Merging Cloned Alloy Models with Colorful Refactorings

Chong Liu[12] and Nuno Macedo[13] and Alcino Cunha[12]

[1] INESC TEC, Porto, Portugal
[2] Universidade do Minho, Braga, Portugal
[3] Faculdade de Engenharia, Universidade do Porto, Porto, Portugal

Abstract. Likewise to code, *clone-and-own* is a common way to create variants of a model, to explore the impact of different features while exploring the design of a software system. Previously, we have introduced *Colorful Alloy*, an extension of the popular Alloy language and toolkit to support feature-oriented design, where model elements can be annotated with feature expressions and further highlighted with different colors to ease understanding. In this paper we propose a catalog of refactorings for Colorful Alloy models, and show how they can be used to iteratively merge cloned Alloy models into a single feature-annotated colorful model, where the commonalities and differences between the different clones are easily perceived, and more efficient aggregated analyses can be performed.

Keywords: Feature-oriented design · Refactoring · Alloy · Model merging · Clone-and-own

1 Introduction

Modern software systems are often highly-configurable, effectively encoding a family of software products, or a *software product line* (SPL). *Feature-oriented software development* [3] is the most successful approach proposed to support the development of such systems, organizing software around the key concept of a *feature*, a unit of functionality that implements some requirements and represents a configuration option. Naturally, software design is also affected by such concerns, and several formal specification languages and analyses have been proposed to support *feature-oriented software design* [31,4,8,26]. In particular, this team has proposed Colorful Alloy [26], a lightweight, annotative approach for Alloy and its Analyzer [17], that allows the introduction of fine-grained variability points without sacrificing the language's flexibility. Although different background colors are used to ease the understanding of variability annotations [18], fine-grained extensions still cause maintainability and obfuscation problems.

Refactorings [30,12] – transformations that change the structure of code but preserve its external behavior – could be employed to address some of those problems and generally improve the quality of variability-annotated formal models. However, classical refactoring is not well-suited for feature-oriented development, since both the set of possible variants and the behavior of each variant must be preserved [36], and refactoring laws are typically too coarse-grained to be applied in this context, focusing on constructs such as entire functions or classes.

One of the standard ways to implement multiple variants is through *clone-and-own*. However, as the cost to maintain the clones and synchronize changes in replicas increases, developers may benefit from migrating (by merging) such variants into a single SPL. Fully-automated approaches for clone merging (e.g., [32]) assume a quantifiable measure of quality that is not easy to define when the goal is to merge code, and even less so when the goal is to merge formal abstract specifications. An alternative approach is to rely on refactoring [11], supporting the user in performing stepwise, semi-automated merge transformations.

In this paper we first propose a catalog of variability-aware refactoring laws for Colorful Alloy, covering all model constructs – from structural declarations to axioms and assertions – and granularity levels – from whole paragraphs to formulas and expressions. Then, we show how these refactorings can be used to migrate a set of legacy Alloy clones into a colorful SPL using an approach similar to [11]. Fine-grained refactoring is particularly relevant in this context: design in Alloy is done at high levels of abstraction and variants often introduce precise changes, refactoring only at the paragraph level would lead to unnecessary code replication and a difficulty to identify variability points. The individual refactoring laws and some automatic refactoring strategies, that compose together several merge refactorings, have been implemented in the Colorful Analyzer. We evaluate them by merging back Alloy models projected from previously developed Colorful Alloy SPLs, and by merging several variants of plain Alloy models that are packaged in its official release.

The remainder of this paper is organized as follows. Section 2 presents an overview of Colorful Alloy. Section 3 presents some of the proposed variability-aware refactoring laws, and Section 4 illustrates how they can be used to refactor a collection of cloned models into an SPL. Section 5 describes the implementation of the technique and its preliminary evaluation. Section 6 discusses related work. Finally, Section 7 concludes the paper and discusses some future work.

2 Colorful Alloy

Colorful Alloy [26] is an extension of the popular Alloy [17] specification language and its Analyzer to support feature-oriented software design, where elements of a model can be annotated with feature identifiers – highlighted in the visualizer with different colors to ease understanding – and be analyzed with feature-aware commands. The annotative approach of Colorful Alloy contrasts with compositional approaches to develop feature-oriented languages (either for modeling or for programming), where the elements of each feature are kept separate in different code units (to be composed together before compilation or analysis). We reckon the annotative approach is a better fit for Alloy (and design languages in general), since changes introduced by a feature are often fine-grained (for example, change part of a constraint) and not easily implemented (nor perceivable) with compositional approaches.

```
1  fact FeatureModel {
2    ②①~~some none~~①②    // ② Hierarchical requires ① Categories
3    ③①~~some none~~①③ }  // ③ Multiple requires ① Categories
4
5  sig Product {
6    images: set Image,
7    ①~~catalog: one Catalog~~①,
8    ①③~~category: one Category~~③①,
9    ①③category: some Category③① }
10
11 sig Image {}
12 sig Catalog { thumbnails: set Image }
13 fact Thumbnails {
14   ①~~all c:Catalog | c.thumbnails in (catalog.c).images~~①
15   ①all c:Catalog | c.thumbnails in (category.(②~~inside~~②+②^inside②).c).images① }
16
17 ①②~~sig Category { inside: one Catalog }~~②①
18 ①②sig Category { inside: one Catalog+Category }②①
19 ①②fact Acyclic { all c:Category | c not in c.^inside }②①
20
21 pred Scenario { some Product.images and ①some Category① }
22 run Scenario for 10
23
24 assert AllCataloged { ②all p:Product | some (p.category.^inside & Catalog)② }
25 check AllCataloged with ①,② for 10
```

Fig. 1: E-commerce specification in Colorful Alloy, where background and strike-through colors denote positive and negative annotations, respectively.

Consider as an example the design of multiple variants of an e-commerce platform, adapted from [9], for which a possible encoding in Colorful Alloy is depicted in Fig. 1. The base model (with no extra feature) simply organizes products into catalogs, illustrated with thumbnail images. Like modeling with regular Alloy, a Colorful Alloy model is defined by declaring *signatures* with *fields* inside (of arbitrary arity), which introduce sets of atoms and relations between them. A signature *hierarchy* can be introduced either by extension (**`extends`**) (with parent signatures being optionally marked as **`abstract`**) or inclusion (**`in`**), and simple *multiplicity* constraints (**`some`**, **`lone`** or **`one`**) can be imposed both on signatures and fields. In Fig. 1 the base model declares the signatures `Product` (l. 5), `Image` (l. 11) and `Catalog` (l. 12). Fields `images` (l. 6) and `catalog` (l. 7) associate each product with a *set* of images and exactly *one* catalog, respectively; field `thumbnails` (l. 12) associates each catalog with a set of images.

Additional model elements are organized as *paragraphs*: *facts* impose axioms while *assertions* specify properties to be checked; *predicates* and *functions* are reusable formulas and expressions, respectively. Atomic formulas are either inclusion (**`in`**) or multiplicity (**`no`**, **`some`**, **`lone`** or **`one`**) tests over relational expressions, which can be composed through first-order logic operators, such as universal (**`all`**) and existential (**`some`**) quantifiers and Boolean connectives (such as **`not`**, **`and`**, **`or`** or **`implies`**). Relational expressions combine the declared signatures and fields (and constants such as the empty relation **`none`** or the universe of atoms **`univ`**) with set operators (such as union `+` or intersection `&`) and relational operators (such as join **`.`** or transitive closure `^`). For the base e-commerce model all catalog thumbnails

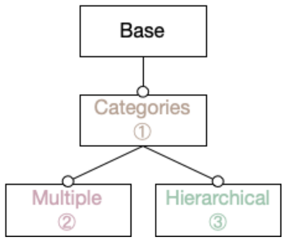


Fig. 2: Feature diagram of the e-commerce specification, where empty bullets denote optional child features.

are assumed to be images of products that appear in that catalog. This is enforced in fact `Thumbnails` (l. 14), where expression `c.thumbnails` retrieves all thumbnails in catalog `c`, `catalog.c` all products in `c`, and `(catalog.c).images` all images of the products in `c`.

This design of the catalog considers 3 optional features: ① allowing products to be classified in categories; ② allowing hierarchical categories; and ③ allowing products to have multiple categories. Not all combinations of these features are valid, as depicted in the feature diagram [3] from Fig. 2: both hierarchical and multiple categories require the existence of categories. In Colorful Alloy certain elements can be annotated with positive ⓒ or negative 🅒 feature delimiters, determining their presence or absence on variants with or without feature c, respectively. Annotations can only be applied to elements of the Alloy AST, either optional elements whose removal does not invalidate the AST – such as declarations and paragraphs – or branches of binary expressions that have a neutral element which can replace the annotated element – such as conjunctions, disjunctions, intersections or unions. Annotations can be nested, which denotes the conjunction of presence conditions. To ease the understanding, and inspired by [18], the Colorful Analyzer employs background colors (for positive annotations) and colored struck-through lines (for negative ones).

In the e-commerce example, feature ① introduces a new signature `Category`, but depending on whether ② is present or not, this signature declares a different field `inside`: without hierarchical categories each category is inside exactly one catalog (l. 17); otherwise, a category can also be inside another category (l. 18). Fields may also be annotated: with categories the `catalog` field of products is removed with a negative annotation ❶ (l. 7) and products are now assigned a category through `category` which, depending on whether ③ is present, assigns exactly one (l. 8) or multiple (l. 9) categories to a product. Hierarchical categories require an additional fact `Acyclic` (l. 19) that forbids categories from containing themselves, either directly or indirectly. Fact `Thumbnails` must be adapted when categories are introduced, so that products are retrieved indirectly from the categories of the catalog. Since one constraint is negatively annotated with ❶ and the other positively with ①, they are actually exclusive. In the latter, depending on the presence of ② either `inside` or its transitive closure `^inside` is used to retrieve all parent categories of products. This finer variability point is introduced by annotating the branches of a union expression; when a presence condition is not met, that branch is replaced by its neutral element, the empty relation. Colorful Alloy does not explicitly support feature models, but the user can restrict valid variants using normal facts. In Fig. 1 fact `FeatureModel` (ll. 1–3) encodes the restrictions from the feature diagram in Fig. 2, forcing ① to be selected whenever ② or ③ are: otherwise formula **`some none`** would be introduced in the model creating an inconsistency.

Like in Alloy, *run* commands can be declared to animate the model under certain properties and *check* commands to verify assertions, both within a specified scope (max size) for signatures. In Colorful Alloy, a scope on features may also be provided, to restrict the variants that should be considered by a command. In Fig. 1 a run command is defined (l. 22) to animate predicate `Scenario` (l. 21): show an instance for any variant (no feature scope is defined) where there are products with images assigned (expression `Product.images` retrieves all images of all products), and, if the variant considers categories, some must also exist. Since no feature scope is imposed, the generated scenario may be for any of the 5 valid variants. To verify the correctness of the design for hierarchical categories, an assertion `AllCataloged` is specified (l. 24) to check whether every product is inside a catalog. The feature scope ①,② of the associated check command (l. 25) restricts analysis to the two variants that have those features selected, those for which `AllCataloged` is relevant.

Some typing rules are imposed on Colorful Alloy models. Roughly, annotations may be nested in an arbitrary order but must not be contradictory, and conditional elements may only be used in compatible annotation contexts. Feature constraints are extracted from simple facts such as `FeatureModel` making these rules more flexible. For instance, `AllCataloged` refers to elements only present in ① variants, but since we known that ② implies ①, that redundant annotation may be omitted from its declaration. This is actually an enhancement of the type system from the original proposal [26]. Another improvement is the support for duplicated signature and field identifiers as long as their annotation context is disjoint. Such is the case of both `Category` declarations. The rule for calling such elements was also relaxed: they may be used in contexts compatible with the union of all the declaration' annotations. For instance, `Category` can be used in any context annotated with ① since one of the two signatures will necessarily exist.

3 Refactoring Laws for Colorful Alloy

Variability-aware refactorings can promote the maintenance of SPLs while preserving the set of variants and their individual behavior. This section proposes a catalog of such refactorings for Colorful Alloy, which complements non variability-aware ones previously proposed for standard Alloy [14,13]. Due to space limitations, only a sample of this catalog is presented.

The refactoring laws for Colorful Alloy are presented in the form of equations between two templates (with square brackets marking optional elements), following the style from [14], under the context of a feature model F extracted from the colorful model under analysis (as described in Section 5). When the preconditions are met and the left or right templates matched, rules can be derived to apply the refactoring in either direction. Throughout the section ⓒ will denote an either positive or negative annotation for c, and **ⓒ** a (possibly empty) sequence of positive or negative annotations. Models are assumed to be type-checked when the rules are applied, so without loss of generality, in an expression **ⓒ**_e_**ⓒ** we assume that the features ***c*** in the closing annotations appear in the reverse order as those in the opening annotations, that there are no contradictory annotations, that only supported elements of the AST are annotated, and that duplicated identifiers have disjoint annotation contexts. We use ***ann*** to refer to any element amenable of being annotated (possibly itself already annotated), ***exp*** for expressions, ***frm*** for formulas, ***n*** for identifiers, ***ds*** for (possibly-annotated) relation declarations, and ***scp*** for scopes on atoms. We assume that the extracted feature model F is encoded as a propositional formula over positive or negative feature annotations [10].

The first set of laws concern the feature annotations themselves, and are often useful to align them in a way that enables more advanced refactorings.

Law 1 (Annotation reordering).

ⓐⓑ*ann*ⓑ**ⓐ**	$=_F$	ⓑ**ⓐ***ann***ⓐ**ⓑ

This basic law originates from the commutativity of conjunction, and allows users to reorganize feature annotations.

Law 2 (Redundant annotations).

ⓐⓑ*ann*ⓑ**ⓐ**	$=_F$	**ⓐ***ann***ⓐ**

provided $F \models$ **ⓐ** $\rightarrow$ ⓑ.

This law relies on the feature model to identify redundant annotations that can be removed or introduced. For instance, if F imposes ② → ① (as in the e-commerce SPL), then whenever a ② annotation is present ① is superfluous, and vice-versa for ❶ and ❷. Note that it can also be used to remove duplicated annotations. Similar laws are defined to manage the feature scopes of commands.

The next set of refactoring laws concerns declarations. The first removes the **abstract** qualifier from signature declarations.

Law 3 (Remove abstract qualifier).

```
ⓐabstract sig n [ext] { ... }ⓐ
ⓐⓑsig n_0 { ... } extends nⓑⓐ
...
ⓐⓒsig n_l { ... } extends nⓒⓐ
```

$=_F$

```
ⓐsig n [ext] { ... }ⓐ
ⓐⓑsig n_0 { ... } extends nⓑⓐ
...
ⓐⓒsig n_l { ... } extends nⓒⓐ
ⓐfact { n=ⓑn_0ⓑ+...+ⓒn_lⓒ }ⓐ
```

provided $l \geq 0$.

Our catalog contains several similar variability-aware laws, some adapted from [13], to remove syntactic sugar from signature and field declarations (e.g., multiplicity annotations) while preserving the behavior in all variants. These laws are used as a preparatory step to enable the following merge refactorings.

Law 4 (Merge top-level signature).

```
ⓐⓑsig n { ds_0,...,ds_k}ⓑⓐ
ⓐ🅑sig n { ds'_0,...,ds'_l}🅑ⓐ
```

$=_F$

```
ⓐsig n {
  ⓑds_0ⓑ,...,ⓑds_kⓑ,
  🅑ds'_0🅑,...,🅑ds'_l🅑 }ⓐ
```

Law 5 (Merge sub-signature).

```
ⓐⓑsig n extends n_0 {
  ds_0,...,ds_k }ⓑⓐ
ⓐ🅑sig n extends n_0 {
  ds'_0,...,ds'_l }🅑ⓐ
ⓒsig n_0 [ext] { ... }ⓒ
```

$=_F$

```
ⓐsig n extends n_0 {
  ⓑds_0ⓑ,...,ⓑds_kⓑ,
  🅑ds'_0🅑,...,🅑ds'_l🅑 }ⓐ
ⓒsig n_0 [ext] { ... }ⓒ
```

provided $F \models$ ⓐ $\rightarrow$ ⓒ.

Signatures cannot be freely merged independently of their annotations, since in Colorful Alloy they are not sufficiently expressive to represent the disjunction of presence conditions. Signatures with the same identifier can be merged if they partition a certain annotation context ⓐ on ⓑ, in which case the latter can be dropped (but pushed down to the respective field declarations). Due to the opposite ⓑ annotations the two signatures never coexist in a variant, and the merged signature will exist in exactly the same variants, those determined by ⓐ. Law 5 considers the merging of the signatures that extend another signature. Here, a precondition guarantees that there are no conflicts after merging: the context ⓒ of the parent signature n_0 must be determined solely by the ⓐ portion of the children annotations (i.e., regardless of ⓑ which will be dropped). Thus, the signature n_0 must be merged beforehand. Notice that these laws act on signatures without qualifiers. If qualifiers were compatible between the two signatures, they can be reintroduced after merging by applying the syntactic sugar laws in the opposite direction.

Returning to the e-commerce example, it could be argued that the declaration of two distinct `Category` signatures under ① depending on whether ② is also selected or not, is not ideal. Since neither signature has other qualifiers, Law 4 can be applied directly from left to right, resulting in the single signature

①sig Category { ❷~~inside: one Catalog~~❷, ②inside: one Catalog+Category② }①

Notice that fields are left unmerged, which are the target of the next law.

Law 6 (Merge binary field).

```
ⓐⓑn: set exp_1ⓑⓐ,
ⓐ🅑n: set exp_2🅑ⓐ
```

$=_F$

```
ⓐn: set ⓑexp_1ⓑ+🅑exp_2🅑ⓐ
```

This law allows binary fields with the same identifier to be merged, even when they have different binding expressions, whenever they partition an annotation context ⓐ. Similar laws are defined for fields of higher arity. Back to the e-commerce example, the duplicated field `inside` introduced by the merging of signature `Category` could be merged into a single field with Law 6, after applying a syntactic refactoring law to move the **`one`** multiplicity annotation to a fact.

```
①sig Category { inside: set ❷~~Catalog~~❷+②Catalog+Category② }①
①❷~~fact{ all this:Category | one this.inside }~~❷①
①②fact{ all this:Category | one this.inside }②①
```

Facts can be soundly merged for whatever feature annotations, since they are all just conjuncted when running a command.

Law 7 (Merge fact).

```
ⓐfact [n] { frm1 }ⓐ
ⓑfact [n] { frm2 }ⓑ
```

$=_F$

```
fact [n] { ⓐfrm1ⓐandⓑfrm2ⓑ }
```

Other elements that can be used in expressions or commands can be merged only when a feature partitions their annotation context. As examples, we show those laws for predicates and assertions.

Law 8 (Merge predicate).

```
ⓐⓑpred n[n0:exp0,...,nk:expk] {
  frm1 }ⓑⓐ
ⓐ🅑pred n[n0:exp'0,...,nk:exp'k] {
  frm2 }🅑ⓐ
```

$=_F$

```
ⓐpred n[
  n0:ⓑexp0ⓑ+🅑exp'0🅑,...,
  nk:ⓑexpkⓑ+🅑exp'k🅑] {
    ⓑfrm1ⓑand🅑frm2🅑 }ⓐ
```

Law 9 (Merge assertion).

```
ⓐⓑassert n { frm1 }ⓑⓐ
ⓐ🅑assert n { frm2 }🅑ⓐ
```

$=_F$

```
ⓐassert n {
  ⓑfrm1ⓑand🅑frm2🅑 }ⓐ
```

Commands are bounded by the feature scope rather than annotated. If two commands act on a partition of the variants, they can be merged into a command addressing their union. As an example, we show a law for merging run commands.

Law 10 (Merge run command).

```
run n scp with ⓐ,ⓑ
run n scp with ⓐ,🅑
ⓒpred n[...] { ... }ⓒ
```

$=_F$

```
run n scp with ⓐ
ⓒpred n[...] { ... }ⓒ
```

provided $F \models$ ⓐ $\rightarrow$ ⓒ.

Likewise Law 5, a precondition guarantees that there are no ambiguities after merging, so that the merged annotation ⓐ completely determines the the predicate to be run. Thus, the respective predicates must be merged beforehand.

Lastly, we provide refactoring laws for formulas and expressions. This distinguishes our approach from other works, addressing finer variability annotations.

Law 11 (Merge common expression).

```
ⓐann op′ ann₁ⓐop🅐ann op′ ann₂🅐   =F   ann op′ (ⓐann₁ⓐop🅐ann₂🅐)
```

where *op* ∈ {`+`, `&`, **`and`**, **`or`**} and *op*′ is *op* or its dual.

This law arises from the distributivity of operators and can be applied to both annotated formulas and expressions. An extreme application is when we have ⓐ*ann*ⓐ*op*🅐*ann*🅐, which can be refactored into *ann*.

Law 12 (Merge left-side inclusion).

```
ⓐexp in exp₁ⓐandⓑexp in exp₂ⓑ   =F   exp in (ⓐexp₁ⓐ&ⓑexp₂ⓑ)
```

Law 13 (Merge right-side inclusion).

```
ⓐexp₁ in expⓐandⓑexp₂ in expⓑ   =F   (ⓐexp₁ⓐ+ⓑexp₂ⓑ) in exp
```

These laws allow the simplification of inclusion tests over the same expression, for whatever annotations, and arise from the properties of intersection and union.

Law 14 (Merge quantification).

```
ⓐⓑqnt n:exp₁ | frm₁ⓑⓐand          =F   ⓐqnt n:ⓑexp₁ⓑ+🅑exp₂🅑 |
  ⓐ🅑qnt n:exp₂ | frm₂🅑ⓐ                  ⓑfrm₁ⓑand🅑frm₂🅑ⓐ
```

where *qnt* ∈ {**`all`**, **`some`**, **`lone`**, **`one`**, **`no`**}.

This law is an example of a simple refactoring for formulas. Our catalog includes laws for other operators, such as composition and multiplicity tests. These, together with Law 11, allow us to merge the two facts that resulted from merging field `inside` into a single fact.

```
①fact{ all this:Category | one this.inside }①
```

We can now apply a syntactic sugar law to move this multiplicity constraint back into the field declaration and remove the fact, which, after an application of Law 11, results in

```
①sig Category { inside: one Catalog+②Category② }①
```

This means that each category is inside exactly one element, which can always be a catalog, or another category if hierarchies are supported. As another example, fact `Thumbnails` can be refactored into

```
fact Thumbnails { all c:Catalog | c.thumbnails in
  (❶~~catalog.c~~❶&①category.(❷~~inside~~❷+②^inside②).c①).images
```

The resulting fact is more compact, but whether it improves model comprehension is in the eyes of the designer.

```
sig Product {
  images: set Image, catalog: one Catalog }
sig Image {}
sig Catalog { thumbnails: set Image }
fact Thumbnails { all c:Catalog |
  c.thumbnails in (catalog.c).images }



pred Scenario {
  some Product.images }
run Scenario for 10
```

Fig. 3: E-commerce base model ❶❷❸.

```
sig Product {
  images: set Image, category: one Category }
sig Image {}
sig Catalog { thumbnails: set Image }
fact Thumbnails { all c:Catalog |
  c.thumbnails in (category.inside.c).images }

sig Category { inside: one Catalog }

pred Scenario {
  some Product.images and some Category }
run Scenario for 10
```

Fig. 4: Clone ①❷❸ introducing categories.

4 Migrating Clones into a Colorful Alloy Model

Approaches to SPL engineering can either be *proactive* – where an *a priori* domain analysis establishes the variability points that guide the development of the product family, *reactive* – where an existing product family is extended as new products and functionalities are developed, or *extractive* – where the family is extracted from existing software products with commonalities [21]. Colorful Alloy was initially conceived with the proactive approach in mind, with annotations being used precisely to extend a base model with the variability points addressing each desired feature. The model in Fig. 1 could be the result of a such a proactive approach to the design of the e-commerce platform.

With plain Alloy, to develop this design we would most likely resort to the clone-and-own approach. First, a base model, such as the one in Fig. 3 would be developed. This model would then be cloned and adapted to specify a new variant adding support for categories, as depicted in Fig. 4. This model would in turn be further cloned and adapted twice to support hierarchical or multiple categories. A final clone would then be developed to combine these two features. Due to space restrictions, these last three clones are not depicted, but they would very likely correspond to something like the projections of the colorful model in Fig. 1 over the respective feature combinations. This section first presents an extractive approach that could be used to migrate all such plain Alloy clone variants into a single Colorful Alloy model. Later we will also show how this technique can be adapted for a reactive scenario, where each new clone variant is migrated into a Colorful Alloy model already combining previous clones.

Our technique follows the idea proposed in [11] for migrating Java code clones into an SPL: first combine all the clones in a trivially correct, but verbose, initial SPL, and then improve it with a step-wise process using a catalog of variant-preserving refactorings. Some of the refactorings used in [11] are similar to those introduced in the previous section (e.g., there is a refactoring for pulling up a class to a common feature that behaves similarly to the merge signature refactoring of Law 4), but in the process they also use several preparatory refactorings to deal with alignment issues: sometimes the name of a method or class is changed in a clone, and in order to apply a merging refactoring the name in the

```
fact FeatureModel { ②❶~~some none~~❶② and ③❶~~some none~~❶③ }

❶❷❸~~sig Product { images: set Image, catalog: one Catalog }~~❸❷❶
...
run Scenario with ❶,❷,❸ for 10
①❷❸~~sig Product { images: set Image, category: one Category }~~❸❷①
...
run Scenario with ①,❷,❸ for 10
①②❸~~sig Product { images: set Image, category: one Category }~~❸②①
...
run Scenario with ①,②,❸ for 10
check AllCataloged with ①,②,❸ for 10
①❷③~~sig Product { images: set Image, category: some Category }~~③❷①
...
run Scenario with ①,❷,③ for 10
①②③sig Product { images: set Image, category: some Category }③②①
...
run Scenario with ①,②,③ for 10
check AllCataloged with ①,②,③ for 10
```

Fig. 5: Part of the initial migrated e-commerce colorful model.

clone should first be made equal to the original one. Although we also require preparatory refactorings (e.g., to remove syntactic sugar from declarations), the name alignment problem is orthogonal to the migration problem, and in this paper we will focus solely on the latter, assuming names in different clones were previously aligned.

To obtain the initial Colorful Alloy model it suffices to migrate every clone to a single model, annotating all paragraphs and commands of each clone with the feature expression that exactly describes the respective variant. For example, for the e-commerce example, the base model of Fig. 3 would be annotated with the feature expression ❶❷❸, since this clone does not specify any of the three features, the clone of Fig. 4 would be annotated with the feature expression ①❷❸, since it specifies the variant implementing only simple categories, and so on. If some feature combinations are invalid (there are only clones for some of the combinations), a fact that prevents the forbidden combinations should also be added, similar to the `FeatureModel` of Fig. 1. For the e-commerce example, part of the initial colorful model with all five variants is depicted in Fig. 5. Notice that, since all of the elements of the different clones are included and annotated with disjoint feature expressions, this Colorful Alloy model trivially and faithfully captures all the variants, although being quite verbose.

After obtaining this initial model, the refactorings presented in the previous section can be repeatedly used in a step-wise fashion to merge common elements, reducing the verbosity (and improving the readability) of the model. For the structural elements the key refactorings are merging signatures (Laws 4 and 5) and fields (Law 6), but, as already explained, some additional preparatory refactorings might be needed to enable those, for example reordering (or removing redundant) feature annotations or removing multiplicity qualifiers.

For example, in the initial model of Fig. 5 we can start by merging signature `Product` (and the respective fields) from clones ❶❷❸ and ①❷❸ and obtain

```
❷❸sig Product {
  images: set Image, ❶catalog: one Catalog❶, ①category: one Category① }❸❷
```

and then merge this with the definition from clone ①②❸ (by first removing the redundant feature annotation ① to enable the application of Law 4 – notice that from the feature model we can infer that ② implies ①) in order to obtain

```
❸sig Product {
  images: set Image, ❶❷catalog: one Catalog❷❶, ①category: one Category① }❸
```

The same result would be obtained if we first merged the declarations of `Product` from clones ①❷❸ and ①②❸, and then the one from clone ❶❷❸ (in this case, to apply Law 4 we would first need to remove the redundant annotation ❷, since from the feature model we can also infer that ❶ implies ❷). By repeatedly merging the variants of `Product` we can eventually get to the ideal (in the sense of having the least duplicate declarations) definition for this signature.

```
sig Product { images: set Image, ❶catalog: one Catalog❶, ①category: set Category① }
①fact { all p:Product | ❸one p.category❸ and ③some p.category③ }①
```

If we repeat this process with all other model elements, we eventually get a (slightly optimized) version of the Colorful Alloy model in Fig. 1. This merging process also has an impact on performance: for instance, the merged command `AllCataloged` with feature scope ①,② and atom scope 10 – which only analyses two variants – takes 13.4s if run in the clones individually, but after the presented merging process the command is checked 1.5x faster at 8.7s in Colorful Alloy.

A similar technique can be used to migrate a new clone into an existing colorful model, thus enabling a reactive approach to SPL engineering. Let us suppose we already have the ideal colorful model for e-commerce, but we decide to introduce a new variant to support multiple catalogs when categories are disabled (a new feature ④). The definition of `Product` for this clone would be

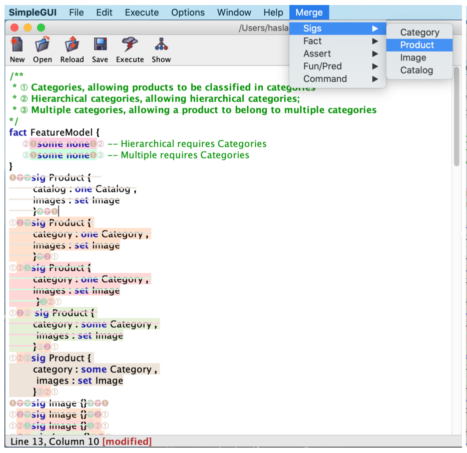
```
sig Product { images: set Image, catalog: some Catalog }
```

To migrate this clone to the existing colorful SPL we would annotate the elements of the new variant with the feature expression that characterizes it, ❶❷❸④, annotate all elements of the existing SPL with ❹ (since it does not support this new feature), refine the feature model to forbid invalid variants (adding **`some none`** annotated with ①④ to forbid the new feature in the presence of categories), and then restart the refactoring process to improve the obtained model.

5 Implementation and Evaluation

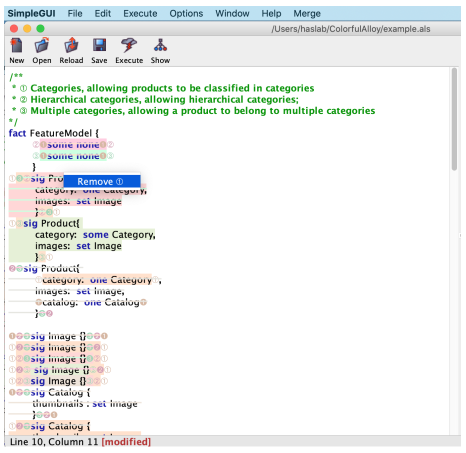
We implemented our catalog of refactorings in the Colorful Alloy Analyzer[4]. Individual refactorings are implemented in a contextual menu, activated by a right-click. The Analyzer automatically detects which refactorings can be applied in a given context. It also scans the model facts to extract feature model constraints

[4] `https://github.com/chongliujlu/ColorfulAlloy/`



```
/**
 * ① Categories, allowing products to be classified in categories
 * ② Hierarchical categories, allowing hierarchical categories;
 * ③ Multiple categories, allowing a product to belong to multiple categories
*/
fact FeatureModel {
    ②❶some none❶② -- Hierarchical requires Categories
    ③❶some none❶③ -- Multiple requires Categories
}
❶②③sig Product {
      catalog : one Catalog ,
      images : set Image
      }③②❶
①❷③sig Product {
      category : one Category ,
      images : set Image
      }③❷①
①②❸sig Product {
      category : one Category ,
      images : set Image
       }❸②①
①❷③ sig Product {
      category : some Category ,
       images : set Image
      }③❷①
①②③sig Product {
      category : some Category ,
       images : set Image
      }③②①
❶②③sig Image {}③②❶
①❷③sig Image {}③❷①
①②❸sig Image {}❸②①
Line 13, Column 10 [modified]
```

Fig. 6: Automatic merge strategies.



```
/Users/haslab/ColorfulAlloy/example.als
/**
 * ① Categories, allowing products to be classified in categories
 * ② Hierarchical categories, allowing hierarchical categories;
 * ③ Multiple categories, allowing a product to belong to multiple categories
*/
fact FeatureModel {
        ②❶some none❶②
        ③❶some none❶③
        }
①❸②sig Pro
        category:  one Category,
        images:  set Image
        }②❸①
①③sig Product{
        category:  some Category,
        images:  set Image
        }③①
❷③sig Product{
        ①category:  one Category①,
        images:  set Image,
        ❶catalog:  one Catalog❶
        }③❷

❶②③sig Image {}③②❶
①❷③sig Image {}③❷①
①②❸sig Image {}❸②①
①❷③ sig Image {}③❷①
①②③sig Image {}③②①
❶②③sig Catalog {
      thumbnails : set Image
      }③②❶
①❷③sig Catalog {
Line 10, Column 11 [modified]
```

Fig. 7: Contextual refactoring menu.

from statements with the shape ⓐ**some none**ⓐ, so that the application of laws with preconditions on feature dependencies (Laws 2, 5 and 10) can be automated. For efficiency reasons, the prototype implements an incomplete decision procedure to check these preconditions, considering only simple implications directly derived from the feature model. This does not affect the soundness of the procedure but may fail to automatically detect some possible rule applications.

To simplify the application of the technique described in the previous section, we also implemented some automatic refactoring strategies to merge signature declarations and other model elements. The strategy to merge signature declarations first applies syntactic sugar refactorings to align the qualifiers of the different declarations. Then, it repeatedly applies Laws 4 and 5 to merge declarations, trying to eliminate features one at a time. Finally, a similar process is applied to merge field declarations with Law 6. To ensure termination, this strategy does not attempt to remove or add redundant features (sometimes it is necessary to add redundant features to enable further merging), being the user responsible for applying that law if necessary (by resorting to the contextual menu). The automatic strategies to merge the remaining paragraphs are similar. The strategy to merge facts essentially attempts to apply laws in the simplification direction (from left to right), until no further law is applicable. Figure 6 shows the menu with the automatic merge refactorings for our running example. If the option to merge signature `Product` is selected we will get the result in Fig. 7, where we still have three declarations for `Product`. To merge these, the user must first use the contextual menu to remove the redundant ① from the first two declarations, through right-clicking in `Product` as shown in Fig. 7. Then, by selecting again the automatic merge signature for `Product` we would get the single `Product` declaration (and additional fact) presented in the previous section.

Our evaluation aimed to answer the following research questions: 1) Since in principle smaller specifications are easier to understand, how effective is the

Table 1: Evaluation results.

SPL	NP	LI	LF	R	DL	RS	US
E-commerce	5	112	31	72.3%	15	101	30
Vending	4	269	94	65.1%	10	140	19
Bestiary	16	140	22	84.3%	7	207	9
RingElection	2	91	52	42.9%	8	25	14
Grandpa	3	99	52	47.5%	11	36	9
AddressBook	3	133	106	20.3%	9	26	18
Hotel	4	324	172	46.9%	9	70	22
Average	5	267	76	54.2%	10	86	17

clone migration technique at reducing the total size of the models? 2) Is our catalog of refactorings sufficient to reach an ideal colorful model specified by an expert? To this purpose we considered various sets of cloned Alloy models that fall in two categories: three examples previously developed by us using a proactive approach with Colorful Alloy (e-commerce, vending machine, and bestiary) and four examples developed by D. Jackson in [17] and packaged with the standard Alloy Analyzer distribution as sample models (ring election, grandpa, address book, and hotel), for which several plain Alloy variants exist (very likely developed with clone-and-own). For the former examples, we generated the plain Alloy clones by projecting the colorful model over all the valid feature combinations.

To answer question 1) we applied our clone migration techniques to all of the examples, until we reached a point where no more merge refactorings could be applied, and compared the size of the resulting Colorful Alloy model with the combined size of all plain Alloy clones (measured in number of lines). The results are presented in Table 1, where NP denotes the number of product clones in the example, LI the initial number of lines, LF the number of lines after migration, R the achieved reduction in lines, DL the number of distinct refactoring laws that were used in the process, RS the number of individual refactoring steps (including those applied during the automatic strategies), and US the number of actions effectively performed by the user (each either a selection of an automatic refactoring strategy or an individual refactoring from a contextual menu). In average we achieved a reduction of around 54% lines, which is quite substantial: the formal design of the full SPL in the final Colorful Alloy model occupies in average half the size of all the plain Alloy clones combined, which in principle considerably simplifies its understanding. The lowest reduction was for the address book example (around 20%), since some of the clones had a completely different approach to specify the system events. The average number of refactoring steps was 86. This number has a strong correlation with the number of clones, since the proposed merging refactorings operate on two clones at a time – if a common element exists in n clones, we will need at least $n - 1$ rule applications to merge it. The average number of steps required by the user was 17, meaning that the proposed technique is quite usable in practice.

For question 2) we relied on the three examples where the clones were derived from previously developed Colorful Alloy models. For all of them, our catalog of refactorings was sufficient to migrate the clones and obtain the original colorful model from which they were derived. As seen in Table 1, these examples required a wider range of refactoring laws than the ones whose variants were developed with clone-and-own in plain Alloy, because the original Colorful Alloy models were purposely complex and diverse in terms of variability annotations, since they were originally developed to illustrate the potential of the Colorful Alloy language.

6 Related Work

Refactoring of SPLs Some work has been proposed on behavior-preserving refactorings for systems with variability, although mostly focusing on compositional approaches [22,36,6,35] (even though some of these could be adapted to the annotative context). Refactorings for an annotative approach are proposed in [23] for C/C++ code with `#ifdef` annotations, which are often used to implicitly encode variability. The AST is enhanced with variability annotations which are considered during variability-aware static analysis to perform transformations that preserve the behavior of all variants. It does not, however, consider the existence of feature models. All these approaches adapt classic refactoring [12] operations, such as renaming or moving functions/fields, while our approach also supports finer-grained refactorings essential to formal software design, including the refactoring of formulas and relational expressions.

Many other refactoring approaches for SPLs have focused only on transforming feature models (e.g., [2]), including some that verify their soundness using Alloy [15,16,39], but without taking into consideration the actual code.

Refactorings have been proposed for formal specification languages such as Alloy [14,13] Object-Z [38,29], OCL-annotated UML [27], Event-B [1] and ASM [37], implementing typical refactorings such as renaming and moving elements, or introducing inheritance. Variability-aware formal specification languages are scarce, and we are not aware of refactorings aimed at them. Our approach relies on the refactorings proposed for normal Alloy [14,13] for the transformations that are not dependent on feature annotations.

Migration into SPLs Since the proactive approach is often infeasible due to the dynamic nature of the software development process, there is extensive work on migrating products into SPLs through extractive approaches, including for clone-and-own scenarios [5]. As detailed in Section 4, the approach presented in this paper can be applied for both the extractive and reactive scenarios, since new variants can be introduced to an already existing Colorful Alloy model.

Nonetheless, only some of this work tackles the migration of multiple variants at the source code level – in contrast to those acting at the domain analysis level, focusing on the feature model. Here, the approach most closely related to ours is the one proposed in [11], which builds on the refactoring operations proposed in [36] to handle the step-wise migration of multiple variants into a single software family. It is has been proposed for feature-oriented programming, a compositional approach, unlike our technique that follows an annotative approach. Again, our refactoring operations are also more fine-grained, while [36] focuses mainly on the refactoring of methods and fields, similarly to our merge signature and fields refactorings. Clone detection is used to semi-automate the process, while our approach is currently manual. In [2] refactorings are also proposed to migrate multiple products into an SPL, but focusing mostly on the feature model level.

Some migration approaches have focused on automating the process, which requires the automatic *comparing*, *matching* and *merging* of artifacts [32,7], including n-way merge [33]. However, such approaches are best-suited to deal with structural models, and not Alloy models rich in declarative constraints. They also assume the existence of quality metrics to guide the process, whose shape would be unclear considering the declarative constraints. Other approaches act on source code of cloned variants to extract variability information [25,34] or high-level architectural models with variability [20,28,19,24] but do not effectively transform the code into an SPL.

Among SPL migration techniques for a single legacy product, it is worth mentioning the one proposed in [40] that converts a product into an annotated colorful SPL using CIDE [18], which was the inspiration for Colorful Alloy [26]. Here, the user must initially mark certain elements as the "seeds" of a feature, and annotations are propagated to related elements automatically.

7 Conclusion and Future Work

In this paper we proposed a catalog of variant-preserving refactoring laws for Colorful Alloy, a language for feature-oriented software design. This catalog covers most aspects of the language, from structural elements, such as signature and field declarations, to formulas in facts and assertions, including analysis commands. Using these refactorings, we proposed a technique for migrating sets of plain Alloy clones, specifying different variants of a system, into a single Colorful Alloy SPL. The technique is step-wise and semi-automated, in the sense that the user is responsible for choosing which elements to merge and selecting occasional preparatory refactorings, being the application of the refactorings automated by the Analyzer. We evaluated the effectiveness of this migration technique with several sets of plain Alloy clones and achieved a substantial reduction in the size of the equivalent Colorful Alloy model, with likely gains in terms of maintainability, understandability, and efficiency of analysis.

In the future we intend to conduct a more extensive evaluation, with more examples and measuring other aspects of model quality (besides number of lines), in order to assess if the positive results achieved in the preliminary evaluation still hold. We also intend to implement a full SAT-based decision procedure for testing the laws preconditions.

Acknowledgments

This work is financed by the ERDF — European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation – COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT – Fundação para a Ciência e a Tecnologia within project PTDC/CCI-INF/29583/2017 – POCI-01-0145-FEDER-029583.

References

1. Abrial, J., Butler, M.J., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: An open toolset for modelling and reasoning in Event-B. Int. J. Softw. Tools Technol. Transf. **12**(6), 447–466 (2010)
2. Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., de Lucena, C.J.P.: Refactoring product lines. In: GPCE. pp. 201–210. ACM (2006)
3. Apel, S., Batory, D.S., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines – Concepts and Implementation. Springer (2013)
4. Apel, S., Scholz, W., Lengauer, C., Kästner, C.: Detecting dependences and interactions in feature-oriented design. In: ISSRE. pp. 161–170. IEEE Computer Society (2010)
5. Assunção, W.K.G., Lopez-Herrejon, R.E., Linsbauer, L., Vergilio, S.R., Egyed, A.: Reengineering legacy applications into software product lines: A systematic mapping. Empirical Software Engineering **22**(6), 2972–3016 (2017)
6. Borba, P., Teixeira, L., Gheyi, R.: A theory of software product line refinement. Theor. Comput. Sci. **455**, 2–30 (2012)
7. Boubakir, M., Chaoui, A.: A pairwise approach for model merging. In: Modelling and Implementation of Complex Systems, pp. 327–340. Springer (2016)
8. Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.: Model checking software product lines with SNIP. Int. J. Softw. Tools Technol. Transf. **14**(5), 589–612 (2012)
9. Czarnecki, K., Pietroszek, K.: Verifying feature-based model templates against well-formedness OCL constraints. In: GPCE. pp. 211–220. ACM (2006)
10. Czarnecki, K., Wasowski, A.: Feature diagrams and logics: There and back again. In: SPLC. pp. 23–34. IEEE (2007)
11. Fenske, W., Meinicke, J., Schulze, S., Schulze, S., Saake, G.: Variant-preserving refactorings for migrating cloned products to a product line. In: SANER. pp. 316–326. IEEE (2017)
12. Fowler, M.: Refactoring – Improving the Design of Existing Code. Addison Wesley object technology series, Addison-Wesley (1999)
13. Gheyi, R.: A Refinement Theory for Alloy. Ph.D. thesis, Universidade Federal de Pernambuco (2007)
14. Gheyi, R., Borba, P.: Refactoring Alloy specifications. Electron. Notes Theor. Comput. Sci. **95**, 227–243 (2004)
15. Gheyi, R., Massoni, T., Borba, P.: A theory for feature models in Alloy. In: Alloy Workshop @ SIGSOFT FSE. pp. 71–80 (2006)
16. Gheyi, R., Massoni, T., Borba, P.: Automatically checking feature model refactorings. J. UCS **17**(5), 684–711 (2011)
17. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2012)

18. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in software product lines. In: ICSE. pp. 311–320. ACM (2008)
19. Klatt, B., Krogmann, K., Seidl, C.: Program dependency analysis for consolidating customized product copies. In: ICSME. pp. 496–500. IEEE (2014)
20. Koschke, R., Frenzel, P., Breu, A.P.J., Angstmann, K.: Extending the reflexion method for consolidating software variants into product lines. Software Quality Journal **17**(4), 331–366 (2009)
21. Krueger, C.W.: Easing the transition to software mass customization. In: PFE. LNCS, vol. 2290, pp. 282–293. Springer (2001)
22. Kuhlemann, M., Batory, D.S., Apel, S.: Refactoring feature modules. In: ICSR. LNCS, vol. 5791, pp. 106–115. Springer (2009)
23. Liebig, J., Janker, A., Garbe, F., Apel, S., Lengauer, C.: Morpheus: Variability-aware refactoring in the wild. In: ICSE (1). pp. 380–391. IEEE (2015)
24. Lima, C., do Carmo Machado, I., de Almeida, E.S., von Flach G. Chavez, C.: Recovering the product line architecture of the Apo-Games. In: SPLC. pp. 289–293. ACM (2018)
25. Linsbauer, L., Lopez-Herrejon, R.E., Egyed, A.: Variability extraction and modeling for product variants. Software and Systems Modeling **16**(4), 1179–1199 (2017)
26. Liu, C., Macedo, N., Cunha, A.: Simplifying the analysis of software design variants with a colorful Alloy. In: SETTA. LNCS, vol. 11951, pp. 38–55. Springer (2019)
27. Markovic, S., Baar, T.: Refactoring OCL annotated UML class diagrams. Software and Systems Modeling **7**(1), 25–47 (2008)
28. Martinez, J., Thurimella, A.K.: Collaboration and source code driven bottom-up product line engineering. In: SPLC (2). pp. 196–200. ACM (2012)
29. McComb, T., Smith, G.: A minimal set of refactoring rules for Object-Z. In: FMOODS. LNCS, vol. 5051, pp. 170–184. Springer (2008)
30. Opdyke, W.F.: Refactoring object-oriented frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign (1992)
31. Plath, M., Ryan, M.: Feature integration using a feature construct. Sci. Comput. Program. **41**(1), 53–84 (2001)
32. Rubin, J., Chechik, M.: Combining related products into product lines. In: FASE. LNCS, vol. 7212, pp. 285–300. Springer (2012)
33. Rubin, J., Chechik, M.: N-way model merging. In: ESEC/SIGSOFT FSE. pp. 301–311. ACM (2013)
34. Schlie, A., Schulze, S., Schaefer, I.: Recovering variability information from source code of clone-and-own software systems. In: VaMoS. pp. 19:1–19:9. ACM (2020)
35. Schulze, S., Richers, O., Schaefer, I.: Refactoring delta-oriented software product lines. In: AOSD. pp. 73–84. ACM (2013)
36. Schulze, S., Thüm, T., Kuhlemann, M., Saake, G.: Variant-preserving refactoring in feature-oriented software product lines. In: VaMoS. pp. 73–81. ACM (2012)
37. Shahir, H.Y., Farahbod, R., Glässer, U.: Refactoring Abstract State Machine models. In: ABZ. LNCS, vol. 7316, pp. 345–348. Springer (2012)
38. Stepney, S., Polack, F., Toyn, I.: Refactoring in maintenance and development of Z specifications. Electron. Notes Theor. Comput. Sci. **70**(3), 50–69 (2002)
39. Tanhaei, M., Habibi, J., Mirian-Hosseinabadi, S.: Automating feature model refactoring: A model transformation approach. Inf. Softw. Technol. **80**, 138–157 (2016)
40. Valente, M.T., Borges, V., Passos, L.T.: A semi-automatic approach for extracting software product lines. IEEE Trans. Software Eng. **38**(4), 737–754 (2012)