The ECCO Tool: Extraction and Composition for Clone-and-Own

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*—Software reuse has become mandatory for companies to compete and a wide range of reuse techniques are available today. However, ad hoc practices such as copying existing systems and customizing them to meet customer-specific needs are still pervasive, and are generically called clone-and-own. We have developed a conceptual framework to support this practice named ECCO that stands for Extraction and Composition for Clone-and-Own. In this paper we present our Eclipse-based tool to support this approach. Our tool can automatically locate reusable parts from previously developed products and subsequently compose a new product from a selection of desired features. The tools demonstration video can be found here: http://youtu.be/N6gPekuxU6o**

I. Introduction

Companies often do not build one-of-a-kind software products, but rather develop a portfolio of similar product variants, each tailored to different customer requirements. These products share a high degree of common functionality (i.e. features) but still differ from each other to address different customer needs. *Software Product Lines (SPLs)* address this problem by providing a configurable system from which all the desired product variants can be derived. However, SPLs require a considerable upfront investment and are more complex to evolve than one-off systems. Because of these drawbacks, software companies refrain or delay the adoption of SPLs. Instead, companies resort to an ad-hoc practice of clone-an-own, where a new product variant is created by copying and pasting artifacts from existing variants which are then modified as necessary [1]. Clone-and-own consists of three informal steps:

1) *extraction* that locates and extracts desired reusable artifacts (e.g. code) from existing variants,
2) *composition* where the selected artifacts are pasted/edited to create the new product, and
3) *completion* where the new product variant is adapted to account for needs that did not exist thus far in any previously available variant.

Clone-and-own lacks systematic methodology and tool support. Furthermore the *extraction* and *composition* processes become more complex as new products are added and consequently more error-prone.

In our previous work (see [2], [3]) we presented an approach for supporting and partially automating the clone-and-own practice and allowing for a more systematic reuse. We defined automated *extraction* and *composition* operators that help software engineers reuse artifacts from previously developed product variants in a clone-and-own manner and provide support during the still manual *completion* by pointing software engineers to open issues that still have to be addressed. Our approach has the advantage that software engineers do not have to change their development practice. They can continue to develop single product variants the way they are used to but get automated support in doing so. We demonstrated the basic feasibility of our approach by performing an evaluation on five case studies. We found that less than 20% of the existing product variants as input allowed for the near optimal construction of new product variants (the other 80% of available products, that were reconstructed by reusing existing functionality). This paper presents a tool that supports our previous work by integrating the defined operators and providing a user interface for the whole workflow in the form of an Eclipse-plugin.

II. Technical Overview

The conceptual framework behind ECCO has been already published in [3]. In this section we give an overview of its main ideas. Figure 1 shows that ECCO consists of two automated steps the *extraction* (2) and the *composition* (4), and one manual step the *completion* (6).

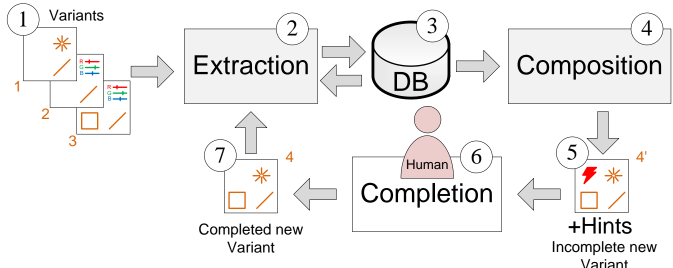

Fig. 1. ECCO Workflow

The starting point of our workflow is the set of product variants (1). For each variant we assume the knowledge of the features it implements and all the artifacts used in its implementation. Artifacts can be anything from lines of source-code (e.g. Java) to fragments of models or other system engineering entities. The product variants are the input to the extraction (2). The result of this step is a database (DB) (3) consisting of traces (i.e. associations between features or features interactions and artifacts), ordering of artifacts and dependencies between traces. Next the *composition* (4) uses the database (3) to generate a new product (5), based on a selection of desired features. New products can be incomplete (e.g. because interactions among features have never occurred before in any previous product variant). Therefore, a manual *completion* (6) has to be performed to finalize the product. The resulting complete product variant (7) can then be fed

back to the *extraction* (2) to refine the information stored in the database (3).

As an illustrative example consider a set of drawing applications with the three variants in Figure 1 already available. These three initial product variants have features for drawing shapes, a feature for coloring them and a feature to wipe the drawing are clean. The first variant shown at the top left corner contains features `LINE` and `WIPE`. The second variant in the middle has features `LINE` and `COLOR` and the third variant has features `LINE`, `RECTANGLE` and `COLOR`. The product variants also have implementation artifacts (not shown in the figure). Next we outline the mechanics of the three steps of ECCO.

A. Extraction

The *extraction* (2) incrementally adds product variants to the database (3), while comparing the new variants with the information already stored in it. By finding commonalities and differences in the features and artifacts of product variants, the *extraction* can map features to corresponding artifacts. The main assumption the *extraction* makes here is, that if two products have features in common, then the artifact they have in common trace to these features. Moreover, the *extraction* is able to deal with feature interactions [2]. Features interactions refer to the fact that the implementation of a feature may change depending on the presence or absence of other features. For instance feature `LINE` is implemented differently if the feature `COLOR` is also present in the product, because it must then allow to define different colors for a line. In addition the *extraction* stores the ordering of artifacts (e.g. the order of statements within a method). Finally ECCO also deals with references among artifacts (e.g. an expression accessing a field has a reference to the field) and is able to extract dependencies between features from these references and from the hierarchy of artifacts.

B. Composition

The *composition* (4) is to a greater extend the reverse operation of the *extraction* (2). It merges extracted fragments together based on selected features. The fragments are selected not only by the features but the *composition* also takes feature interactions into account. Moreover, even references between artifacts and the order of artifacts are considered. The result of the *composition* is a product consisting of the selected features, as far as the *extraction* was able to distinguish them. For features that could not have been extracted from the previously implemented products ECCO also provides some hints to guide the software engineer in the *completion* (6) step. For instance, in Figure 1 the composed product (5) is incomplete, symbolized by the red thunderbolt, because the feature interaction between features `RECTANGLE` and `WIPE` is missing, as these two features have not appeared together in a variant before.

C. Completion

The software engineer can use the hints about missing or surplus features and feature interactions in the manual *completion* (6) of the product. A missing hint shows which features or feature interactions need to be potentially implemented, because they were not present in any of the previous products. A surplus hint tells the software engineer where features and feature interactions could not be separated, and artifacts have to be removed manually. Also the hints for references are important to find artifacts that depend on other artifacts that are missing in the composed product. Moreover, *composition* can compute alternative orderings of artifacts which the user can choose. After a product is completed (7), it can be fed back into the *extraction*, which will refine the database with the gained knowledge.

III. User Experience

The tool is implemented as a plugin to the Eclipse IDE. Therefore software engineers can use a platform with which they can already be familiar. In this section we now describe how our tool supports ECCO.

To start the workflow the user first creates a project and selects the previously developed product variants as input. Complete variants are displayed in the folder "Product-Variants" in the tool, see Figure 2 part 1, in the ECCO navigator. For our illustration we use a set of very simple drawing applications containing just maximum four classes which implement up to five features in a product variant. The tool automatically performs the *extraction* on the variants. It is possible to view the extracted trace information in the tool in the center of the view, see part 2 Figure 2.

Each trace is represented by one of the expanding boxes, where the user can view the features and features interactions, in a simplified form of feature algebra [2], and the artifacts that trace to them. The feature algebra contains feature interactions denoted like $\delta^{n}(\mathtt{c_0})/\delta(\mathtt{c_1})/.../\delta(\mathtt{c_n})$, where c_i is `F` (if feature `F` is selected) or $\neg$`F` (if not selected), and n is the order of the interaction. For instance $\delta^{1}(\mathtt{rectangle})/\delta(\mathtt{wipe})$ represents the feature interaction between features `RECTANGLE` and `WIPE`. A feature interaction of order n thus represents the interaction of $n+1$ features. Hence an order of 0 represents the implementation of a feature regardless of any interactions with any other features (We called this a *base module* in [2], [3]).

For easier navigation through the traces also the contained features are displayed on the top line of each trace. To show the user where the artifacts are located in the tree the tool utilizes an outline view (see Figure 2 part 3) that shows the entire tree containing a selected artifact (e.g. a class in Java). The outline view allows users to find the location of a selected artifact in the hierarchy and to navigate to the trace that contains the artifact.

The next step in the workflow is to compose a new variant. For that purpose the tool offers a view where the user is asked to select the features the new product should have, see Figure 3 part 1. The *composition* produces this product and its hints. First, the tool requests to fix unresolved references. Next, it displays the missing and surplus features and interactions, as shown in Figure 3 part 2. For the surplus hints it is possible to select a feature or a feature interaction and the tool displays in the outline view the code that it traces to. Finally, the user is asked to decide on the order of artifacts, see Figure 3 part 3. The tool displays the possible orderings which the user can

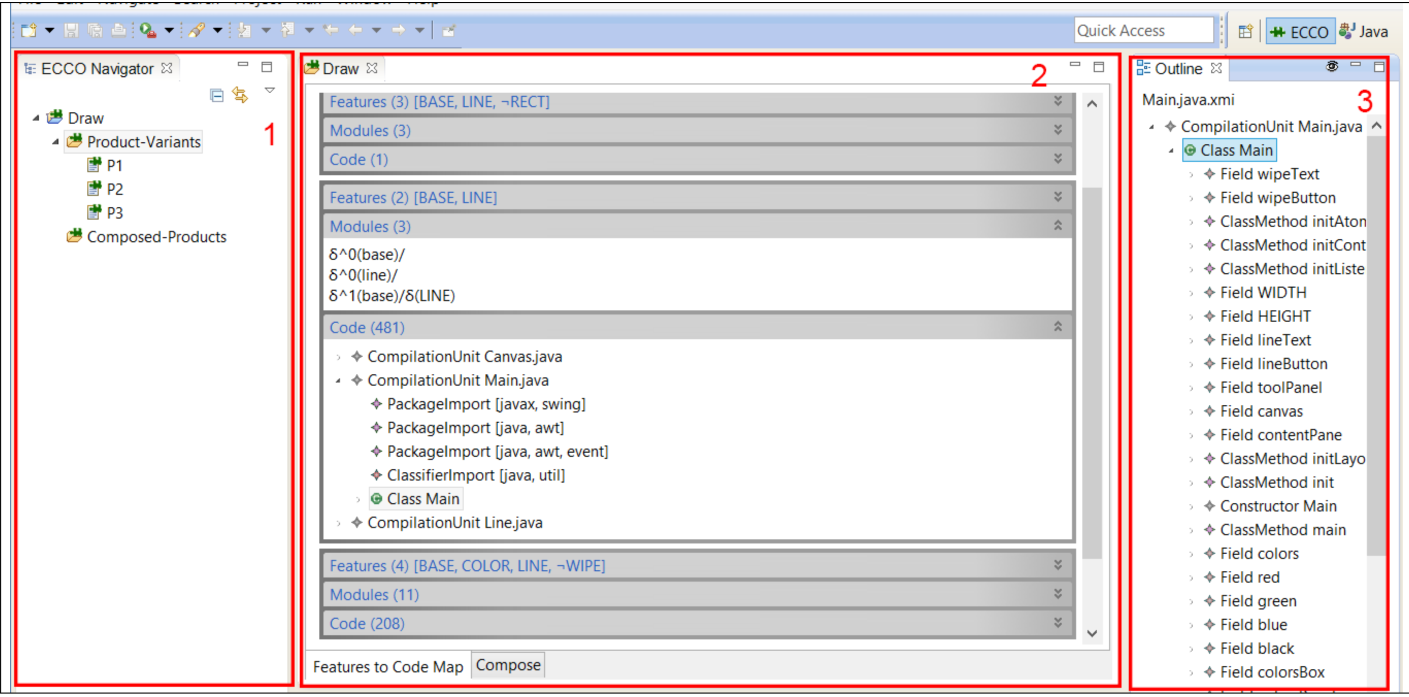

Fig. 2. Tool perspective with ECCO Navigator view, Traces view and Outline view.

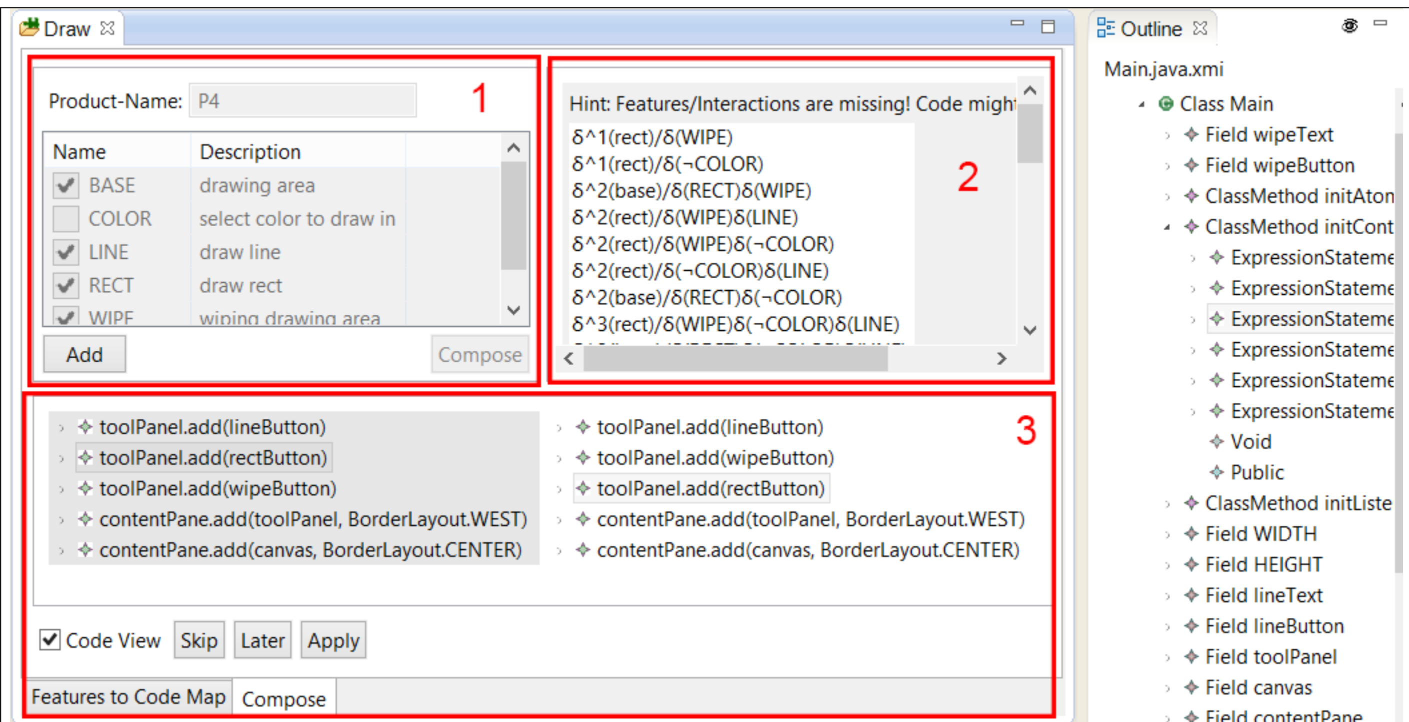

Fig. 3. Composition View with the provided Hints.

choose from. Should none of these orders be the correct one, the user can also choose a custom order by rearranging the artifact via drag and drop.

After all the orderings are decided, the new variant will appear in the ECCO navigator view in the folder ”Composed-Products”. From there, the new variant can be exported to an arbitrary location on the used computer or into a new project. For instance, for Java code, the tool is able to generate a Java project containing the products implementation. Therefore the user can utilize the Java editor provided by Eclipse for the manual *completion* of the product.

The completed product can be used in the tool again and put in the folder ”Product-Variants”. The *extraction* then uses the new product variant to further refine the traces. Therefore,

the results of the *composition* will get better the more products were already implemented.

IV. Evaluation

We evaluated our approach on five case studies, which are SPLs, ranging from 12 to 256 member products with up to 344KLOC code sizes. The evaluation consisted of the extraction of a random subset of the available product variants for each case study as input (a subset of the 12 to 256 variants) and the subsequent automated composition of all its remaining variants, based on the selection of their features. The composed variants were then compared with the original ones (the ones that we did not use as input) to determine how closely the composition matches the original products.

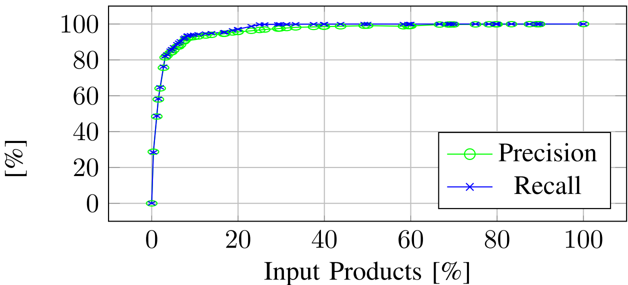

Fig. 4. Average Precision & Recall for Case Studies

In particular, we computed precision and recall of the composed products. Precision reveals if there was surplus code in the composed variants that would have to be deleted manually after using our approach. Complementary, recall indicates if there was any code missing from the composed variants that would have to be added manually. The question we answered was how good precision and recall would get over time. In Figure 4 we see a summary of our findings that show that both precision and recall increase quickly and approach a near optimum with only about 20% of the available products as input. This implies that once 20% of the variants were known, the approach was able to generate the remaining product variants nearly perfectly. However even with just a few products ECCO can provide help for the software engineer in reusing parts of previously developed products. With more variants added to the database the results get even better, since the extracted trace information gets more complete and precise. For a more detailed evaluation of ECCO please refer to the technical paper [3].

V. Related Work

Rubin et al. proposed a *conceptual* framework for managing product variants that are the result of clone-and-own practices [4]. They outlined a series of operators that described the set of processes and activities to manage software variants in the different scenarios they encountered in their case studies. The extraction and composition processes presented in this work could be seen as an *actual implementation* of some of their proposed operators.

Work by Kästner et al. defines *variability mining* as a process that identifies features in legacy code and rewrites them as optional or alternative features (i.e. that can be selected from a group of features) to effectively construct an SPL from a *single* legacy code base [5]. This approach relies on domain expert knowledge or standard feature location tools (as defined by Dit et al. [6]) to find the *seeds* from which to start the search for the complete code fragments that implement a feature. They propose an iterative process where a developer drives the search guided by *variability-aware type checking*, a form of type checking that considers all valid feature combinations of an SPL [7].

The tool FeatureIDE by Thüm et al. is a framework for Feature-Oriented Software Development (FOSD) that provides composition functionality for several FOSD implementation techniques [8].

Another tool by Kästner et al. called CIDE [9] helps to decompose legacy applications into features by color coding features in source code, hiding certain features and supporting the analysis of interactions among features.

VI. Conclusion

We presented ECCO, a tool for supporting the practice of clone-and-own in software engineering. ECCO enables automated systematic reuse of existing arbitrary development artifacts. Our tool is implemented as a plug-in for the Eclipse IDE which is a well known development environment for most software developers. Our evaluation demonstrated the feasibility of our approach on five case studies.

Acknowledgment

This research was funded by the Austrian Science Fund (FWF) projects PP25513-N15 and P25289-N15.

References

[1] 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, pp. 25–34.

[2] L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Recovering traceability between features and code in product variants," in *SPLC 2013, Tokyo, Japan - August 26 - 30, 2013*, T. Kishi, S. Jarzabek, and S. Gnesi, Eds. ACM, 2013, pp. 131–140.

[3] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Enhancing clone-and-own with systematic reuse for developing software variants," in *ICSME-30*, 2014, pp. 391–400.

[4] J. Rubin, K. Czarnecki, and M. Chechik, "Managing cloned variants: a framework and experience," in *SPLC*, 2013.

[5] C. Kästner, A. Dreiling, and K. Ostermann, "Variability mining: Consistent semi-automatic detection of product-line features," *IEEE Trans. Software Eng.*, vol. 40, no. 1, pp. 67–82, 2014.

[6] 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.

[7] C. Kästner, S. Apel, T. Thüm, and G. Saake, "Type checking annotation-based product lines," *ACM Trans. Softw. Eng. Methodol.*, vol. 21, no. 3, p. 14, 2012.

[8] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich, "FeatureIDE: an extensible framework for feature-oriented software development," *Sci. Comput. Program.*, vol. 79, pp. 70–85, 2014.

[9] C. Kästner, "CIDE: decomposing legacy applications into features," in *SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings. Second Volume (Workshops)*. Kindai Kagaku Sha Co. Ltd., Tokyo, Japan, 2007, pp. 149–150.