 4 software system variants, and a corresponding tree map set diagram. Instead of showing structure hierarchy elements, the tree map areas are used to display set intersections. The name of each area indicates its membership in the input sets (in Fig. 17 we use binary name coding for readability), and the area size corresponds to the cardinality of the set intersection. In contrast to the Venn diagrams, this visualization can present the set intersections for a higher number of sets in an understandable way, while also graphically indicating the relative area sizes. Furthermore, it uses similar shapes for the shown areas, which facilitates visual comparison. Different diagram layouts (grouping of intersections in the visualized tree structure) can be used depending on the analysis task. Fig. 18 shows the diagram for six example BSD systems. We provide more details on this visualization in [16]. Similar visualization was independently published later by Alsallakh and Ren [49].

Content view, already schematically drawn in Fig. 8 in Section 3, is used for viewing the similarity information on the lowest detail level of source code. Fig. 19 shows a screenshot from the tool, annotated with explanations of the visualization mechanisms. The set membership of each text line (core, shared, unique, or a user-defined set theoretic operation) is indicated with line background coloring and an icon – except for lines which were ignored during the analysis, such as empty lines. A tooltip showing set membership details is provided on user demand. Hence, the defined visualizations show the set-based similarity information on every level of system hierarchy, from the system structure root down to every single content element.

Phylogenetic distance diagram, shown in Fig. 20, supports identification of system and folder variant pairs and groups which exhibit particularly high similarity. The diagram uses the branch layout to visualize the identified variant groups, and indicates the relative similarities between and within the groups using the branch length [55]. Hence, it provides a third way of investigating set similarity, together with the bar diagrams and the tree map set diagram.

Phylogenetic variant-version diagram, shown in Fig. 21, uses the set similarity information to reconstruct a probable evolution history of the analyzed system, folder or file variants. We start its construction by ignoring all set intersections with cardinality falling below a defined threshold – by default, 1% of the union code size. Then, we construct a Hasse diagram of the remaining intersections and lay out the diagram as a tree, with branch lengths proportional to the sizes of particular intersections [18]. Hence, the diagram shows an inclusion

System	Component	Core	Shared	Unique	All Lines	No Query
386_0_1		278 875	170 575	727 631	1 177 081	0
FreeBSD_2_0_5		278 875	988 847	389 662	1 657 384	0
NetBSD_1_1		278 875	1 227 498	166 488	1 672 861	0
NetBSD_1_2		278 875	1 592 317	12 405	1 883 597	0
NetBSD_1_3		278 875	1 786 267	637 312	2 702 454	0
OpenBSD_2_0		278 875	1 902 853	491 130	2 672 858	0
Union_Model		278 875	2 195 522	2 424 628	4 899 025	0

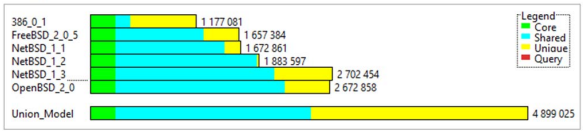

Fig. 16 A bar diagram screenshot for the example group of six BSD systems. "Query" is a user-defined set theoretic operation.

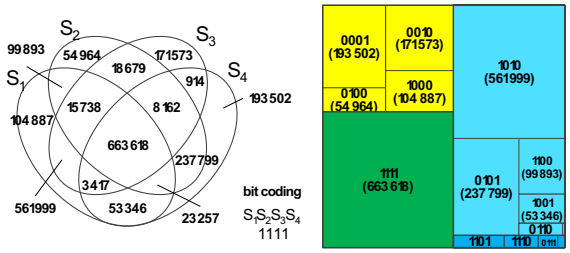

Fig. 17 A Venn diagram for four intersecting sets (left). A tree map set diagram for the same sets, using color coding as in the bar diagram (right).

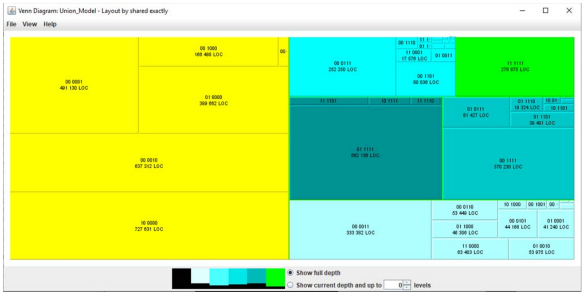

Fig. 18 A screenshot of the tree map set diagram for the root folder of the example six BSD systems.

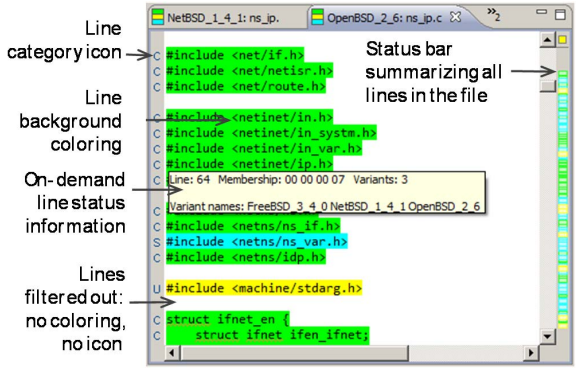

Fig. 19 An annotated screenshot of code-level similarity visualization with line background coloring, category icons, and on demand details.

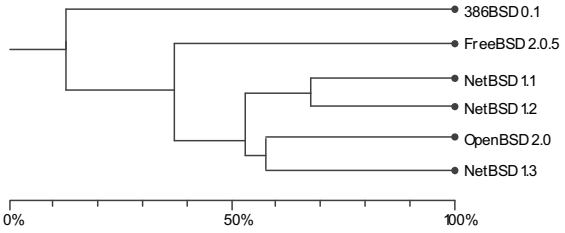

Fig. 20 Phylogenetic distance diagram (dendrogram) for the BSD systems. Location of branching points corresponds to the similarity of the tree branches.

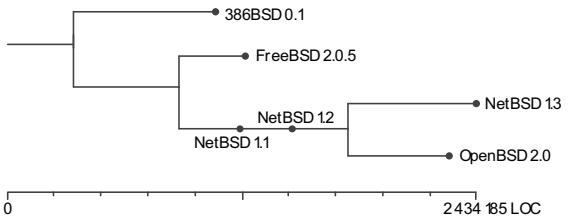

Fig. 21 Phylogenetic variant-version diagram for the six BSD systems. The length of branch sections is proportional to the amount of shared or unique code.

hierarchy of the sets and the larger intersections. The diagram layout coincides with the software evolution history: as the software typically grows with time, an earlier software version contains less code than a later one, and the code of the earlier version is almost fully included in the code of the later version (see the three analyzed NetBSD versions for an example). Hence, the diagram shows these versions as successors on a single branch. In contrast to that, two cloned variants developed in parallel both contain a larger amount of unique code, and appear in the diagram as parallel branches. Consequently, the diagram helps to distinguish component versions from variants, and indicates the relative changes between them. In most cases, the Hasse diagram can be reduced to a tree. If the reduction is not possible, the alternative locations of a variant can be displayed as parallel, alternative tree branches.

A table view of the similarity information can be obtained by exporting the basic set model metrics, such as the amount of core, shared and unique code, to an Excel table for all unified tree elements. This is useful for tasks such as sorting and filtering the tree elements according to metrics values.

VI. A Controlled Experiment on Set-Based Comparison

The core idea of our approach is the use of set similarity model for N-way comparison. We postulate that the similarity abstraction in the form of a set model is easier to analyze and understand than the similarity abstraction of pairwise comparison. In this section we describe a controlled experiment we performed to evaluate this claim. A detailed experiment description, including all documents and the raw data, is provided in the thesis [16].

Experiment goal. An evaluation of complete comparison approaches would not determine if the measured differences were caused by the use of the set model, the abstractions, the visualizations, or the different user interface, but would only provide results for a combination of those. Consequently, in the experiment we decided to evaluate just the core idea of our approach, the set similarity model, in isolation from the other factors. The experiment goal [56] was therefore to:

TABLE I. The Goal of the Controlled Experiment

Analyze	the pairwise and set-based similarity models
for the purpose of	comparison
with respect to	analysis efficiency, correctness and cognitive load
from the viewpoint of	a software developer
in the context of	Software Product Lines university course, with students analyzing file variants for code similarity.

The experimental hypotheses we evaluated were:

- **H1 Efficiency.** The use of the set similarity model reduces the effort for analyzing similarity information as compared to the use of pairwise comparison model (metric: analysis time).
- **H2 Correctness.** The use of the set similarity model allows for understanding the similarity information with a higher correctness compared to the use of pairwise comparison model (metric: ratio of incorrect answers).
- **H3 Cognitive Load.** The use of the set similarity model allows for analyzing the similarity information with a lower cognitive load compared to the use of pairwise comparison model (metric: the SMEQ scale [57]).

Experiment Process. The experiment participants were 22 computer science students attending the Software Product Lines course at the Technical University of Kaiserslautern. None of the students had a prior contact with our approach or tool. We presented the experiment procedure to all participants, and then randomly assigned them to one of the two groups of 11 students each: the treatment group, using only the set similarity model, or the control group, using only the pairwise similarity model. After the groups split, each participant received an identical printed document containing the introductory information, the briefing questionnaire, the experimental tasks, and the debriefing questionnaire. After answering the briefing questions, they received a printed tool tutorial, which was also presented to them as a slide show. The tutorial was identical for both groups, except for the part concerning the similarity model. After the tutorial, the participants familiarized themselves with the tool and answered two sample warm-up questions. Then, all participants were individually solving the experimental tasks. Finally, they filled out the debriefing questionnaire.

Independent and Dependent Variables. The only independent variable varied between the experimental groups was the used similarity model: pairwise or set-based. All other differences were removed: the groups solved identical tasks, used identical documents, and worked in parallel in two equivalent laboratories. The same system variants were analyzed by both groups, and the used similarity information was technically the same – we verified that all the pairwise similarity relations were included in the constructed set model. The dependent variables investigated in the experiment are the analysis effort, the answer correctness, and the cognitive load.

Tasks. The participants analyzed the code similarity in files of five system variants, written in Java. The systems contained ca. 20 files each in a simple package structure. The participants answered 16 questions, printed in the experiment documents, concerning file variant similarity. The terms used in the questions (similar code, common or unique code) were explained before in the tutorial. Example questions were:

- Which two variants of the file *EclipseFigure.java* are the most similar to each other?
- Which variants of the file *UndoableTool.java* have identical code?

All questions needed only a short answer, such as stating the names of the variants. To view the similarity information, the students used a reduced variant of our analysis tool, in which we disabled all visualizations and mechanisms except for the system hierarchy navigation (as in Fig. 14, but with no similarity information) and the code view (as in Fig. 19). Hence, the students were only able to locate the files in the system structure diagram, identical for both groups, and to view the code of file variants in the code editor. In the editor, the background of the displayed code lines was colored according to the similarity information provided by the pairwise (control group) or set-based (treatment group) similarity model. The icons and tooltips provided for each line were also model-dependent. Except for these differences, all other user interface mechanisms were identical for both groups. The participants were not allowed to use other tools, but could take notes on paper.

The briefing questionnaire focused on the participant background: field of study, semester, and color blindness. We further asked about participants' experience in programming and in the use of comparison methods and tools, all rated on a five-point Likert scale. The differences between the groups were not statistically significant (two-tailed Mann-Whitney U test at $p = 0.05$), except for one question: the control group had more experience in using diff tools than the treatment group. Hence, the control group was more experienced in a method similar to the one they used in the experiment. However, as reported below, the control group achieved consistently worse task results. Hence, we consider the different experience to not influence the hypothesis evaluation, as the control group result would be probably worse if its members had less experience.

The debriefing questionnaire focused on the experienced cognitive load (discussed below) and a number of control questions using five-point Likert scale: whether the participants understood the tasks, used the tool as intended, and had sufficient time. The 22 participants confirmed that they understood the tasks and the tool and had sufficient time, with no statistically significant difference between the groups.

Hypothesis testing. In Fig. 22 we use boxplots to present the results of time and answer correctness measurements. All participants from the treatment group, using the set model, finished their tasks faster (maximum: 18 minutes) than the fastest participant from the control group (25 minutes). The treatment group participants needed on average 14.0 minutes to complete the tasks (median: 14.0 min, σ=2.28 min), while the control group participants needed on average 33.7 minutes (median: 32.0 min, σ=7.38 min). One control group participant did not provide the finishing time, so we report the task time results for a group size of 10. However, we know that this participant was neither the fastest, nor the slowest in the group.

The **task correctness** was higher in the set group (Fig. 22), where one participant made 2 errors and all others provided fully correct answers (sum: 2, average: 0.18, median: 0.0, σ=0.60). In the pairwise (control) group, only three participants provided correct answers for all 16 questions, while the others made between 1 and 6 errors (sum: 25, average: 2.27, median: 2.0, σ=2.05).

To evaluate the **cognitive load**, we use the Subjective Mental Effort Question (SMEQ) scale [57], validated in usability research. The SMEQ presents a continuous scale, labeled in nine locations with categories ranging from "absolutely no effort" to "extreme effort" (see Fig. 23 left). The respondents indicate their subjectively felt cognitive load, experienced during the tasks, by placing a mark on the scale and then converting it to an integer value between 0 and 150. The SMEQ measurements are provided on an interval scale, as the category locations were psychometrically calibrated [57]. This allows a convenient response analysis, as the calculation of averages and distances is meaningful for interval scale data. In Fig. 23 we present the cognitive load results provided by the participants. The treatment group cognitive load (average: 19.0, median: 15.0, σ=9.06) was much lower than the control group load (average: 50.0, median: 48.5, σ=35.56). In the control group, one participant did not provide an answer. Consequently, we report the data for 10 group participants.

Table II presents the results of **hypothesis testing**. We tested the task time and cognitive load data with the Student's t-test, as they are normally distributed (Shapiro-Wilk normality test) and on at least interval scale. The task error series, where the data of the treatment group is not normally distributed according to the normality test, were tested using the Mann-Whitney U test. For all three hypotheses we also provide the p value. Finally, we quantify the observed improvement by comparing the averages for both groups: the treatment group needed 58% less time and made 92% fewer errors. All three evaluated hypotheses were accepted in the experiment. Hence, the experiment result indicates that the use of the set-based comparison allows for analyzing code similarity of file variants with a lower effort, higher correctness, and lower cognitive load, as compared to the use of pairwise comparison.

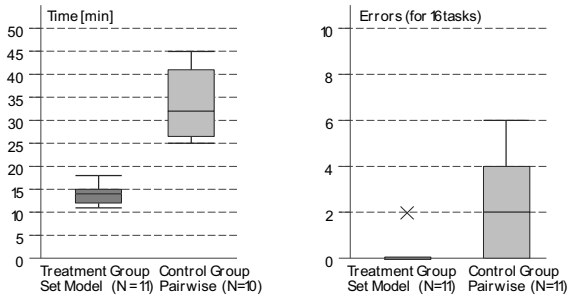

Fig. 22 Boxplots showing the experiment results: task time (left) and task errors (right) for the two groups.

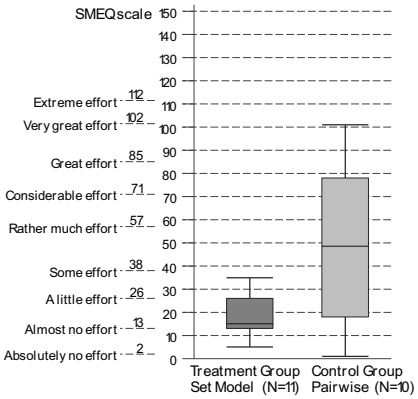

Fig. 23 The SMEQ scale for the cognitive load (left) and the boxplots showing the distribution of the load values for the two experiment groups.

TABLE II. STATISTICAL TESTING OF THE EXPERIMENT HYPOTHESES

Hypothesis	Accepted at p<0.05	p	Observed improvement
HS1 Efficiency	Yes (t-test)	3.7e-08	Avg.: 14.0 to 33.7→ 58.5%
HS2 Correctness	Yes (U test)	0.0048	Avg.: 0.18 to 2.27→ 92.1%
HS3 Cognitive load	Yes (t-test)	0.0057	Avg.: 19.0 to 50.0→ more than 1 category

Threats to validity. The **internal validity** concerns the degree to which the experiment outcome was caused by the treatment. We mitigated this threat by random assignment of participants to groups, and by removing all differences between the groups whenever possible. We consider the only difference, which was the higher experience of the control group in using diff tools, to not endanger the validity, as discussed above. We also mitigated the effects of learning a new method and tool by a pre-experiment tutorial and providing example analysis tasks.

The **external validity** (generalizability) of the experiment result is affected by the choice of experiment participants, compared systems, and tasks. The external validity threats can be mitigated by a replication of the presented experiment in different settings. It would be especially interesting to understand how the differences in effort and correctness of similarity understanding, using the set-based and pairwise methods, vary with a growing number of compared variants.

VII. Industrial Application

We implemented the N-way diff approach in our Variant Analysis tool [19], including all visualizations listed in Section 5. During the industrial consultancy projects at Fraunhofer IESE, we applied the approach to several groups of software systems. The goal of the analyses was to find reusable components and to identify if some variants cover the code of other variants. A representative choice of these systems is summarized in Table III, and described in more detail in [16].

User feedback. In all cases, the information provided by the analyses was assessed by the customers as useful in their maintenance decisions, and not possible to obtain with other means. The analyses confirmed the already known similarity, while revealing new, previously unknown similarity facts. The possibility to trace the component differences down to the code level using diff was considered to be very helpful.

Table III. Groups of Analyzed Industrial System Variants

Domain	Variants	Average variant code size	Core code size	Average unique code size
Machine construction	4	1319 KLOC	664 KLOC	131 KLOC
Power electronics	10	427 KLOC	161 KLOC	152 KLOC
Automotive	14	186 KLOC	132 KLOC	2 KLOC
Telecommu -nication	6	202 KLOC	145 KLOC	36 KLOC

VIII. Discussion

In this paper, we show how the **set-based similarity representation** can be constructed for the task of N-way comparison, even if the input similarity relation is not transitive and the input structure trees were modified. In turn, the set-based similarity representation enables many benefits for the N-way comparison. The similarity of sets is simple to understand, as it bases only on element membership in sets, set theoretic operations, and on counting the elements. Furthermore, using sets makes it possible to **reuse many of the existing approaches** for set processing and visualization. Set types data supports a broad range of analysis tasks, as discussed in Section 3. Intersecting sets scale well with the growing number of variants: they can be understood and visualized even if the number of sets is large (50 or more) [58]. Finally, the concept of sets is very generic and can be applied to many types of analyzed content.

Transitivity of the similarity relation is a prerequisite for expressing the similarity with sets. If the input similarity is not transitive, we propose to use a transitive subset of the similarity graph for set model construction. In the case of diff, the transitive subset retains 99.25% or more of the original edges of element similarity graphs. In our opinion, the advantages of using the set model and visualizations overweight the possibility of a minor underestimation of the found similarity. However, for each content type and similarity detection algorithm this tradeoff needs to be evaluated. Providing the percentage of ignored graph edges for each group of sets can inform the user about the degree of inaccuracy in the set-based representation of input similarity.

System structure. We show how to construct a set model on every level of the system structure hierarchy, in spite of different structures of the system variants. In consequence, the same set-based concepts and visualizations can be applied for files, components, and whole systems, both small and large (as shown for the MLOC-sized BSD systems). This results in a scalable and understandable abstraction of the analysis result, for both dimensions of the system size and of the number of variants.

Generalization. We used an example of an N-way diff to illustrate the set-based comparison. The diff algorithm has several drawbacks: for example, it cannot recognize text blocks moved within the file, and cannot recognize code having only semantic similarity. However, note that the particular similarity functions used for the tree structure matching and set model construction can be independently exchanged, while still using the overall analysis framework and the visualizations. For example, a variant of diff which recognizes moves [59] can be used. Furthermore, similarity functions known from clone detection can be applied if we represent the code not as text lines, but as tokens or syntax tree nodes. In general, **the set-based approach can be used for similarity analysis of any artifacts, not only software ones, given two conditions**. First, the artifacts should be decomposable in a tree structure, as defined in Section 3. Second, an equivalence function for tuple-matching needs to be defined for the tree structure elements and the basic *Content Elements*.

Performance and scalability measurements of the N-way diff are documented in [16]. For example, the 4 variants from the first row of Table III, where no search for renamed files was needed, were analyzed in 263 seconds on a standard laptop. Set theoretic operations on the resulting MLOC-sized set model were calculated in 102 ms, thanks to a fast implementation of set membership information with bit vectors. However, the search for renamed files requires more time: from a few minutes for mid-sized systems having hundreds of files, to a few hours for systems having thousands of files.

IX. Conclusion

In this paper we discuss the code-level comparison and similarity analysis of software system variants. We propose to represent the N-way comparison result as a model of N intersecting sets, and show how to construct the model for any level of the system hierarchy (files, folders, and whole systems). On an example of an N-way diff, we present the mechanisms for hierarchical aggregation of the set models in the structure tree, and define set-based visualizations supporting a range of similarity analysis tasks. We discuss the problems of non-transitive input similarity and of matching the elements of modified system structure trees. The proposed mechanisms and visualizations are general and can be applied for different kinds of input content (e.g., software, model, or genomes), different similarity analysis algorithms, and different approaches to structure tree matching across the variants.

We postulate that the similarity abstraction in the form of a set model is easy to analyze and understand. We performed a controlled experiment which indicates that the use of the set-based comparison allows for analyzing code similarity of file variants with a lower effort, higher correctness, and lower cognitive load, as compared to the use of pairwise comparison. We also obtained positive feedback from industrial collaborations.

An interesting **future work** is to define further instances of the described hierarchical set-based similarity analysis approach, especially for models and non-software content such as genomes. Moreover, the N-way software comparison can be extended by other comparison and matching algorithms, as well as by other set visualization techniques, such as those listed in the related work section. Furthermore, more evaluation is needed to investigate the benefits of set-based similarity analysis, and of the different similarity visualizations, from the psychology and program understanding point of view. Finally, in the future we would like to provide an open source version of our analysis framework.

REFERENCES

[1] J.W. Hunt and M.D. McIlroy, "An algorithm for differential file comparison". Computing Science Technical Report 41, Bell Laboratories, 1976.

[2] Z. Xing and E. Stroulia, "UMLDiff: an algorithm for object-oriented design differencing". *20th IEEE/ACM International Conference on Automated Software Engineering* (ASE '05), pp. 54—65, doi: 10.1145/1101908.1101919

[3] U. Kelter, J. Wehren, and J. Niere, "A generic difference algorithm for UML models". In *Software Engineering*, Lecture Notes in Informatics Vol. 64, pp. 105—116, 2005.

[4] T. J. Carver, K. M. Rutherford, M. Berriman, M.-A. Rajandream, B. G. Barrell, and J. Parkhill, "ACT: the Artemis comparison tool," Bioinformatics, vol. 21, no. 16, pp. 3422–3423, June 2005, doi: 10.1093/bioinformatics/bti553

[5] E. Heinzen, R. Lennon, and A. Hanson, "arsenal: An arsenal of 'R' functions for large-scale statistical summaries. The comparedf function." [Online]. Available: https://cran.r-project.org/web/packages/arsenal/vignettes/comparedf.html Last visited: 22 June 2020.

[6] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An exploratory study of cloning in industrial software product lines". *17th European Conference on Software Maintenance and Reengineering* (CSMR 2013), pp. 25–34, doi: 10.1109/csmr.2013.13

[7] T. Mende, F. Beckwermert, R. Koschke, and G. Meier, "Supporting the grow-and-prune model in software product lines evolution using clone detection". *12th European Conference on Software Maintenance and* Reengineering (CSMR 2008), pp. 163–172, doi: 10.1109/csmr.2008.4493311

[8] T. Yamamoto, M. Matsushita, T. Kamiya, and K. Inoue, "Measuring similarity of large software systems based on source code correspondence". *6th International Conference on Product Focused Software Process Improvement* (PROFES 2005), pp. 530–544, doi: 10.1007/11497455_41

[9] K. Yoshimura, D. Ganesan, and D. Muthig, "Assessing merge potential of existing engine control systems into a product line". 2006 *International Workshop on Software Engineering for Automotive Systems* (SEAS '06), pp. 61–67, doi: 10.1145/1138474.1138485

[10] X. Argout, J. Salse, J.M. Aury, M.J. Guiltinan, G. Droc, J. Gouzy, et al., "The genome of theobroma cacao". *Nature Genetics*, vol. 43, no. 2, pp. 101–108, Dec. 2010, doi: 10.1038/ng.736

[11] K. Jahn, H. Sudek, and J. Stoye, "Multiple genome comparison based on overlap regions of pairwise local alignments," *BMC Bioinformatics*, vol. 13, no. S7, Dec. 2012, doi: 10.1186/1471-2105-13-s19-s7

[12] Z. Madak-Erdogan, T.-H. Charn, Y. Jiang, E. T. Liu, J. A. Katzenellenbogen, and B. S. Katzenellenbogen, "Integrative genomics of gene and metabolic regulation by estrogen receptors α and β, and their coregulators," *Molecular Systems Biology*, vol. 9, no. 1, p. 676, 2013, doi: 10.1038/msb.2013.28

[13] M. Wang, Y. Zhao, and B. Zhang, "Efficient test and visualization of multi-set intersections". *Scientific Reports*, vol. 5, no. 1, Nov. 2015, doi: 10.1038/srep16923

[14] H. M. Walline, C. Komarck, J.B. McHugh, S.A. Byrd, M.E. Spector, S.J. Hauff, et al., "High-risk human papillomavirus detection in oropharyngeal, nasopharyngeal, and oral cavity cancers - comparison of multiple methods". In JAMA Otola*ryngology–Head & Neck Surgery*, vol. 139, no. 12, pp. 1320–1327, Dec. 2013, doi: 10.1001/jamaoto.2013.5460

[15] S. Duszynski, J. Knodel, and M. Becker, "Analyzing the source code of multiple software variants for reuse potential". 1*8th Working Conference on Reverse Engineering* (WCRE 2011), pp. 303–307, doi: 10.1109/WCRE.2011.44

[16] S. Duszynski, "Analyzing similarity of cloned software variants using hierarchical set models". Dissertation, Technical University of Kaiserslautern, Germany, 2015. [Available online]. Permalink: http://publica.fraunhofer.de/documents/N-332392.html.

[17] V. L. Tenev, "Directed coloured multigraph alignments for variant analysis of software systems". Bachelor Thesis, Fraunhofer IESE Report 112.11/E, Kaiserslautern, Germany, 2011.

[18] V. L. Tenev and S. Duszynski, "Applying bioinformatics in the analysis of software variants". *20th IEEE International Conference on Program Comprehension* (ICPC 2012), 2012, pp. 259–260, doi: 10.1109/ICPC.2012.6240499

[19] V. L. Tenev, S. Duszynski, and M. Becker, "Variant Analysis: set-based similarity visualization for cloned software systems". *21st International Systems and Software Product Line Conference* (SPLC 2017), Vol. B, pp. 22–27, doi: 10.1145/3109729.3109753

[20] J. Businge, M. Openja, S. Nadi, E. Bainomugisha, and T. Berger, "Clone-based variability management in the Android ecosystem". *IEEE International Conference on Software Maintenance and Evolution* (ICSME 2018), doi: 10.1109/icsme.2018.00072

[21] G. Gousios, M. Pinzger, and A. van Deursen, "An exploratory study of the pull-based software development model". *36th International Conference on Software Engineering* (ICSE 2014), doi: 10.1145/2568225.2568260

[22] B. Ray and M. Kim, "A case study of cross-system porting in forked projects". *20th International Symposium on the Foundations of Software Engineering - FSE 12*, 2012, doi: 10.1145/2393596.2393659

[23] S. Stanciulescu, S. Schulze, and A. Wasowski, "Forked and integrated variants in an open-source firmware project". *IEEE International Conference on Software Maintenance and Evolution* (ICSME 2015), doi: 10.1109/icsm.2015.7332461

[24] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. l. Traon, "Automating the extraction of model-based software product lines from model variants". *30th IEEE/ACM International Conference on Automated Software Engineering* (ASE 2015), pp. 396–406, doi: 10.1109/ASE.2015.44

[25] H. Abukwaik, A. Burger, B. K. Andam, and T. Berger, "Semi-automated feature traceability with embedded annotations". *IEEE International Conference on Software Maintenance and Evolution* (ICSME 2018), pp. 529–533, doi: 10.1109/ICSME.2018.00049

[26] J. Rubin, K. Czarnecki, and M. Chechik, "Managing cloned variants". *17th International Software Product Line Conference (SPLC 2013),* 2013, doi: 10.1145/2491627.2491644

[27] T. Pfofe, T. Thüm, S. Schulze, W. Fenske, and I. Schaefer, "Synchronizing software variants with variantsync". *20th International Systems and Software Product Line Conference* (SPLC 2016), 2016, doi: 10.1145/2934466.2962726

[28] S. Zhou, "Improving collaboration efficiency in fork-based development". *34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019)*, 2019, pp. 1218–1221, doi: 10.1109/ASE.2019.00144

[29] W. Fenske, J. Meinicke, S. Schulze, S. Schulze, and G. Saake, "Variant-preserving refactorings for migrating cloned products to a product line". *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER 2017)*, 2017, doi: 10.1109/saner.2017.7884632

[30] A. Shatnawi, A.-D. Seriai, and H. Sahraoui, "Recovering software product line architecture of a family of object-oriented product variants". *Journal of Systems and Software*, vol. 131, pp. 325–346, Sep. 2017, doi: 10.1016/j.jss.2016.07.039

[31] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code". *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, July 2002, doi: 10.1109/tse.2002.1019480

[32] J. R. Cordy, "Exploring large-scale system similarity using incremental clone detection and live scatterplots". *IEEE 19th International Conference on Program Comprehension* (ICPC 2011), 2011, doi: 10.1109/icpc.2011.25

[33] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on Android markets". *36th International Conference on Software Engineering (ICSE 2014)*, 2014, doi: 10.1145/2568225.2568286

[34] A. Hemel and R. Koschke, "Reverse engineering variability in source code using clone detection: a case study for Linux variants of consumer electronic devices". *19th Working Conference on Reverse Engineering (WCRE 2012)*, 2012, doi: 10.1109/wcre.2012.45

[35] D. Moser and H. Menke, "Diffuse – graphical tool for merging and comparing text files". [Online]. Available: http://diffuse.sourceforge.net Last visited: 22 June 2020.

[36] R. Koschke, P. Frenzel, A. P. J. Breu, and K. Angstmann, "Extending the reflexion method for consolidating software variants into product lines". *Software Quality Journal*, vol. 17, no. 4, pp. 331–366, Mar. 2009, doi: 10.1007/s11219-009-9077-8.

[37] Y. Wu, Y. Yang, X. Peng, C. Qiu, and W. Zhao, "Recovering object-oriented framework for software product line reengineering". International Conference on Software Reuse (ICSR 2011), 2011, pp. 119–134, doi: 10.1007/978-3-642-21347-2_10

[38] S. Eick, T. Graves, A. Karr, A. Mockus, and P. Schuster, "Visualizing software changes". *IEEE Transactions on Software Engineering*, vol. 28, no. 4, Apr. 2002, pp. 396-412, doi: 10.1109/TSE.2002.995435

[39] H. Kagdi, M. Collard, and J. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution". *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 19, issue 2, 2007, pp. 77-131, doi: 10.1002/smr.344

[40] C. Hurter, O. Ersoy, and A. Telea, "Smooth bundling of large streaming and sequence graphs". IEEE Pacific Visualization Symposium (PacificVis), 2013, doi: 10.1109/PacificVis.2013.6596126

[41] A. Telea and D. Auber, "Code Flows: visualizing structural evolution of source code". *Computer Graphics Forum*, vol. 27, issue 3, pp. 831-838, 2008, doi: 10.1111/j.1467-8659.2008.01214.x

[42] J. Rubin and M. Chechik, "N-way model merging". 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013), 2013, doi: 10.1145/2491411.2491446

[43] D. Reuling, M. Lochau, and U. Kelter, "From imprecise N-way model matching to precise N-way model merging". *The Journal of Object Technology*, vol. 18, no. 2, p. 8:1, 2019, doi: 10.5381/jot.2019.18.2.a8

[44] A. Schlie, S. Schulze, and I. Schaefer, "Recovering variability information from source code of clone-and-own software systems". *14th International Working Conference on Variability Modelling of Software-Intensive Systems (VAMOS 2020)*, 2020, doi: 10.1145/3377024.3377034

[45] T. Ishio, Y. Sakaguchi, K. Ito, and K. Inoue, "Source file set search for clone-and-own reuse analysis". *IEEE/ACM 14th International Conference on Mining Software Repositories (MSR 2017)*, 2017, doi: 10.1109/msr.2017.19

[46] Y. Sakaguchi, T. Ishio, T. Kanda, and K. Inoue, "Extracting a unified directory tree to compare similar software products". *IEEE 3rd Working Conference on Software Visualization (VISSOFT 2015)*, 2015, doi: 10.1109/vissoft.2015.7332430

[47] A. Lex, N. Gehlenborg, H. Strobelt, R. Vuillemot, and H. Pfister, "UpSet: visualization of intersecting sets". *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 1983–1992, Dec. 2014, doi: 10.1109/tvcg.2014.2346248

[48] J.-B. Lamy and R. Tsopra, "RainBio: proportional visualization of large sets in biology". *IEEE Transactions on Visualization and Computer Graphics*, p. 1, 2019, doi: 10.1109/tvcg.2019.2921544

[49] B. Alsallakh and L. Ren, "PowerSet: a comprehensive visualization of set intersections". *IEEE Transactions on Visualization and Computer Graphics*, vol. 23, no. 1, pp. 361–370, Jan. 2017, doi: 10.1109/tvcg.2016.2598496

[50] B. Alsallakh, L. Micallef, W. Aigner, H. Hauser, S. Miksch, and P. Rodgers, "The state-of-the-art of set visualization". *Computer Graphics Forum*, vol. 35, no. 1, pp. 234–260, Nov. 2015, doi: 10.1111/cgf.12722

[51] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences". *Communications of the ACM*, vol. 18, no. 6, pp. 341–343, Jun. 1975, doi: 10.1145/360825.360861

[52] D. Gusfield, "Efficient methods for multiple sequence alignment with guaranteed error bounds". *Bulletin of Mathematical Biology*, vol. 55, no. 1, pp. 141–154, Jan. 1993, doi: 10.1007/bf02460299

[53] B. B. Bederson, B. Shneiderman, and M. Wattenberg, "Ordered and quantum treemaps". *ACM Transactions on Graphics (TOG)*, vol. 21, no. 4, pp. 833–854, Oct. 2002, doi: 10.1145/571647.571649

[54] J. Venn, "I. On the diagrammatic and mechanical representation of propositions and reasonings". *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 10, no. 59, pp. 1–18, Jul. 1880, doi: 10.1080/14786448008626877

[55] L. L. McQuitty, "Similarity analysis by reciprocal pairs for discrete and continuous data". *Educational and Psychological Measurement*, vol. 26, no. 4, pp. 825–831, Dec. 1966, doi: 10.1177/001316446602600402

[56] L. C. Briand, C. M. Differding, and H. D. Rombach, "Practical guidelines for measurement-based process improvement". *Software Process: Improvement and Practice*, vol. 2, no. 4, pp. 253–280, Dec. 1996, doi: 10.1002/(sici)1099-1670(199612)2:4<253::aid-spip53>3.0.co;2-g

[57] F. Zijlstra, "Efficiency in work behaviour: a design approach for modern tools". Delft: Delft University Press, 1993.

[58] M. Wortschack, "A scalable visualization of set-typed data". Thesis (Diplom), Technische Universität Wien, 2016 [Online]. Available: https://repositum.tuwien.ac.at/urn:nbn:at:at-ubtuw:1-2757

[59] W. F. Tichy, "The string-to-string correction problem with block moves". *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 4, pp. 309–321, Nov. 1984, doi: 10.1145/357401.357404.