Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants

Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon and Alexander Egyed
Johannes Kepler University Linz, Austria
{stefan.fischer, lukas.linsbauer, roberto.lopez, alexander.egyed}@jku.at

***Abstract*—To keep pace with the increasing demand for custom-tailored software systems, companies often apply a practice called clone-and-own, whereby a new variant of a software system is built by coping and adapting existing variants. Instead of a single and configurable system, clone-and-own leads to ad hoc product portfolios of multiple yet similar variants that soon become impossible to maintain effectively. Clone-and-own has widespread industrial use because it requires no major upfront investments and is intuitive, but it lacks a methodology for systematic reuse. In this work we propose $\mathcal{ECCO}$ *(Extraction and Composition for Clone-and-Own)*, a novel approach to enhance clone-and-own that actively supports the development and maintenance of software product variants. A software engineer selects the desired features and $\mathcal{ECCO}$ finds the proper software artifacts to reuse and then provides guidance during the manual completion by hinting which software artifacts may need adaptation. We evaluated our approach on 6 case studies, covering 402 variants having up to 344KLOC, and found that precision and recall of composed products quickly reach a near optimum (>95% reuse).**

I. Introduction

It has become industrial practice to build product variants tailored to individual customers or customer groups. This is evident in many domains including automotive, avionics, manufacturing, or consumer electronics. The product variants such companies produce are often quite similar but they may also exhibit key differences in their features. Naturally, these differences affect the software controlling the products. Yet, many companies do not build software product lines. Instead, we often find that companies build and maintain a software variant for each hardware variant they sell. The reasons are simplicity, maintainability, and certification. In such cases, new software variants then tend to be developed concurrently with new hardware variants using a *clone-and-own* practice whereby the new software variant is derived by merging parts of already existing software variants from earlier hardware [1], [2]. Even though clone-and-own offers a simple and fast alternative to develop new software variants from existing ones, it is highly problematic because it lacks a systematic methodology for reuse, causing severe maintenance problems even when a small number of variants is considered [3].

This paper proposes $\mathcal{ECCO}$ *(Extraction and Composition for Clone-and-Own)*, a novel approach to support software engineers in applying clone-and-own. $\mathcal{ECCO}$ supports the incremental construction of a product portfolio and the reuse of an existing one. It consists of two automated steps and a third tool-supported one[1]. In Step 1, an extraction algorithm compares the existing variants of a portfolio to identify the commonalities and differences, and maps them to their features and feature interactions. In Step 2, a composition algorithm generates new product variants on the basis of the mapping identified in Step 1. $\mathcal{ECCO}$ supports not only the reuse of feature implementations but also relevant feature interactions. In Step 3 a manual but tool-supported completion is performed. This is necessary for features and feature interactions that: *i)* did not exist in any of the previous variants and hence need to be implemented, or *ii)* have always appeared together in all existing variants and hence need to be separated for a new variant in which case $\mathcal{ECCO}$ automatically provides help to direct the software engineer to all these occurrences. Once the completion is done, the next clone-and-own cycle will include the newly composed/completed variant in Step 1. $\mathcal{ECCO}$ is thus an incremental mechanism that can improve in quality with every new variant being constructed.

Our evaluation is three fold. First, we demonstrate on 6 case studies, having between 5 and 256 variants with up to 344 KLOC, that $\mathcal{ECCO}$'s precision and recall of composed products increases quickly with the number of variants in the portfolio, e.g. merely 10 products in the product portfolio already lead to an average precision and recall of higher than 95% for newly composed products. Second, we show that $\mathcal{ECCO}$ can also be used on legacy variants as well as incrementally developed ones. Third, we demonstrate that $\mathcal{ECCO}$'s guidance, for example to separate indistinguishable features, also gets more precise as the number of variants increases which means that the manual completion merely needs to focus on small parts of a newly composed product.

II. Problem Statement and Example

The practice of clone-and-own requires as a starting point a set of existing product variants and for each variant the knowledge of what features it implements. When a software engineer then develops a new product variant, three steps must be performed: 1) *extract* implementation fragments from existing product variants that will be reused in the new variant, 2) *compose* the extracted implementation fragments to form the new variant, and 3) *complete* the new variant, if needed, by adding for example features/interactions that did not yet exist in any existing variant.

In practice, these three steps are exercised in an ad hoc and undisciplined manner but the real problem is that these steps are presently done manually and thus are subject to errors. The manual extraction of relevant artifacts is also a time consuming task that requires detailed knowledge of all product variants. Artifacts can be easily missed or misidentified — leading to

[1]Tool Demo at http://www.isse.jku.at/tools/ecco

extracted fragments with missing or unnecessary implementation. What makes the extraction task specially difficult is the identification of implementation fragments that are responsible for interactions among features. The manual composition requires the merging of all relevant implementation artifacts while remaining faithful to structure and artifact dependencies. This step is difficult because merged implementation fragments are rarely correct or complete. Finally, the completion step has to fill in missing artifacts that could not be found during the extraction (i.e. new features and feature interactions, or perhaps artifacts that were overlooked by software engineers). The completion also has to address the shortcomings of manual extraction and composition. For example, misidentified artifacts need to be eliminated or repaired. All this additional completion work may lead to different implementations of the same functionality which in turn can make the future maintenance of variants a significantly harder endeavor.

To illustrate clone-and-own, consider a set of drawing applications. Each variant supports a subset of the following features: the ability to handle a drawing area (BASE), draw lines (LINE) and rectangles (RECT), select a color to draw with (COLOR) and wipe the drawing area clean (WIPE). Let us assume that three variants $\mathtt{P}_1$, $\mathtt{P}_2$ and $\mathtt{P}_3$ are available already, each providing a distinct set of features, see Table I, and each having its own distinct implementation, see code snippets in Figure 1.

Products	BASE	LINE	RECT	COLOR	WIPE
Product $\mathtt{P}_1$	✓	✓			✓
Product $\mathtt{P}_2$	✓	✓		✓	
Product $\mathtt{P}_3$	✓	✓	✓	✓	

TABLE I: Initial Drawing Application Product Variants

Definition 1: A *Product* $\mathtt{P} \in \mathbb{P}$ is a two-tuple $(\mathtt{Features}, \mathtt{AT})$ where $\mathtt{Features} \subseteq \mathbb{F}$ is the set of features that P provides and AT is an artifact tree, a generic tree structure containing the artifacts that implement the provided features. $\mathbb{P}$ denotes the universe of all products, $\mathbb{F}$ the universe of all features, while P.Features and P.AT denote respectively the Features and AT elements of tuple product P.

Artifacts represent the implementation of variants and can be anything from source code to models, test cases or requirements, etc. The generic tree structure employed is capable of representing their hierarchy and order. For further details refer to Section IV.

What makes clone-and-own problematic is that the behavior of a single feature may depend on the presence or absence of other features. The fact that features influence each other is referred to as *feature interaction* [4] and is a well-known problem in software reuse. To distinguish whether artifacts of a product implement a single feature or a feature interaction we follow a notation and terminology inspired by Liu et. al [5]. A detailed description is presented in [6], here we describe the basic concepts and notations. We distinguish *modules* of two kinds: base modules and derivative modules.

Definition 2: A *base* module labels artifacts that implement a given feature without its interactions. We refer to them with the feature's name written in lowercase.

For example, field List<Line> lines which can be found in Figure 1 Lines 2, 17 and 30 (for $\mathtt{P}_1$, $\mathtt{P}_2$ and $\mathtt{P}_3$ respectively) is code that belongs to the base module of line. This artifact must be present in all products that include feature LINE independent of any other features/interactions.

Definition 3: A *derivative* module $\delta^{\mathtt{n}}(\mathtt{c}_0, \mathtt{c}_1, ..., \mathtt{c}_\mathtt{n}) = \{c_0, c_1, ..., c_n\}$ labels artifacts that implement feature interactions, where c_i is F (if feature F is selected) or $\neg$F (if not selected), and n is the order of the derivative. A derivative module of order n thus represents the interaction of $n+1$ features. We treat derivative modules simply as sets of cardinality $n+1$ containing all features (positive or negative) that are involved in the interaction.

An example of a derivative module is the constructor Line in Figure 1 which is found in all three variants but differs in arguments. Derivative module $\delta^1(\mathtt{line}, \neg\mathtt{color})$ reflects the

Product $\mathtt{P}_1$ (BASE, LINE, WIPE):

```
 1 class Canvas {
 2   List<Line> lines;
 3   void wipe() {
 4     this.lines.clear();
 5   } ...
 6 }
 7 class Line {
 8   Line(Point start) {...} ...
 9 }
10 class Main extends JFrame{
11   initContentPane() {
12     toolPanel.add(lineButton);
13     toolPanel.add(wipeButton);
14   } ...
15 }
```

Product $\mathtt{P}_2$ (BASE, LINE, COLOR):

```
16 class Canvas {
17   List<Line> lines;
18   void setColor(String c) {...} ...
19 }
20 class Line {
21   Line(Color c, Point start) {...} ...
22 }
23 class Main extends JFrame{
24   initContentPane() {
25     toolPanel.add(lineButton);
26     toolPanel.add(colorsPanel);
27   } ...
28 }
```

Product $\mathtt{P}_3$ (BASE, LINE, RECT, COLOR):

```
29 class Canvas {
30   List<Line> lines;
31   List<Rect> rects;
32   void setColor(String c) {...} ...
33 }
34 class Line {
35   Line(Color c, Point start) {...} ...
36 }
37 class Rect {
38   Rect(Color c, int x, int y) {...} ...
39 }
40 class Main extends JFrame{
41   initContentPane() {
42     toolPanel.add(lineButton);
43     toolPanel.add(rectButton);
44     toolPanel.add(colorsPanel);
45   } ...
46 }
```

Fig. 1: Source Code Snippets for Drawing Applications

Product $\mathtt{P}_4$ (BASE, LINE, RECT, WIPE)

```
 1 class Canvas {
 2   List<Line> lines;
 3   List<Rect> rects;
 4   void wipe() {
 5     this.lines.clear();
 6     this.rects.clear(); // added
 7   } ...
 8 }
 9 class Line {
10   Line(Point start) {...} ...
11 }
12 class Rect {
13   Rect(int x, int y) // changed
14     {...} ...
15 }
16 class Main extends JFrame{
17   initContentPane() {
18     toolPanel.add(lineButton);
19     toolPanel.add(rectButton); // 1st
20     toolPanel.add(wipeButton); // 2nd
21   } ...
22 }
```

Fig. 2: Source Code for *Completed* Product 4

situation where feature LINE is selected but feature COLOR is not as in the constructor `Line(Point start)` in Line 8. Notice that here the color argument is not present as compared to Line 21 where both features LINE and COLOR are selected.

Now consider we want to extend the set of drawing applications by creating a new product $\mathtt{P}_4$ with features BASE, LINE, RECT and WIPE, by applying clone-and-own. The goal is to extract as much code from $\mathtt{P}_1$, $\mathtt{P}_2$, and $\mathtt{P}_3$ as possible. For example, a software engineer might start off by copying the entire product $\mathtt{P}_1$ into $\mathtt{P}_4$ because it is a "close fit" and then extract and compose code for feature RECT from product $\mathtt{P}_3$. Doing so is not trivial. For example, we would need to copy the `Rect` class from $\mathtt{P}_3$ to $\mathtt{P}_4$ but change its constructor as it currently has a `Color c` argument (feature COLOR was not selected for $\mathtt{P}_4$). So feature RECT without feature COLOR behaves differently and therefore the extracted code from $\mathtt{P}_3$ contains *surplus code* that the software engineer has to remove. Figure 2 depicts a possible $\mathtt{P}_4$. There are other problems. Since feature RECT has never appeared with feature WIPE before, there is no code that can be associated with module $\delta^1(\mathtt{rect}, \mathtt{wipe})$. Indeed, without this feature interaction the new variant would fail to wipe rectangles. The software engineer would have to add this *missing code* (see Line 6 in Figure 2). Moreover, the software engineer would also need to decide on the order of certain statements. Consider for instance method `initContentPane()` in Line 17 of Figure 2. While it may be clear that the buttons associated for drawing lines and rectangles and for wiping the drawing area clean need to be added to the drawing area, and that the button for drawing lines always goes first, it is not clear in which order the buttons for rectangles and wiping shall appear as the feature interaction among features RECT and WIPE has not been present in any of the initial products. The software engineer then has to decide manually on an order (see for example Lines 19 and 20 in Figure 2). In the following sections we introduce $\mathcal{ECCO}$ and highlight its advantages over manual clone-and-own.

III. ECCO Framework Overview

This section outlines the general work flow of $\mathcal{ECCO}$ and explains how it is useful for dealing with the challenges introduced in Section II. An overview of the proposed framework is shown in Figure 3. $\mathcal{ECCO}$ is an iterative approach that automates two steps of clone-and-own, *Extraction* and *Composition*, and guides software engineers during the *Completion* step by providing hints.

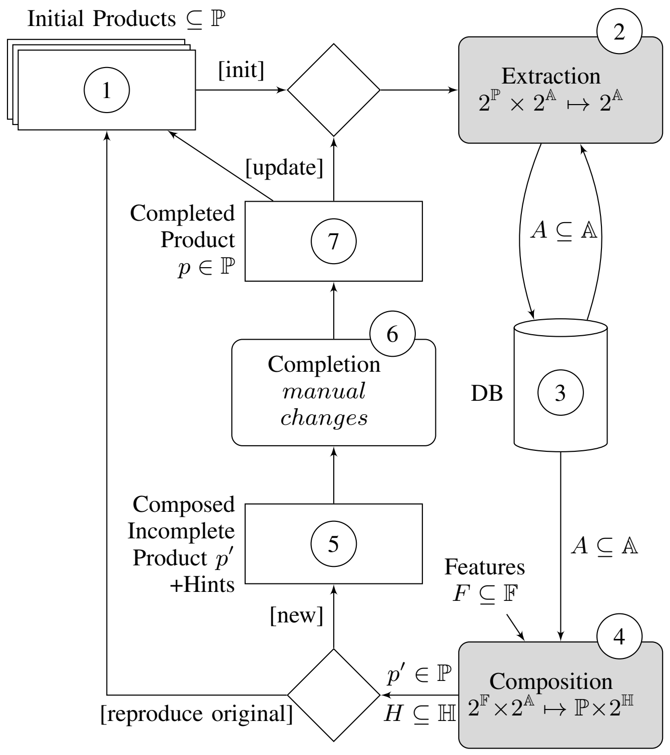

Initial Products $\subseteq \mathbb{P}$
1
[init]
2
Extraction
$2^{\mathbb{P}} \times 2^{\mathbb{A}} \mapsto 2^{\mathbb{A}}$
[update]
$A \subseteq \mathbb{A}$
Completed Product $p \in \mathbb{P}$
7
6
Completion
manual changes
DB
3
Composed Incomplete Product p' +Hints
5
$A \subseteq \mathbb{A}$
Features $F \subseteq \mathbb{F}$
[new]
4
$p' \in \mathbb{P}$
Composition
$2^{\mathbb{F}} \times 2^{\mathbb{A}} \mapsto \mathbb{P} \times 2^{\mathbb{H}}$
[reproduce original]
$H \subseteq \mathbb{H}$

Fig. 3: $\mathcal{ECCO}$ Framework Overview

Using our example from the previous section, the iteration that composes product $\mathtt{P}_4$ starts with the three already known products $\mathtt{P}_1$, $\mathtt{P}_2$, and $\mathtt{P}_3$. These are the initial products used as input (1) for the *Extraction* (2), that is responsible for tracing the implementation artifacts to their features and feature interactions (i.e. base and derivative modules) and stores this information in a database (DB) (3) in the form of *Associations* between artifacts and the modules they trace to. This *Extraction* is automated and replaces the manual extraction illustrated in Section II. It incrementally refines an initially empty set of associations according to new information that becomes available when adding a new input product variant, i.e. given a product P and a set of associations A the *Extraction* produces a refined set of associations A'.

Definition 4: Extraction : $2^{\mathbb{P}} \times 2^{\mathbb{A}} \mapsto 2^{\mathbb{A}}$ where $\mathbb{P}$ denotes the universe of all products and $\mathbb{A}$ denotes the universe of all associations.

The *Composition* (4) uses the associations stored in the database to automatically construct a product that is specified by the set of features selected by the software engineer, referred to as *Composed Incomplete Product* (5). Of course the *Composition* cannot generate artifacts responsible for modules which were not present in any of the products used to build up the database. Also the *Composition* may not be able to uniquely determine the ordering of artifacts that never appeared together, or separate the artifacts of modules that never appeared separately, in any of the input products (these points were discussed above). Nonetheless, $\mathcal{ECCO}$ provides *Hints* at what modules could not be separated and therefore may cause additional surplus artifacts, or for which artifacts there are multiple order options available. These *Hints* can then effectively guide software engineers during the manual *Completion* ⑥ to finalize a *Completed Product* ⑦. Note that this completion is not something incurred by the use of $\mathcal{ECCO}$ but inherent to clone-and-own and thus would also have to be performed even without the use of $\mathcal{ECCO}$, the crucial difference is that $\mathcal{ECCO}$ provides the benefit of hints that guide the software engineers.

$\mathcal{ECCO}$'s *Composition* thus maps a set of desired features F and a set of previously extracted associations A to a product p' and a set of hints H, defined as follows:

Definition 5: Composition : $2^{\mathbb{F}} \times 2^{\mathbb{A}} \mapsto \mathbb{P} \times 2^{\mathbb{H}}$ where $\mathbb{F}$, $\mathbb{A}$, $\mathbb{P}$ and $\mathbb{H}$ denote the universe of features, associations, products and hints respectively.

A completed product, containing all the required implementation (e.g. new module artifacts), can then be used in the next iteration as an additional input product to improve the quality of the composition for future product variants.

The benefits of $\mathcal{ECCO}$ can be summarized as follows. The software engineer:

- does not need to manually find the implementations of all features and feature interactions in existing product variants.
- does not need to merge the different implementations, preserving structure and semantics.
- needs to manually complete the new product though is guided through hints.

IV. ECCO's Realization

This section discusses how the *Extraction*, *Composition* and *Completion* processes work in $\mathcal{ECCO}$.

A. Data Structures and Operations

We refer to the Java code in Figure 1 as *Artifacts*. In fact, artifacts can be anything: text strings, *AST (Abstract Syntax Tree)* nodes from a programming language parser, or Ecore objects from for example UML models. Artifacts can contain references to other artifacts, e.g. in Java a statement calling a method references the called method. Artifacts are organized as *Artifact Trees* that represent the hierarchy and the order of the artifacts. In Java for example a statement will be contained in a method which again will be contained in a class. Knowing about this hierarchy is important for reuse.

Definition 6: An *Artifact Tree* is a tree of artifact nodes with arbitrary depth and structure. An *Artifact Node* is a four-tuple $(\mathtt{SN}, \mathtt{Artifact}, \mathtt{Ordered}, \mathtt{Solid})$. $\mathtt{SN} \in \mathbb{N}$ is the node's sequence number. $\mathtt{Artifact} \in \mathbb{A}$ is an arbitrary artifact. $\mathtt{Ordered} \in \{true, false\}$ determines if the node is ordered. $\mathtt{Solid} \in \{true, false\}$ determines if the node is a solid node. Two nodes n_1 and n_2 from two different artifact trees are equivalent ($n_1 \equiv n_2$) iff their sequence numbers are equal, their artifacts are equal and their parent nodes are respectively equivalent.

A node's sequence number is initially 0 and for children of unordered nodes it remains 0. For ordered nodes the order of their children matters, for example a method in Java whose children are statements whose order of course matters. The sequence number is necessary because children of ordered nodes are no longer uniquely identified by their artifact (for example in Java a method can contain the same statement several times at different positions). A solid node is considered part of the tree, whereas non-solid nodes are just placeholders to keep a path to the root of the tree for solid nodes further down in the tree. Therefore every leaf node in an artifact tree must always be a solid node. Initially in a product every node is solid. For example, Figures 4a and 4b show the artifact trees for the class `Canvas` of products $\mathtt{P}_1$ and $\mathtt{P}_2$ respectively.

To be able to compare and combine artifact trees we define a number of operations on trees that resemble their set counterparts.

Definition 7: An artifact tree AT_1 is a subset of another artifact tree AT_2 ($AT_1 \subseteq AT_2$) iff for every solid node in AT_1 there is an equivalent solid node in AT_2.

Definition 8: An artifact tree AT is the intersection of two other artifact trees AT_1 and AT_2 ($AT = AT_1 \cap AT_2$) iff $AT \subseteq AT_1$ and $AT \subseteq AT_2$ and for every solid node in AT_1 for which there is an equivalent solid node in AT_2 there is also an equivalent solid node in AT.

Definition 9: An artifact tree $AT = AT_1 \setminus AT_2$ iff for every solid node in AT_1 for which there is no equivalent solid node in AT_2 there is an equivalent solid node in AT and $AT \subseteq AT_1$.

Definition 10: An artifact tree AT is the union of two other artifact trees AT_1 and AT_2, denoted with $AT = AT_1 \cup AT_2$, iff $AT_1 \subseteq AT$ and $AT_2 \subseteq AT$ and for every solid node in AT there is an equivalent solid node in AT_1 or AT_2 or in both.

The leaf nodes of an artifact tree are always solid nodes. Non-solid nodes have the sole purpose of providing a way to preserve the structure for solid nodes. This is for example necessary when a node is part of a trace whose parent is not. This is expressed by the child node being solid and the parent node being non-solid. An example of these operations is given in Figure 4 by means of products $\mathtt{P}_1$ and $\mathtt{P}_2$, considering only the artifacts of class `Canvas` for simplicity. Solid nodes are depicted with a solid border and non-solid nodes with a dotted border. Figure 4d shows the result of $\mathtt{P}_1.\mathtt{AT} \setminus \mathtt{P}_2.\mathtt{AT}$. Those artifacts being unique to $\mathtt{P}_1.\mathtt{AT}$ remain solid while the common artifacts do not. The intersection operation is shown in Figure 4f which depicts the result of $\mathtt{P}_1.\mathtt{AT} \cap \mathtt{P}_2.\mathtt{AT}$ containing all the solid artifacts being common to $\mathtt{P}_1.\mathtt{AT}$ and $\mathtt{P}_2.\mathtt{AT}$. Figure 4e shows the union of the two artifact trees in 4d and 4c.

For ordered nodes a trace is more than just the information of whether an artifact is required for the implementation of a module or not. In addition, the ordering of the artifacts must be considered. The implementation of a certain module could for example be reflected in the change of the order of

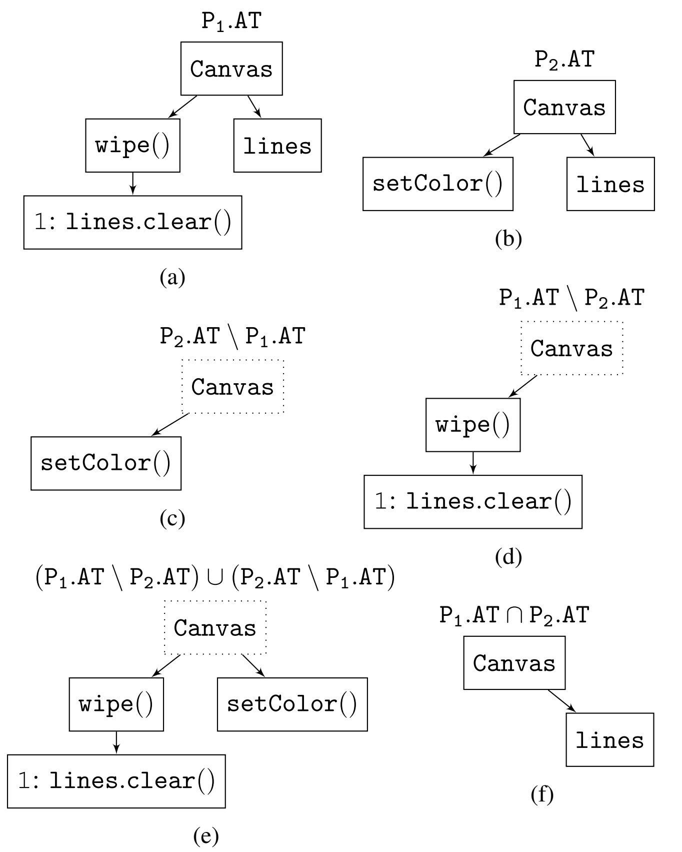


Fig. 4: Minus (c)(d), Union (e) and Intersection (f) operations between Artifact Trees of Product $\mathtt{P}_1$ (a) and Product $\mathtt{P}_2$ (b)

artifacts. Or when merging the artifacts of an ordered node that stem from different traces it is necessary to know in what order they must be merged. Therefore for every set of equivalent ordered nodes a sequence graph is maintained, which is basically a partial order relation describing the order of the nodes' children among all traces. A sequence graph is a directed, acyclic graph with exactly one start node and exactly one end node. Transitions between nodes are labeled with the child artifact nodes of the ordered node the sequence graph belongs to. Every possible path from the start node to the end node describes a possible ordering and contains every child artifact node exactly once. The nodes of the sequence graph themselves do not contain any information.

Prior to the comparison of artifact trees an alignment is performed that aligns every new ordered node's children to the corresponding sequence graph in the database. If no such sequence graph exists then a new one is created. During the alignment every new ordered node's children's sequence numbers are updated in such a way that the corresponding sequence graph (i.e. the respective partial order relation) is not violated (remember: nodes' sequence numbers determine, in addition to the nodes' artifacts, whether two nodes are considered equivalent) and a cost function is minimized. For our purpose we use a cost function that minimizes the number of new unique sequence numbers assigned, i.e. the cost function maximizes the number of matched nodes, meaning as many nodes as possible with an artifact equal to an already existing node's artifact in the sequence graph are assigned the same sequence number if possible without violating the sequence graph (i.e. the underlying partial order relation). After the alignment the sequence graph is updated to reflect the newly learned orderings of artifacts.

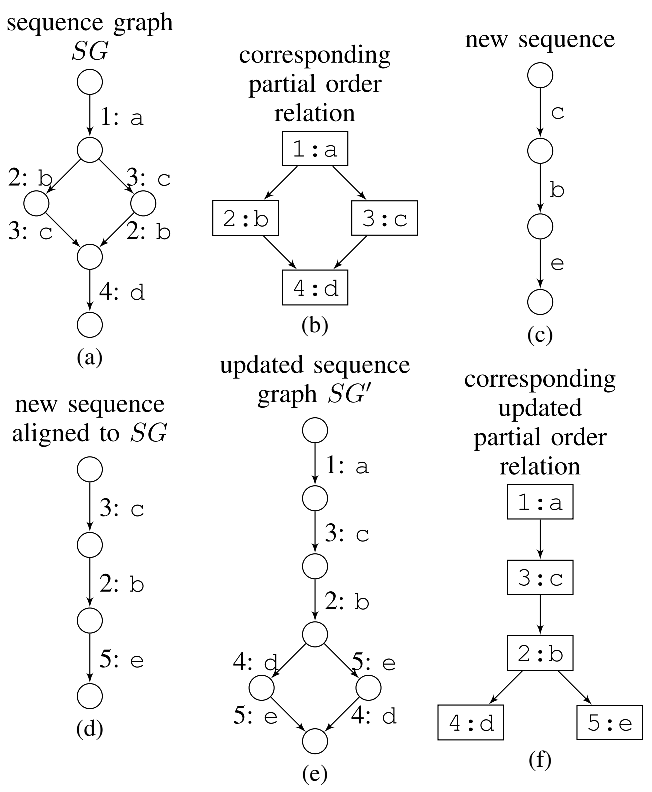


Fig. 5: Sequence Graph Example

An example of such a sequence graph, the alignment of a new artifact sequence and the subsequent update of the sequence graph is shown in Figure 5. Figures 5a and 5b show a sequence graph SG and the corresponding partial order relation respectively. They express that node `[1:a]` (i.e. the node with artifact a and sequence number 1) always goes first and node `[4:d]` always last. The order of `[2:b]` and `[3:c]` in between is not determined. Figure 5c shows the artifact node sequence for the same ordered node of a new product that is being added, initially without sequence numbers assigned. After the alignment of the new sequence in 5c to the sequence graph SG in 5a the artifact nodes in the new sequence have sequence numbers assigned to them as shown in Figure 5d. The sequence numbers 2 and 3 were reused for artifacts b and c respectively as this did not violate SG, and the artifact node with artifact e got a new sequence number 5 assigned. After this alignment the sequence graph SG is updated. It is shown in Figure 5e along with the corresponding updated partial order relation in Figure 5f. The order of nodes `[2:b]` and `[3:c]` is now fixed, however, now it is not certain anymore which node goes last, `[4:d]` or the new node `[5:e]`. This process is repeated whenever a new product is added for every pair of ordered nodes in the database and the new product.

B. Extraction

The *Extraction* has 5 rules. Given two products A and B:

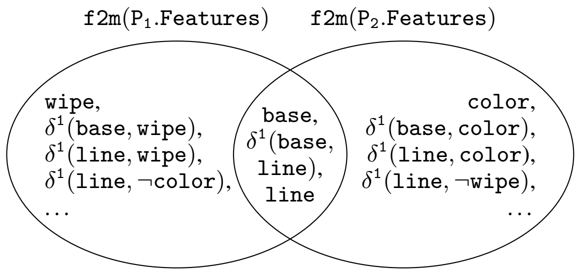
1) Common artifacts *very likely* trace to common modules.

2) Artifacts in A and not B *very likely* trace to modules that are in A and not B, and vice versa.
3) Artifacts in A and not B *cannot* trace to modules that are in B and not A, and vice versa.
4) Artifacts in A and not B can *at most* trace to modules that are in A, and vice versa.
5) Artifacts in A and B can *at most* trace to modules that are in A or B.

These rules require the comparison of module sets as well as different artifact trees. The comparison of the module sets is based on simple set operations. We define a number of auxiliary functions for the work with features and modules. To compute for a set of features F the set $\bar{F}$ of the same features negated: $\bar{F} = negateFeatures(F) = nF(F) = \{\neg f \mid f \in F\}$. To compute the set of modules M from a set of positive (i.e. selected) features F and a set of negative (i.e. not selected) features $\bar{F}$: $M = f2m(F, \bar{F}) = \{p \cup n \mid p \in 2^F \setminus \emptyset \wedge n \in 2^{\bar{F}}\}$. To update a set of modules M with a set of previously unknown features $\bar{F}$: $M' = uM(M, \bar{F}) = \{m \cup n \mid m \in M \wedge n \in 2^{\bar{F}}\}$. Figure 6 shows the comparison of the modules of product $\mathtt{P}_1$ and product $\mathtt{P}_2$. The comparison of artifact trees is performed in a similar fashion using the operators defined in the previous subsection. The extracted information is stored in the form of associations between artifacts and modules that trace to these artifacts.

Definition 11: An *Association* $\mathtt{A} \in \mathbb{A}$ is a 5-tuple $(\mathtt{M}, \mathtt{All}, \mathtt{Max}, \mathtt{Not}, \mathtt{AT})$ where `M` is the set of modules to which the association's artifacts *very likely* trace, `All` is the set of all modules with which the artifacts have ever been associated, `Max` is the set of modules to which the artifacts can *at most* trace and `Not` is the set of modules to which the artifacts *cannot* trace. `AT` is the artifact tree containing the association's artifacts.

The high level pseudo code for the extraction process is shown in Algorithm 1. Lines 3 to 7 do the initialization. Lines 9 to 26 iterate over every association a in A, update its modules with new features (Line 11), perform the alignment and sequencing of matching ordered nodes (Line 13) and then compare it to the new association a_{new} (Lines 15 to 23) by computing an association for the intersection a_{int} and updating the old association a and the new association a_{new} accordingly. Line 25 adds the intersection association a_{int} to the set of associations A_{new} to be returned. Lastly the remainder of the new association a_{new} is added to the set in Line 27 and then returned.



f2m(P1.Features)
f2m(P2.Features)
wipe, δ1(base, wipe), δ1(line, wipe), δ1(line, ¬color), ...
base, δ1(base, line), line
color, δ1(base, color), δ1(line, color), δ1(line, ¬wipe), ...

Fig. 6: Comparison of Modules of Product $\mathtt{P}_1$ and Product $\mathtt{P}_2$

Algorithm 1 Extraction Algorithm

```
1  Input: Product $p$, Set of Associations $A$
2
3  $F_{all}$ = set of all features occuring in any $a \in A$
4  $M$ = $f2m(p.Features, nF(F_{all} \setminus p.Features))$; // modules for $p$
5  $F_{new}$ = $p.Features \setminus F_{all}$; // new features
6  $a_{new}$ = $(M, M, M, \emptyset, p.AT)$; // association for $p$
7  $A_{new}$ = $\emptyset$; // set of new association
8
9  for $a$ in $A$ do
10   // update modules in $a$
11   $a$ = ($uM(a.M, F_{new})$, $uM(a.All, F_{new})$,
         $uM(a.Max, F_{new})$, $uM(a.Not, F_{new})$, a.AT);
12
13   $doAlignmentAndSequencing(a_{new}, a)$;
14
15   // compute intersection
16   $a_{int}$ = ($a.M \cap a_{new}.M$, $a.All \cup a_{new}.All$, $\emptyset$, $\emptyset$,
         $a.AT \cap a_{new}.AT$);
17   $a_{int}.Max$ = $a_{int}.All \setminus a_{int}.Not$;
18   // update existing association $a$
19   $a$ = ($a.M \setminus a_{int}.M$, $a.All$, $\emptyset$, $a.Not \cup a_{new}.All$,
         $a.AT \setminus a_{int}.AT$);
20   $a.Max$ = $a.All \setminus a.Not$;
21   // update new association $a_{new}$
22   $a_{new}$ = ($a_{new}.M \setminus a_{int}.M$, $a_{new}.All$, $\emptyset$,
         $a_{new}.Not \cup a.All$, $a_{new}.AT \setminus a_{int}.AT$);
23   $a_{new}.Max$ = $a_{new}.All \setminus a_{new}.Not$;
24
25   $A_{new}$ = $A_{new} \cup \{a_{int}, a\}$;
26 end for
27 $A_{new}$ = $A_{new} \cup \{a_{new}\}$;
28
29 return $A_{new}$;
```

C. Composition

To create a new product, a software engineer must specify the required set of features. $\mathcal{ECCO}$ then uses the extracted associations in the database to compose the corresponding artifacts along with a set of applicable hints to support the software engineer, see step ④ in Figure 3. Recall that the artifacts of a product are stored using an artifact tree. The artifact trees of the individual products have been split up during the extraction and are now stored across different associations. Hence, the composition procedure needs to first gather all relevant artifact trees and then combine them into a single tree. The relevant artifact trees are those that trace to one of the modules of the new product to be composed. If available the *likely* traces of an association are used, if there are none then the set of modules to which the artifacts can *at most* trace determine whether an association's artifact tree is included in a product.

When composing a new product p_{new} we include all *solid* nodes' artifacts of $p_{new}.AT$, i.e. all artifacts that the modules of p_{new} actually trace to, and their ancestors in $p_{new}.AT$. Thus for instance, if an artifact is a statement inside a method we include the method and the class that contain it. Assume for instance that the artifact tree of Figure 4e is the result of the union of all for p_{new} required associations' artifact trees then not only artifacts `lines.clear()`, `wipe()` and `setColor()` are included in the composed p_{new} but also their ancestor `Canvas`.

Additionally we also consider dependencies between artifacts, for instance when a method needs to be declared before it can be called. If there are unresolved dependencies between artifacts during composition we offer four options from which the software engineer can decide the best way to resolve them:

- **Insert referenced artifact:** Include the referenced artifact into the composition.
- **Insert referenced association:** Include all artifacts of the association *assoc* that contains the referenced artifact into the composition, i.e. $p_{new} = (p_{new.features}, p_{new.AT \cup assoc.AT})$. This may lead to adding artifacts from unselected features.
- **Remove referencing artifacts:** Remove the artifacts that reference non-existent artifacts.
- **Leave the reference unresolved:** Leave the referencing artifacts in the composed product without resolving their references. In the case of source code this will raise syntax-errors in the composed product variant, however this can be useful as it marks places in the source code where manual changes are required.

ECCO is capable of issuing two different types of hints:

- **Surplus artifacts that need to be manually removed.** Let M be the set of modules that should be contained in new product p_{new}, and M' be the set of modules that trace to the artifacts in $p_{new}.AT$. The surplus modules are $surplusModules = M' \setminus M$.
- **Manual reordering of artifacts.** In case there exists more than one valid ordering in the sequence graph of an ordered node of the newly composed product, a hint is provided that contains all the possible orderings and lets the software engineer choose.

The high level pseudo code for the composition is shown in Algorithm 2. Lines 3 to 6 do the initialization. Lines 8 to 16 iterate over every association and decide whether its artifacts should be included in the composed product (Line 10). If included the artifacts are merged into the composed product (Line 12) and the hints are updated (Line 13). Finally the dependencies between the artifacts are resolved (Line 19) and the final composed product is assembled (Line 21) and returned.

Product $\mathtt{P}'_4$ shown in Figure 7 is the result of performing $\mathcal{ECCO}$'s *Extraction* using as input products $\mathtt{P}_1$, $\mathtt{P}_2$ and $\mathtt{P}_3$ from our running example of Section II, followed by *Composition* selecting features `BASE`, `LINE`, `RECT` and `WIPE`. $\mathcal{ECCO}$ provides two hints to help the completion of this product. The first hint is that module δ^1(`rect`, `color`) is contained, even though it should not be, because it could not be separated from the desired base module `rect`, which results in the surplus code in Line 13 of $\mathtt{P}'_4$. The second hint is the two possible orderings inside method `initContentPane()` in Line 17 of $\mathtt{P}'_4$ for the statements in Lines 20 and 21. The software engineer must pick which of these statements appears first during $\mathcal{ECCO}$'s *Completion* step.

Algorithm 2 Composition Algorithm

```
1  In: Set of Features $F$, Set of Associations $A$
2
3  $F_{all}$ = set of all features occuring in any $a \in A$
4  $M$ = $f2m(F, nF(F_{all} \setminus F))$; //compute modules for $F$
5  $AT$ = new empty tree; // initial empty tree
6  $H$ = $\emptyset$; // set of hints
7
8  for $a$ in $A$ do
9    // check if $a$ is needed for the new product
10   if ($a.M \cap M \neq \emptyset$ or ($a.M = \emptyset$ and $a.Max \cap M \neq \emptyset$))
         then
11     // merge
12     $AT$ = $AT \cup a.AT$;
13     // add hints
14     $H$ = $H \cup computeHints(M, a)$;
15   end if
16 end for
17
18 // resolve dependencies
19 $AT_{resolved}$ = $resolveDependencies(AT, A)$;
20
21 $P$ = ($F$, $AT_{resolved}$); // put together product
22 return ($P$, $H$);
```

Product $\mathtt{P}'_4$ (`BASE`, `LINE`, `RECT`, `WIPE`)

```
 1 class Canvas {
 2   List<Line> lines;
 3   List<Rect> rects;
 4   void wipe() {
 5     this.lines.clear();
 6     // missing code
 7   } ...
 8 }
 9 class Line {
10   Line(Point start) {...} ...
11 }
12 class Rect {
13   Rect(Color c, int x, int y) // surplus code
14     {...} ...
15 }
16 class Main extends JFrame{
17   initContentPane() {
18     toolPanel.add(lineButton);
19     // uncertain order
20     toolPanel.add(rectButton);
21     toolPanel.add(wipeButton);
22   } ...
23 }
```

Fig. 7: Source Code for *Composed* Incomplete Product $\mathtt{P}'_4$

D. Completion

During this step the software engineer must manually look at the hints provided by $\mathcal{ECCO}$ and determine whether action is required in order to fix the final product. E.g. to fix the surplus module δ^1(`rect`, `color`) the parameter `color` must be removed from the constructor of class `Rect` in Line 13 of Figure 7. In case of choosing an ordering of artifacts where there were multiple options $\mathcal{ECCO}$ not only provides a hint but also an interactive interface in which it provides the software engineer with all possible orderings and the option to make additional changes and select one. Finally, when executing the product, the software engineer will notice that the behaviour of the feature `WIPE` in combination with feature `RECT` is not correct because these two features have never appeared together before. The code for wiping rectangles, the call to `rects.clear()` as shown in Line 6 of Figure 2, is missing and must be added to method `wipe()` of class `Canvas`.

V. Evaluation

The goal of the evaluation is to assess the quality of the composition, by comparing how close a composed product is to a completed product, and the guidance provided for order and surplus problems.

A. Evaluation Metrics

Measuring how close the composed and completed products are is important because in $\mathcal{ECCO}$ the completion is manual, so the closer they are the less effort is required. We used two standard information retrieval metrics, $precision$ and $recall$, for this purpose. Let p be a completed product and p' a composed product (see $\textcircled{7}$ and $\textcircled{5}$ respectively in Figure 3). We have that:

$$Precision[\%] : \frac{TP}{TP + FP} \quad , \quad Recall[\%] : \frac{TP}{TP + FN}$$

False Positive (FP): are the artifacts in the composed product that are not found in the completed product (i.e. have to be deleted manually during completion). Therefore $FP = count(p'.AT \backslash p.AT)$ where $count(AT)$ is the number of solid nodes in AT. In our running example a FP in P'_4 is in line 13 in Figure 7.

False Negative (FN): are the artifacts in the completed product that are not found in the composed product (i.e. have to be added manually during completion). Hence $FN = count(p.AT \backslash p'.AT)$. In our running example FN are the lines 14 and 6 in Figure 2, that are only in P_4.

True Positive (TP): are the artifacts that were in the composed product and still are in the completed product. Hence $TP = count(p.AT \cap p'.AT)$. In Figure 7 all the code lines are TP except for line 13.

For assessing the guidance provided by $\mathcal{ECCO}$ we employ the next two metrics.

Definition 12: Surplus is the percentage of associations used during the composition of a product p' that $\mathcal{ECCO}$ tagged with surplus hints, denoted with $A_{surplus_{p'}}$, to all the associations at the time of composition, denoted by A.

$$Surplus[\%] : \frac{|A_{surplus_{p'}}|}{|A|}$$

Recall that $\mathcal{ECCO}$ may not distinguish two features/interactions when there are no variants where one is present and the other is not. This is the main reason for the false positives discussed above which requires manual deletion during completion. $\mathcal{ECCO}$ provides surplus hints for all associations that may contain false positives. For $\mathcal{ECCO}$ guidance to be useful the surplus hints should be a small percentage of all associations – hence only few artifacts would need to be manually investigated to identify deletions.

Definition 13: Ordering identifies the percentage of ordered artifact nodes (i.e. statements in a method) where there may be 2 or more orderings that cannot be decided automatically (recall Section IV-C).

$$Orderings[\%] : \frac{ambiguous(p'.AT)}{ordered(p'.AT)}$$

This $Orderings$ metric tells us for how many ordered artifact nodes (e.g. code blocks) a order hint is triggered while composing the product, where $ambiguous(AT)$ computes the number of ordered nodes with no clear unique ordering in AT and $ordered(AT)$ computes the number of all ordered artifact nodes in AT. For $\mathcal{ECCO}$ to be useful, the ordering hints should only affect a small percentage of all ordered artifacts.

B. Evaluation Scheme

As it would have been impractical and expensive to have had large numbers of products manually completed, we applied $\mathcal{ECCO}$ on case studies for which we already had a number of completed products available. The idea was to provide $\mathcal{ECCO}$ with only a randomly selected subset of the completed products available for each case study and then use $\mathcal{ECCO}$ to compose the products that were not given as input. This process provided us with the completed products p and allowed us to compare them against composed products p'. Let us illustrate this process with Figure 8. Let us assume that we have four products: P_1, P_2, P_3, and P_4. As input to $\mathcal{ECCO}$, however, we ignore one of the four products (e.g., P_4) and then compose P'_4 with the same features as P_4 but based on the three input products. The composed P'_4 can then be compared to the original P_4 which we ignored. The key advantage of this process is that it allowed us to assess $\mathcal{ECCO}$'s composition quality at varying percentages of provided input products. How good would P'_4 have been with two input products only?

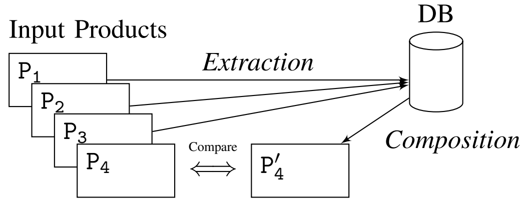


Fig. 8: Evaluation Scheme

For our evaluation we used 6 case studies shown in Table II which had between 5 - 256 completed products. The draw case study is the complete implementation of our running example. ZipMe is a product line for a decomposed version of the Java-Pack-Library. VOD is a product line for video-on-demand streaming applications. GameOfLife are different variants of the popular cellular automaton. ArgoUML is an open source project that has been made into a product line of UML-modelling tools [7], and ModelAnalyzer is a consistency checking and repair technology that has been developed by a team of over a dozen junior and senior researchers [8].

C. Analysis

Figure 9a depicts $\mathcal{ECCO}$'s precision which is rising fast with the number of input products. This is expected because

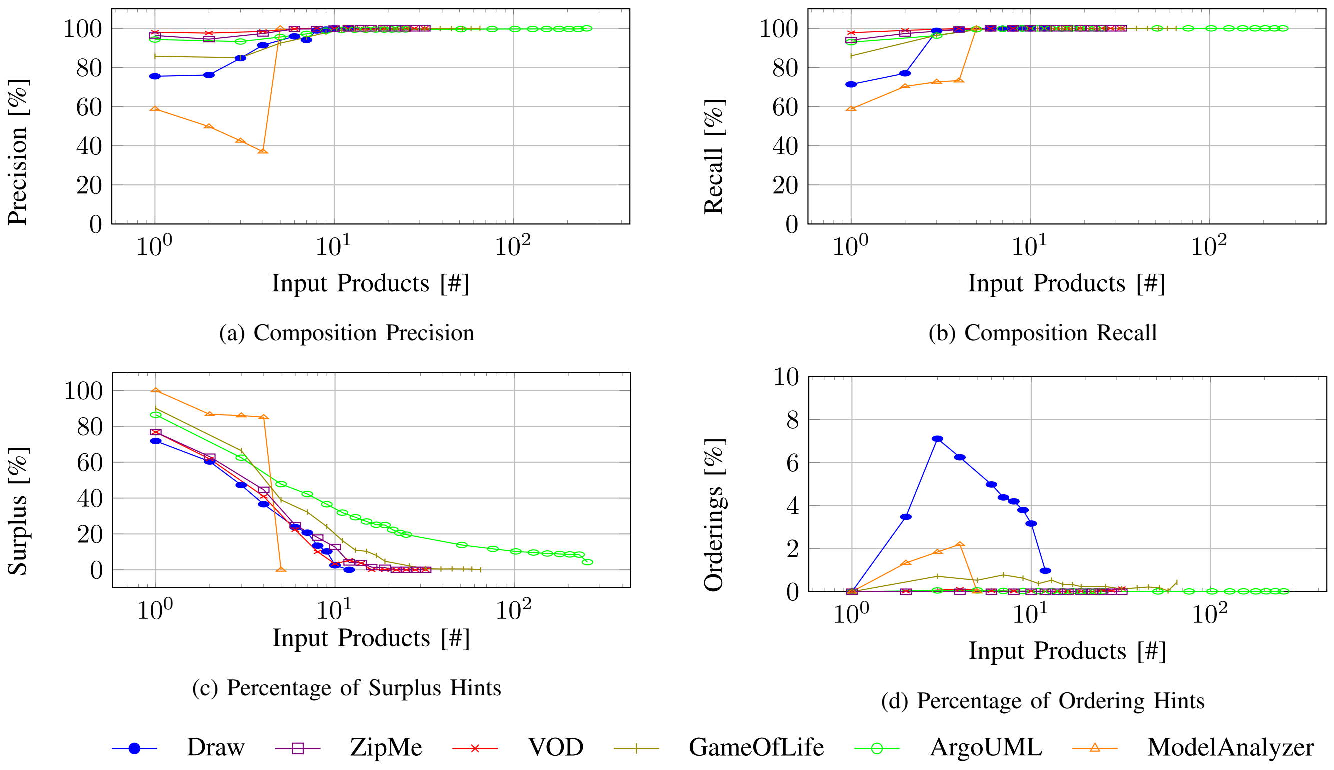


Fig. 9: Metric Plots

Case-Study	#F	#P	LoC	#Art
Draw	5	12	287 - 473	491
ZipMe	7	32	5K - 6.2K	5.2K+
VOD	11	32	4.7K - 5.2K	5.5K+
GameOfLife	15	65	874 - 1.9K	1.3K+
ArgoUML	11	256	264K - 344K	192K+
ModelAnalyzer	13	5	35K - 59K	94K+

#F: Number of Features, #P: Number of Products, LoC: Range of Lines of Code, #Art: Number of Distinct Artifacts

TABLE II: Case Studies Overview

the database gets more precise with every new product. For 5 of the 6 systems, $\mathcal{ECCO}$ becomes near optimal around 10 products, already reaching more than 50% precision after the first few products. This finding suggests that $\mathcal{ECCO}$ is quite useful even when only a few product variants are available as input. The fact that $\mathcal{ECCO}$ becomes near optimal around 10 product variants implies that the manual completion does not require extensive code deletions (FP). The ModelAnalyzer case study deviates because of less overlaps among the products' artifacts that have diverged strongly due to evolutionary changes or bug fixes that have only been applied to some of the products. Despite this divergence the precision is still around 40%.

Like precision, Figure 9b depicts a similarly beneficial recall, again with respect to the number of input products. We found that the recall rises even faster than the precision as more implementation artifacts are added to the database with each product. For the 6 case study systems, $\mathcal{ECCO}$ was able to generate 50% of all code with few variants already, reaching near optimal code generation after 5 product variants. This implies that $\mathcal{ECCO}$ is able to generate much of the needed code and manual completion does not require extensive code additions.

In Figure 9c we can see the percentage of associations which were responsible for triggering surplus hints. This percentage is falling with the number of input products. With more input products the database gets more precise and therefore $\mathcal{ECCO}$ can compose the products with less surplus artifacts. Surplus hints need to be addressed manually during completion but this metric suggests that a strongly decreasing number of associations need to be investigated manually with increasing number of input product variants. Much of the code generated during composition is therefore reliable and manual effort is limited.

Finally, Figure 9d shows the percentage of ordered artifact nodes that had 2 or more ordering choices with respect to the total number of input products. We can see that the number of ordering hints rises at first, since the input products introduce new artifacts respectively have artifacts missing and therefore it is not clear how these have to be ordered. However after a few input products the number of order ambiguities drops, which reduces human effort. Indeed, only very few ordered artifacts are even affected by this. The $\mathcal{ECCO}$ hints are thus quite precise and useful.

Please also note that it only took roughly 2.5 hours for adding all the product variants of our largest case study ArgoUML into the database and around 10 minutes for recomposing all of them. These are reasonable runtimes considering that the generation of the database usually only happens once. The evaluation was performed on an Intel Core i7-4770@3.40GHz Haswell, 16GB of Memory, 64Bit environment.

D. Threats to Validity

While all our case studies are implemented in Java, we believe $\mathcal{ECCO}$ to be applicable to other languages as well.

$\mathcal{ECCO}$, as well as clone-and-own, assumes that the different product variants have major portions of their implementation artifacts in common. This was true for all case studies though least so for the ModelAnalyzer project which is part of the reason of its lower (but still useful) results, since its variants diverged strongly from one another. The other 5 case studies where $\mathcal{ECCO}$ performed the best were product lines, which means their variants were well maintained and did not diverge from one another [9]. However this is also what we would expect when $\mathcal{ECCO}$ is used continuously from the very first product variant on. We thus believe that the 5 product line case studies were representative for continuously maintained clone-and-own systems.

VI. Related Work

Xue et al. use diffing algorithms to identify the common and variable parts of product variants, which are subsequently partitioned using Formal Concept Analysis [10]. To these partitions, Information Retrieval algorithms are applied to identify the code units specific to a feature. In contrast with $\mathcal{ECCO}$, they do not explicitly distinguish the code units that stem from single features from those of feature interactions. However, we will explore how to leverage advanced diffing techniques employed in this work for detecting a wider spectrum of software artifact changes.

Rubin et al. propose a framework for managing product variants that are the result of clone-and-own practices [11]. They outline a series of operators and how they were applied in three industrial case studies. These operators serve to provide a more formal footing to describe the set of processes and activities that were carried out to manage the software variants in the different scenarios encountered in the case studies. We believe that $\mathcal{ECCO}$ can provide the functionality of some of these operators and plan on investigating the applicability of $\mathcal{ECCO}$ to such scenarios.

Koschke et al. aim to reconstruct the module view of product variants and establish a mapping of code entities to architecture entities, with the goal of consolidating software product variants into software product lines by inferring the software product line architecture [1]. For this they adapt the reflection method by applying it incrementally to a set of variants taking advantage of commonalities in their code, for which they use clone detection and function similarity measures. In [12] Rubin et al. survey feature location techniques for mapping features to their implementing software artifacts. The extraction process in our work can also be categorized as a feature location technique, only that we also consider additional problems like feature interactions instead of just single features and also the order of artifacts instead of just their presence or absence. Another feature location survey exists by Dit et al. [13]. Other traceability and information mining algorithms are presented by Ali et al. in [14] or by Kagdi et al. in [15] who use information retrieval techniques in combination with information mined from software repositories to locate features in the source code. Such techniques could help to further improve the results of $\mathcal{ECCO}$.

Nguyen et al. present JSync [16], a tool for managing clones in software systems. Techniques like these could be useful for us when performing the extraction on legacy product variants that have diverged significantly.

VII. Conclusions and Future Work

In this paper we presented $\mathcal{ECCO}$, an approach for supporting the practice of clone-and-own in software engineering. $\mathcal{ECCO}$ leverages the benefits of clone-and-own (e.g. no major upfront investments and reduced time-to-market for the initial products) while still providing the benefits of systematic reuse. The proposed framework was evaluated on 6 diverse case studies of different sizes and domains. Part of our future work will be to improve the results even further by applying clone detection techniques in order to make $\mathcal{ECCO}$ more robust against evolutionary changes and work better with a highly diverged set of product variants, and adding more and diverse case studies with non-code artifacts like models.

References

[1] 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, 2009.

[2] T. Mende, F. Beckwermert, R. Koschke, and G. Meier, "Supporting the grow-and-prune model in software product lines evolution using clone detection," in *CSMR-12*, 2008, pp. 163–172.

[3] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An exploratory study of cloning in industrial software product lines," in *CSMR-17*, 2013.

[4] S. Apel and C. Kästner, "An overview of feature-oriented software development," *Journal of Object Technology*, vol. 8, no. 5, pp. 49–84, 2009.

[5] J. Liu, D. Batory, and C. Lengauer, "Feature oriented refactoring of legacy applications," in *ICSE-28*. ACM, 2006, pp. 112–121.

[6] L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Recovering traceability between features and code in product variants," in *SPLC-7*, 2013, pp. 131–140.

[7] M. V. Couto, M. T. Valente, and E. Figueiredo, "Extracting software product lines: A case study using conditional compilation," in *CSMR*, 2011, pp. 191–200.

[8] A. Egyed, "Automatically detecting and tracking inconsistencies in software design models," *IEEE Trans. Software Eng.*, vol. 37, no. 2, pp. 188–204, 2011.

[9] G. Bavota, A. D. Lucia, A. Marcus, and R. Oliveto, "Using structural and semantic measures to improve software modularization," *Empirical Software Engineering*, vol. 18, no. 5, pp. 901–932, 2013.

[10] Y. Xue, Z. Xing, and S. Jarzabek, "Feature location in a collection of product variants," in *WCRE*. IEEE Computer Society, 2012, pp. 145–154.

[11] J. Rubin, K. Czarnecki, and M. Chechik, "Managing cloned variants: a framework and experience," in *SPLC*, 2013, pp. 101–110.

[12] J. Rubin and M. Chechik, "A survey of feature location techniques," *Domain Engineering: Product Lines, Conceptual Models, and Languages. Springer*, pp. 29–58, 2013.

[13] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[14] N. Ali, Y.-G. Guéhéneuc, and G. Antoniol, "Trustrace: Mining software repositories to improve the accuracy of requirement traceability links," *IEEE Trans. Software Eng.*, vol. 39, no. 5, pp. 725–741, 2013.

[15] H. H. Kagdi, M. Gethers, and D. Poshyvanyk, "Integrating conceptual and logical couplings for change impact analysis in software," *Empirical Software Engineering*, vol. 18, no. 5, pp. 933–969, 2013.

[16] 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.