Science of Computer Programming 220 (2022) 102829

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Merging cloned Alloy models with colorful refactorings

Chong Liu [a,c], Nuno Macedo [b,a,*], Alcino Cunha [c,a]

[a] INESC TEC, Porto, Portugal
[b] Faculty of Engineering of the University of Porto, Porto, Portugal
[c] University of Minho, Braga, Portugal

A R T I C L E   I N F O

Article history:
Received 1 May 2021
Received in revised form 31 March 2022
Accepted 15 May 2022
Available online 23 May 2022

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

A B S T R A C T

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 refactoring laws 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. We then show how these refactorings can be composed in an automated merging strategy that can be used to migrate Alloy clones into a Colorful Alloy SPL in a single step.

The paper extends a conference version [1] by formalizing the semantics and type system of the improved Colorful Alloy language, allowing the simplification of some rules and the evaluation of their soundness. Additional rules were added to the catalog, and the evaluation extended. The automated merging strategy is also novel.

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* [2] is one of the most popular approaches 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* [3–6]. In particular, this team has proposed Colorful Alloy [6], a lightweight, annotative approach for Alloy and its Analyzer [7], 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 [8], fine-grained extensions still cause maintainability and obfuscation problems.

Refactorings [9,10] – 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

* Corresponding author at: Faculty of Engineering of the University of Porto, Porto, Portugal.
E-mail address: nmacedo@fe.up.pt (N. Macedo).

https://doi.org/10.1016/j.scico.2022.102829

and the behavior of each variant must be preserved [11], 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., [12]) 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 [13], supporting the user in performing stepwise, semi-automated merge transformations.

In this paper we first propose a catalog of variability-aware refactoring laws for an improved, more flexible, version of the Colorful Alloy language [6], covering all model constructs – from structural declarations to axioms and assertions – and granularity levels – from whole paragraphs to formulas and expressions.[1] 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 one previously proposed for Java clones [13], and propose a strategy to automate this process. 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, and refactoring only at the paragraph level would lead to unnecessary code replication and a difficulty to identify variability points. The individual refactoring laws and the automatic merging strategy, that composes together several refactorings in a single step, 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 rest 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. Section 4 illustrates how they can be used to merge a collection of cloned models into an SPL, and presents the automatic merging strategy that can be used to perform such merge in a single step. 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.

This paper extends a conference version [1] by presenting: *i*) the semantics and type system of the improved Colorful Alloy language, which were informally presented; *ii*) additional refactoring rules not previously presented and simplified versions of some of the previously presented rules enabled by the clarification of the language semantics[2]; *iii*) a partial proof of the soundness of the refactoring rules, enabled by the formalization of the improved type system and semantics; *iv*) an improved automated merging strategy that can now be used to migrate Alloy clones into a Colorful Alloy SPL in a single step; *v*) an extended evaluation with four additional examples.

2. Colorful Alloy

2.1. A primer on Colorful Alloy

Colorful Alloy [6] is an extension of the popular Alloy [7] 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.

Consider as an example the design of multiple variants of an e-commerce platform, adapted from the literature [16], 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, respectively. 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. 15) 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 re-usable 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 signatures and fields (and constants such as the empty set **`none`** or the universe

[1] Literature [14,15] often defines as laws fundamental transformations which are composed into higher-granularity refactoring transformations. In this paper, however, we are interested in fine-grained refactorings, so the two terms often overlap.

[2] Laws 3, 6, 8, 10, 12, 14, 16, 17, 18, 20, 23, 24, 26 and 27, and Law 5, respectively.

```
1   fact FeatureModel {
2     ②❶~~some none~~❶②  // ② Hierarchical requires ① Categories
3     ③❶~~some none~~❶③  // ③ Multiple requires ① Categories
4   }
5   sig Product {
6     images: set Image,
7     ❶~~catalog: one Catalog~~❶,
8     ①❸~~category: one Category~~❸①,
9     ①③category: some Category③①
10  }
11  sig Image {}
12  sig Catalog {
13    thumbnails: set Image
14  }
15  fact Thumbnails {
16    ❶~~all c:Catalog | c.thumbnails in (catalog.c).images~~❶
17    ①all c:Catalog | c.thumbnails in (category.(❷~~inside~~❷ + ②^inside②).c).images①
18  }
19  ①❷~~sig Category {~~
20    ~~inside: one Catalog~~
21  ~~}~~❷①
22  ①②sig Category {
23    inside: one Catalog + Category
24  }②①
25  ①②fact Acyclic {
26    all c:Category | c not in c.^inside
27  }②①
28
29  pred Scenario {
30    some Product.images and ①some Category①
31  }
32  run Scenario for 10
33
34  assert AllCataloged {
35    ②all p:Product | some (p.category.^inside & Catalog)②
36  }
37  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.

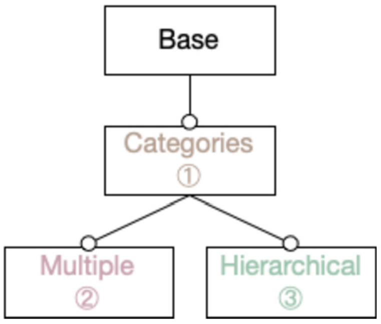


Fig. 2. Feature diagram of the e-commerce specification, where empty bullets denote optional child features.

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 are assumed to be images of products that appear in that catalog. This is enforced in fact `Thumbnails` (l. 15), 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 the assignment of multiple categories to products. Not all combinations of these features are valid, as depicted in the feature diagram [2] from Fig. 2: both hierarchical and multiple categories require the existence of categories in the catalog structure. In Colorful Alloy certain elements can be annotated with positive ⓒ or negative 🅒 feature delimiters, determining their presence or absence in 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 global declarations and paragraphs – or branches of binary expressions that have a neutral element – conjunctions, disjunctions, intersections or unions – which can replace the annotated element when the annotation is not satisfied in a particular variant. Annotations can be nested, which denotes the conjunction of presence conditions. To ease the understanding, and inspired by existing studies [8], the Colorful Analyzer employs background colors (for positive annotations) and colored struck-through lines (for negative ones) in its editor, mixing the colors when annotations are nested.

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. 20); otherwise, a category can also be inside another category (l. 23). Fields may also be annotated: with categories the `catalog` field of products is removed with a negative annotation ❶ (l. 8) 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. 25) 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 interpreted as 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` (l. 1) 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. Alloy models are self-contained, containing both the model, and the commands to be analyzed, so specifying the feature model inside the colorful model is aligned with the Alloy practice.

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. 32) to animate predicate `Scenario` (l. 29): 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. 34) to check whether every product is inside a catalog. The feature scope ①,② of the associated check command (l. 37) 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. Duplicated signature and field identifiers are only allowed if their annotation context is disjoint. Such is the case of both `Category` declarations. Such annotated elements can be called in more relaxed annotation contexts: they may be used in contexts compatible with the union of all the declarations' annotations. For instance, `Category` can be used in any context annotated with ① since one of the two signatures will necessarily exist, as either ② or ❷ will hold. Feature constraints are extracted from simple facts such as `FeatureModel` making these rules more flexible. For instance, `AllCataloged` refers to elements only present in variants with feature ① present, but since we know that ② implies ①, that redundant annotation may be omitted from its specification. This flexibility to allow several declarations for the same signature or field was one of major the improvements to the Colorful Alloy language implemented in the context of this work. In the original proposal of the language [6] only one declaration per signature or field was allowed. The next section presents the syntax, semantics, and type system of this improved version of Colorful Alloy.

2.2. Language syntax, semantics, and type system

The full syntax of the Colorful Alloy language is presented in Fig. 3, highlighting changes with regard to the regular Alloy language. Through the paper, symbol $\textcircled{c}$ denotes either ⓒ or 🅒 for a feature c, and $\neg\textcircled{c}$ converts between the positive and negative version. All paragraphs can be annotated except commands, which are assigned a feature scope to control the analysis procedures. Currently, only non-annotated modules can be imported, such as the libraries packaged with the standard Analyzer. Conjunctions, disjunctions, intersections and unions can have their branches annotated, as well as expressions inside a formula block.

Additional type rules are imposed over models conforming to that syntax, focusing on the arity of expressions (inherited from standard Alloy) and on the annotation contexts (novel for Colorful Alloy). The context of a type rule is a mapping Γ from identifiers to the color annotation (a set of positive and negative feature marks) and arity of their declaration, and a color annotation $\boldsymbol{c}$ under which the rule is being evaluated. Since the same entity can be declared multiple times, as long as their color annotations are disjoint, Γ is actually a relation that associates declared identifiers with a set of pairs with color annotations and arities. A singleton mapping for an identifier n, color annotation $\boldsymbol{c}$, and arity k is denoted by $n \mapsto (\boldsymbol{c}, k)$. The union of mappings can be done with $\cup$, and $+\!\!+$ denotes overriding. This context can be collected from a colorful model using function decls, defined in Fig. 4. For simplicity, this definition considers only block commands, omitting those that call predicates or assertions. Also, function arity used here is an oversimplification, since calculating the arity of an expression requires prior knowledge of the arity of other declared signatures and fields. Throughout the paper, possibly subscripted p denotes a paragraph (or, abusing notation, a module or import statement), *ann* any element amenable of being annotated,

```
spec        ::= module qualName [ [ name,+ ] ] import* paragraph*
import      ::= ⓒ open qualName [ [ qualName,+ ] ] ⓒ
paragraph   ::= colPara | cmdDecl
colPara     ::= ⓒ colPara ⓒ | sigDecl | factDecl | funDecl | predDecl | assertDecl
sigDecl     ::= [ abstract ] [ mult ] sig name,+ [ sigExt ] { colDecl,* } [ block ]
sigExt      ::= extends qualName | in qualName [ + qualName ]*
mult        ::= lone | some | one
decl        ::= [ disj ] name,+ : [ disj ] expr
colDecl     ::= ⓒ colDecl ⓒ | decl
factDecl    ::= fact [ name ] block
assertDecl  ::= assert [ name ] block
funDecl     ::= fun name [ [ decl,* ] ] : expr block
predDecl    ::= pred name [ [ decl,* ] ] block
expr        ::= const | qualName | @name | this | unOp expr | expr binOp expr
              | colExpr colBinOp colExpr | expr arrowOp expr | expr [ expr,* ]
              | expr [ ! | not ] compareOp expr | expr ( ⇒ | implies ) expr else expr
              | quant decl,+ blockOrBar | ( expr ) | block | { decl,+ blockOrBar }
colExpr     ::= ⓒ colExpr ⓒ | expr
const       ::= none | univ | iden
unOp        ::= ! | not | no | mult | set | ~ | * | ^
binOp       ::= ⇔ | iff | ⇒ | implies | − | ++ | <: | :> | .
colBinOp    ::= || | or | && | and | + | &
arrowOp     ::= [ mult | set ] → [ mult | set ]
compareOp   ::= in | =
letDecl     ::= name = expr
block       ::= { colExpr* }
blockOrBar  ::= block | | expr
quant       ::= all | no | mult
cmdDecl     ::= [ check | run ] [ qualName ] ( qualName | block ) [ colScope ] [ typeScopes ]
typeScopes  ::= for number [ but typeScope,+ ] | for typeScope,+
typeScope   ::= [ exactly ] number qualName
colScope    ::= with [ exactly ] [ ⊗ | ⓒ ],+
qualName    ::= [ this/ ] ( name/ )* name
```

Fig. 3. Concrete syntax of the Colorful Alloy language (additions w.r.t. the Alloy syntax are colored red).

$$\mathsf{decls}(\boldsymbol{c}, p_1, \dots, p_i) = \mathsf{decls}(\boldsymbol{c}, p_1) \cup \dots \cup \mathsf{decls}(\boldsymbol{c}, p_i)$$

$$\mathsf{decls}(\boldsymbol{c}, \textcircled{c}\; p\; \textcircled{c}) = \mathsf{decls}(\boldsymbol{c} \cup \{\textcircled{c}\}, p)$$

$$\mathsf{decls}(\boldsymbol{c}, \texttt{module}\ n\ [\ n_1, \dots, n_k\ ]) = n_1 \mapsto (\emptyset, 1) \cup \dots \cup n_k \mapsto (\emptyset, 1)$$

$$\mathsf{decls}(\boldsymbol{c}, \texttt{open}\ n\ [\ n_1, \dots, n_k\ ]) = \mathsf{decls}(\boldsymbol{c}, p_1, \dots, p_i), \text{ where } p_1, \dots, p_i \text{ are the paragraphs of } n$$

$$\mathsf{decls}(\boldsymbol{c}, [\texttt{abstract}]\ [m]\ \texttt{sig}\ n_1\ [\texttt{extends}\ n_2]\ \{\ ds_1, \dots, ds_i\ \}\ [\{\ frm\ \}]) =$$
$$n_1 \mapsto (\boldsymbol{c}, 1) \cup \mathsf{decls}(\boldsymbol{c}, ds_1) \cup \dots \cup \mathsf{decls}(\boldsymbol{c}, ds_i)$$

$$\mathsf{decls}(\boldsymbol{c}, [m]\ \texttt{sig}\ n\ \texttt{in}\ n_1 + \dots + n_k\ \{\ ds_1, \dots, ds_i\ \}\ [\{\ frm\ \}]) =$$
$$n_1 \mapsto (\boldsymbol{c}, 1) \cup \mathsf{decls}(\boldsymbol{c}, ds_1) \cup \dots \cup \mathsf{decls}(\boldsymbol{c}, ds_i)$$

$$\mathsf{decls}(\boldsymbol{c}, \textcircled{c}\; ds\; \textcircled{c}) = \mathsf{decls}(\boldsymbol{c} \cup \{\textcircled{c}\}, ds)$$

$$\mathsf{decls}(\boldsymbol{c}, n\ :\ exp) = n \mapsto (\boldsymbol{c}, \mathsf{arity}(exp))$$

$$\mathsf{decls}(\boldsymbol{c}, \texttt{fact}\ \{\ frm\ \}) = \emptyset$$

$$\mathsf{decls}(\boldsymbol{c}, \texttt{pred}\ n\ [\ ds_1, \dots, ds_i\ ]\ \{\ frm\ \}) = n \mapsto (\boldsymbol{c}, i)$$

$$\mathsf{decls}(\boldsymbol{c}, \texttt{fun}\ n\ [\ ds_1, \dots, ds_i\ ]\ :\ exp_1\ \{\ exp_2\ \}) = n \mapsto (\boldsymbol{c}, i + \mathsf{arity}(exp_1))$$

$$\mathsf{decls}(\boldsymbol{c}, \texttt{run}\ \{\ frm\ \}\ [\texttt{with}\ [\texttt{exactly}]\ \boldsymbol{c}_0]\ [\texttt{for}\ scp]) = \emptyset$$

$$\mathsf{decls}(\boldsymbol{c}, \texttt{check}\ \{\ frm\ \}\ [\texttt{with}\ [\texttt{exactly}]\ \boldsymbol{c}_0]\ [\texttt{for}\ scp]) = \emptyset$$

Fig. 4. Collecting a typing context from declarations.

frm a formula, and *exp* a relational expression, all of them possibly annotated. Additionally, subscripted *ds* represents a declaration, *n* an identifier, *m* a multiplicity keyword, *scp* a scope on atoms, $\boldsymbol{c}$ a color annotation, and k an arity.

Let $\lceil \boldsymbol{c} \rceil$ be a function that computes the set of all concrete variants valid according to $\boldsymbol{c}$, taking into consideration only the features used in the model, which we denote by $\boldsymbol{c}_S$, formally:

$$\{\boldsymbol{c}_p \cdot \boldsymbol{c} \subseteq \boldsymbol{c}_p \wedge \forall \textcircled{c} \in \boldsymbol{c}_p \cdot (\textcircled{c} \in \boldsymbol{c}_S \wedge \neg\textcircled{c} \notin \boldsymbol{c}_p)\}$$

$$\frac{\Gamma, \boldsymbol{c} \cup \{\textcircled{c}\} \vdash p \quad \vdash \textcircled{c}, \boldsymbol{c}}{\Gamma, \boldsymbol{c} \vdash \textcircled{c}\, p\, \textcircled{c}} \qquad \frac{\vdash \boldsymbol{c} \quad \neg\textcircled{c} \notin \boldsymbol{c}}{\vdash \textcircled{c}, \boldsymbol{c}}$$

$$\frac{}{\Gamma, \emptyset \vdash \texttt{module}\ n\ [\ n_1,\ldots,n_i\ ]} \qquad \frac{\Gamma, \boldsymbol{c} \vdash_1 n_1 \quad \ldots \quad \Gamma, \boldsymbol{c} \vdash_1 n_i}{\Gamma, \boldsymbol{c} \vdash \texttt{open}\ n\ [\ n_1,\ldots,n_i\ ]}$$

$$\frac{\Gamma, \boldsymbol{c} \vdash_{k_1} ds_1 \quad \ldots \quad \Gamma, \boldsymbol{c} \vdash_{k_i} ds_i \quad \Gamma, \boldsymbol{c} \vdash_0 frm \quad \Gamma, \boldsymbol{c} \vdash_1 n_2 \quad k_1 \ldots k_i > 0}{\Gamma, \boldsymbol{c} \vdash [\texttt{abstract}]\,[m]\,\texttt{sig}\ n_1\ [\texttt{extends}\ n_2]\,\{\ ds_1,\ldots,ds_i\ \}\,[\{\ frm\ \}]}$$

$$\frac{\Gamma, \boldsymbol{c} \vdash_{k_1} ds_1 \quad \ldots \quad \Gamma, \boldsymbol{c} \vdash_{k_i} ds_i \quad \Gamma, \boldsymbol{c} \vdash_0 frm \quad \Gamma, \boldsymbol{c} \vdash_1 n_1 \quad \ldots \quad \Gamma, \boldsymbol{c} \vdash_1 n_j \quad k_1 \ldots k_i > 0}{\Gamma, \boldsymbol{c} \vdash [m]\,\texttt{sig}\ n\ \texttt{in}\ n_1 + \ldots + n_j\ \{\ ds_1,\ldots,ds_i\ \}\,[\{\ frm\ \}]}$$

$$\frac{\Gamma, \boldsymbol{c} \cup \{\textcircled{c}\} \vdash_k ds \quad \vdash \textcircled{c}, \boldsymbol{c}}{\Gamma, \boldsymbol{c} \vdash_k \textcircled{c}\, ds\, \textcircled{c}} \qquad \frac{\Gamma, \boldsymbol{c} \vdash_k exp \quad k > 0}{\Gamma, \boldsymbol{c} \vdash_k n\ :\ exp}$$

$$\frac{\Gamma, \boldsymbol{c} \vdash_0 frm}{\Gamma, \boldsymbol{c} \vdash \texttt{fact}\ \{\ frm\ \}} \qquad \frac{\Gamma, \boldsymbol{c} \vdash_1 ds_1 \quad \ldots \quad \Gamma, \boldsymbol{c} \vdash_1 ds_i \quad \Gamma, \boldsymbol{c} \vdash_0 frm}{\Gamma, \boldsymbol{c} \vdash \texttt{pred}\ n\ [\ ds_1,\ldots,ds_i\ ]\ \{\ frm\ \}}$$

$$\frac{\Gamma, \boldsymbol{c} \vdash_1 ds_1 \quad \ldots \quad \Gamma, \boldsymbol{c} \vdash_1 ds_i \quad \Gamma, \boldsymbol{c} \vdash_k exp_1 \quad \Gamma, \boldsymbol{c} \vdash_k exp_2 \quad k > 0}{\Gamma, \boldsymbol{c} \vdash \texttt{fun}\ n\ [\ ds_1,\ldots,ds_i\ ]\ :\ exp_1\ \{\ exp_2\ \}}$$

$$\frac{\Gamma, \boldsymbol{c} \vdash_0 frm \quad \vdash \boldsymbol{c}}{\Gamma, \emptyset \vdash \texttt{run}\ \{\ frm\ \}\,[\texttt{with}\ \boldsymbol{c}]\,[\texttt{for}\ scp]} \qquad \frac{\Gamma, \lfloor \boldsymbol{c} \rfloor \vdash_0 frm \quad \vdash \boldsymbol{c}}{\Gamma, \emptyset \vdash \texttt{run}\ \{\ frm\ \}\,[\texttt{with exactly}\ \boldsymbol{c}]\,[\texttt{for}\ scp]}$$

$$\frac{\Gamma, \boldsymbol{c} \vdash_0 frm \quad \vdash \boldsymbol{c}}{\Gamma, \emptyset \vdash \texttt{check}\ \{\ frm\ \}\,[\texttt{with}\ \boldsymbol{c}]\,[\texttt{for}\ scp]} \qquad \frac{\Gamma, \lfloor \boldsymbol{c} \rfloor \vdash_0 frm \quad \vdash \boldsymbol{c}}{\Gamma, \emptyset \vdash \texttt{check}\ \{\ frm\ \}\,[\texttt{with exactly}\ \boldsymbol{c}]\,[\texttt{for}\ scp]}$$

Fig. 5. Type rules for kernel paragraphs.

$$\frac{}{\Gamma, \boldsymbol{c} \vdash_1 \texttt{none}} \qquad \frac{}{\Gamma, \boldsymbol{c} \vdash_1 \texttt{univ}} \qquad \frac{}{\Gamma, \boldsymbol{c} \vdash_2 \texttt{iden}} \qquad \frac{\forall \boldsymbol{c}_0 \in \lceil \boldsymbol{c} \rceil \cap F \cdot \exists n \mapsto (\boldsymbol{c}_1, k) \in \Gamma \cdot \boldsymbol{c}_1 \subseteq \boldsymbol{c}_0}{\Gamma, \boldsymbol{c} \vdash_k n}$$

$$\frac{\Gamma, \boldsymbol{c} \vdash_2 exp}{\Gamma, \boldsymbol{c} \vdash_2 \hat{}\,exp} \qquad \frac{\Gamma, \boldsymbol{c} \vdash_2 exp}{\Gamma, \boldsymbol{c} \vdash_2 \sim\! exp} \qquad \frac{\Gamma, \boldsymbol{c} \vdash_0 frm}{\Gamma, \boldsymbol{c} \vdash_0 \texttt{not}\ frm} \qquad \frac{\Gamma, \boldsymbol{c} \vdash_0 frm_1 \quad \Gamma, \boldsymbol{c} \vdash_0 frm_2}{\Gamma, \boldsymbol{c} \vdash_0 frm_1\ \texttt{and}\ frm_2}$$

$$\frac{\Gamma, \boldsymbol{c} \vdash_k exp_1 \quad \Gamma, \boldsymbol{c} \vdash_k exp_2 \quad k > 0}{\Gamma, \boldsymbol{c} \vdash_0 exp_1\ \texttt{in}\ exp_2} \qquad \frac{\Gamma, \boldsymbol{c} \vdash_k exp_1 \quad \Gamma, \boldsymbol{c} \vdash_k exp_2 \quad k > 0 \quad \square \in \{\&, +, -\}}{\Gamma, \boldsymbol{c} \vdash_k exp_1 \,\square\, exp_2}$$

$$\frac{\Gamma, \boldsymbol{c} \vdash_{k_i} exp_1 \quad \Gamma, \boldsymbol{c} \vdash_{k_j} exp_2 \quad k = k_i + k_j - 2 \quad k_i, k_j, k > 0}{\Gamma, \boldsymbol{c} \vdash_k exp_1\ .\ exp_2}$$

$$\frac{\Gamma, \boldsymbol{c} \vdash_{k_i} exp_1 \quad \Gamma, \boldsymbol{c} \vdash_{k_j} exp_2 \quad k = k_i + k_j \quad k_i, k_j > 0}{\Gamma, \boldsymbol{c} \vdash_k exp_1 \rightarrow exp_2}$$

$$\frac{\Gamma, \boldsymbol{c} \vdash_1 exp \quad \Gamma, \boldsymbol{c} + \!\!\! + \, n \mapsto (\emptyset, 1) \vdash_0 frm}{\Gamma, \boldsymbol{c} \vdash_0 \texttt{all}\ n\ :\ exp\ |\ frm} \qquad \frac{\Gamma, \boldsymbol{c} \cup \{\textcircled{c}\} \vdash_k ann \quad \vdash \textcircled{c}, \boldsymbol{c}}{\Gamma, \boldsymbol{c} \vdash_k \textcircled{c}\, ann\, \textcircled{c}}$$

Fig. 6. Type rules for kernel expressions.

Definition 1 *(Well-formed typing context).* A typing context Γ is *well-formed* iff the color annotations of all declarations with the same identifier are disjoint and agree on arity, that is

$$\forall n_0 \mapsto (\boldsymbol{c}_0, i), n_1 \mapsto (\boldsymbol{c}_1, j) \cdot n_0 = n_1 \rightarrow i = j \wedge (\boldsymbol{c}_0 \neq \boldsymbol{c}_1 \rightarrow \lceil \boldsymbol{c}_0 \rceil \cap \lceil \boldsymbol{c}_1 \rceil = \emptyset)$$

For example, for e-commerce $\lceil\{①, ❷\}\rceil = \{\{①, ❷, ③\}, \{①, ❷, ❸\}\}$ since $\boldsymbol{c}_S = \{①, ②, ③\}$, so the (well-formed) context collected from the model declarations with decls would be

$$\{\texttt{Product} \mapsto (\{\}, 1), \texttt{images} \mapsto (\{\}, 2), \texttt{catalog} \mapsto (\{❶\}, 2),$$
$$\texttt{category} \mapsto (\{①, ❸\}, 2), \texttt{category} \mapsto (\{①, ③\}, 2), \ldots\}$$

The typing rules for paragraphs are presented in Fig. 5. The fact that a paragraph p is well-typed in context Γ with color annotation $\boldsymbol{c}$ is denoted by $\Gamma, \boldsymbol{c} \vdash p$. Imported modules must also be well-typed according to the same rules. The typing rules for paragraphs are mainly responsible for aggregating color annotations as we traverse the model, to be later used when type checking expressions, as well as detecting (erroneous) color annotations with the same feature occurring positively and negatively. The feature scope of commands is also used to type check the expression inside the command. When the feature scope is exact, the respective color annotation must be expanded with the negation of all marks not present in it, which is done by function $\lfloor \boldsymbol{c} \rfloor$. For example, in the e-commerce example $\lfloor\{①, ②\}\rfloor = \{①, ②, ❸\}$.

The typing rules for expressions are presented in Fig. 6 for a kernel of operators. The fact that an expression exp of arity k is well-typed is denoted by $\Gamma, \boldsymbol{c} \vdash_k exp$ ($\Gamma, \boldsymbol{c} \vdash_0 frm$ for formulas). Again, most rules just aggregate feature marks as the expression is traversed, detecting contradictory marks and checking the arity. However, these rules also check whether the occurrence of identifiers is performed in well-formed contexts, represented by the rule in the upper-right corner. A reference to an identifier n is well-typed in a context Γ and color annotation $\boldsymbol{c}$ if that identifier is declared in all possible

$$\langle \textcircled{c}\ p\ \textcircled{c} \rangle_{\mathbf{c}} \equiv \begin{cases} \langle p \rangle_{\mathbf{c}} & \text{if } \textcircled{c} \in \lfloor \mathbf{c} \rfloor \\ \epsilon & \text{otherwise} \end{cases}$$

$$\langle \texttt{module}\ n\ [\ n_1, \ldots, n_i\ ] \rangle_{\mathbf{c}} \equiv \texttt{module}\ n\ [\ n_1, \ldots, n_i\ ]$$

$$\langle \texttt{open}\ n\ [\ n_1, \ldots, n_i\ ] \rangle_{\mathbf{c}} \equiv \texttt{open}\ n\ [\ n_1, \ldots, n_i\ ]$$

$$\langle [\texttt{abstract}]\ [m]\ \texttt{sig}\ n_1\ [\texttt{extends}\ n_2]\ \{\ ds_1, \ldots, ds_i\ \}[\{\ frm\ \}] \rangle_{\mathbf{c}} \equiv$$

$$[\texttt{abstract}]\ [m]\ \texttt{sig}\ n_1\ [\texttt{extends}\ n_2]\ \{ \langle ds_1 \rangle_{\mathbf{c}}, \ldots, \langle ds_i \rangle_{\mathbf{c}}\ \}[\{ \langle frm \rangle_{\mathbf{c}}\ \}]$$

$$\langle [m]\ \texttt{sig}\ n\ \texttt{in}\ n_1 + \ldots + n_j\ \{\ ds_1, \ldots, ds_i\ \}[\{\ frm\ \}] \rangle_{\mathbf{c}} \equiv$$

$$[m]\ \texttt{sig}\ n\ \texttt{in}\ n_1 + \ldots + n_j\ \{ \langle ds_1 \rangle_{\mathbf{c}}, \ldots, \langle ds_i \rangle_{\mathbf{c}}\ \}[\{ \langle frm \rangle_{\mathbf{c}}\ \}]$$

$$\langle \textcircled{c}\ ds\ \textcircled{c} \rangle_{\mathbf{c}} \equiv \begin{cases} \langle ds \rangle_{\mathbf{c}} & \text{if } \textcircled{c} \in \lfloor \mathbf{c} \rfloor \\ \epsilon & \text{otherwise} \end{cases}$$

$$\langle n\ :\ exp \rangle_{\mathbf{c}} \equiv n\ :\ \langle exp \rangle_{\mathbf{c}}$$

$$\langle \texttt{fact}\ \{\ frm\ \} \rangle_{\mathbf{c}} \equiv \texttt{fact}\ \{\ \langle frm \rangle_{\mathbf{c}}\ \}$$

$$\langle \texttt{pred}\ n\ [\ ds_1, \ldots, ds_i\ ]\ \{\ frm\ \} \rangle_{\mathbf{c}} \equiv \texttt{pred}\ n\ [\ \langle ds_1 \rangle_{\mathbf{c}}, \ldots, \langle ds_i \rangle_{\mathbf{c}}\ ]\ \{ \langle frm \rangle_{\mathbf{c}}\ \}$$

$$\langle \texttt{fun}\ n\ [\ ds_1, \ldots, ds_i\ ]\ :\ exp_1\ \{\ exp_2\ \} \rangle_{\mathbf{c}} \equiv \texttt{fun}\ n\ [\ \langle ds_1 \rangle_{\mathbf{c}}, \ldots, \langle ds_i \rangle_{\mathbf{c}}\ ]\ :\ \langle exp_1 \rangle_{\mathbf{c}}\ \{ \langle exp_2 \rangle_{\mathbf{c}}\ \}$$

$$\langle \texttt{run}\ \{\ frm\ \}\ [\texttt{for}\ scp] \rangle_{\mathbf{c}} \equiv \texttt{run}\ \{\ \langle frm \rangle_{\mathbf{c}}\ \}\ [\texttt{for}\ scp]$$

$$\langle \texttt{run}\ \{\ frm\ \}\ \texttt{with}\ \mathbf{c}_0\ [\texttt{for}\ scp] \rangle_{\mathbf{c}} \equiv \begin{cases} \texttt{run}\ \{\ \langle frm \rangle_{\mathbf{c}}\ \}\ [\texttt{for}\ scp] & \text{if } \mathbf{c}_0 \subseteq \lfloor \mathbf{c} \rfloor \\ \epsilon & \text{otherwise} \end{cases}$$

$$\langle \texttt{run}\ \{\ frm\ \}\ \texttt{with exactly}\ \mathbf{c}_0\ [\texttt{for}\ scp] \rangle_{\mathbf{c}} \equiv \begin{cases} \texttt{run}\ \{\ \langle frm \rangle_{\mathbf{c}}\ \}\ [\texttt{for}\ scp] & \text{if } \lfloor \mathbf{c}_0 \rfloor = \lfloor \mathbf{c} \rfloor \\ \epsilon & \text{otherwise} \end{cases}$$

$$\langle \texttt{check}\ \{\ frm\ \}\ [\texttt{for}\ scp] \rangle_{\mathbf{c}} \equiv \texttt{check}\ \{\ \langle frm \rangle_{\mathbf{c}}\ \}\ [\texttt{for}\ scp]$$

$$\langle \texttt{check}\ \{\ frm\ \}\ \texttt{with}\ \mathbf{c}_0\ [\texttt{for}\ scp] \rangle_{\mathbf{c}} \equiv \begin{cases} \texttt{check}\ \{\ \langle frm \rangle_{\mathbf{c}}\ \}\ [\texttt{for}\ scp] & \text{if } \mathbf{c}_0 \subseteq \lfloor \mathbf{c} \rfloor \\ \epsilon & \text{otherwise} \end{cases}$$

$$\langle \texttt{check}\ \{\ frm\ \}\ \texttt{with exactly}\ \mathbf{c}_0\ [\texttt{for}\ scp] \rangle_{\mathbf{c}} \equiv \begin{cases} \texttt{check}\ \{\ \langle frm \rangle_{\mathbf{c}}\ \}\ [\texttt{for}\ scp] & \text{if } \lfloor \mathbf{c}_0 \rfloor = \lfloor \mathbf{c} \rfloor \\ \epsilon & \text{otherwise} \end{cases}$$

Fig. 7. Paragraph projection.

variants $\mathbf{c}_0 \in \lceil \mathbf{c} \rceil$. For example, in e-commerce expression ①**some** `Category`① is well-typed because in any variant where ① is selected, either ①②`Category`②① or ①❷`Category`❷① is declared. Unfortunately, this rule is too restrictive when the feature model actually restricts the possible set of variants. For example, expression ②**some** `Category`② would not be considered well-typed because in some variants where ② is selected `Category` is not declared, for example, in variant {❶, ②, ③}. However, this variant is not allowed by the feature model of this example, since fact `FeatureModel` requires ① to be selected when ② is selected. Thus, we assume that model is first scanned to detect feature model constraints (as described in Section 5), from which the set F containing all possible valid variants in the model is computed (five, in the case of our running example). The rule then only considers variants that are valid according to F, so that less spurious counter-examples are returned.[3] Note that typing rules do not check whether the color annotations are consistent with this feature model, so certain paragraphs can be absent in all variants, and commands may have no valid feature configuration in its scope.

Definition 2 *(Well-typed model).* A well-formed colorful model comprised by paragraphs $p_1 \ldots p_i$, with typing context $\Gamma = \mathsf{decls}(\emptyset, p_1, \ldots, p_i)$, is *well-typed*, which is denoted by $\vdash p_1\ \ldots\ p_i$, if

$$\Gamma, \emptyset \vdash p_1 \quad \ldots \quad \Gamma, \emptyset \vdash p_i$$

The semantics of Colorful Alloy can be defined in terms of projection over all valid variants, using a projection operator that extracts from a colorful model a plain Alloy model representing a concrete variant. This projection is defined in Fig. 7 for paragraphs and is rather straightforward: basically it projects away paragraphs and declarations not relevant in that variant, namely those enclosed in an annotation ⓒ that is not selected in $\mathbf{c}$. The projection of expressions is also straightforward and is defined in Fig. 8. Recall that only direct sub-expressions of binary operators with a neutral element can be annotated in Colorful Alloy. In this case, if both sub-expressions are to be projected out, the parent expression will be replaced by the respective neutral element, defined as follows.

[3] Besides the support for disjoint duplicated identifiers, the consideration of the feature model in the type system was another improvement to the originally proposed language [6]. Both these extensions were essential to support the merging of clone variants.

$$\langle \texttt{none} \rangle_{\boldsymbol{c}} \equiv \texttt{none}$$

$$\langle \texttt{univ} \rangle_{\boldsymbol{c}} \equiv \texttt{univ}$$

$$\langle \texttt{iden} \rangle_{\boldsymbol{c}} \equiv \texttt{iden}$$

$$\langle n \rangle_{\boldsymbol{c}} \equiv n$$

$$\langle \Box\ ann \rangle_{\boldsymbol{c}} \equiv \Box\ \langle ann \rangle_{\boldsymbol{c}}$$

$$\langle ann_1 \Box\ ann_2 \rangle_{\boldsymbol{c}} \equiv \langle ann_1 \rangle_{\boldsymbol{c}} \Box\ \langle ann_2 \rangle_{\boldsymbol{c}} \qquad \text{if } \Box \notin \{+, \&, \textbf{or}, \textbf{and}\}$$

$$\langle \boldsymbol{c}_1\ ann_1\ \boldsymbol{c}_1 \Box\ \boldsymbol{c}_2\ ann_2\ \boldsymbol{c}_2 \rangle_{\boldsymbol{c}} \equiv \begin{cases} \langle ann_1 \rangle_{\boldsymbol{c}} \Box\ \langle ann_2 \rangle_{\boldsymbol{c}} & \text{if } \boldsymbol{c}_1 \subseteq \lfloor \boldsymbol{c} \rfloor \text{ and } \boldsymbol{c}_2 \subseteq \lfloor \boldsymbol{c} \rfloor \\ \langle ann_1 \rangle_{\boldsymbol{c}} & \text{if } \boldsymbol{c}_1 \subseteq \lfloor \boldsymbol{c} \rfloor \text{ and } \boldsymbol{c}_2 \nsubseteq \lfloor \boldsymbol{c} \rfloor \\ \langle ann_2 \rangle_{\boldsymbol{c}} & \text{if } \boldsymbol{c}_1 \nsubseteq \lfloor \boldsymbol{c} \rfloor \text{ and } \boldsymbol{c}_2 \subseteq \lfloor \boldsymbol{c} \rfloor \\ \text{neutral}(\Box, \text{arity}(ann_1)) & \text{otherwise} \end{cases} \quad \text{if } \Box \in \{+, \&, \textbf{or}, \textbf{and}\}$$

$$\langle \texttt{all}\ n\ :\ exp\ \mid\ frm \rangle_{\boldsymbol{c}} \equiv \texttt{all}\ n\ :\ \langle exp \rangle_{\boldsymbol{c}} \mid \langle frm \rangle_{\boldsymbol{c}}$$

Fig. 8. Expression projection.

$$\text{neutral}(+, a) = \underbrace{\texttt{none} \rightarrow \ \ldots\ \rightarrow \texttt{none}}_{a}$$

$$\text{neutral}(\&, a) = \underbrace{\texttt{univ} \rightarrow \ \ldots\ \rightarrow \texttt{univ}}_{a}$$

$$\text{neutral}(\textbf{or}, a) = \texttt{some none}$$

$$\text{neutral}(\textbf{and}, a) = \texttt{no none}$$

Definition 3 *(Colorful semantics).* An instance M is valid in a well-formed and well-typed colorful model comprised by paragraphs $p_1 \ldots p_i$, whose relevant features have been collected as $\boldsymbol{c}_S$, iff there exists a variant $\boldsymbol{c} \subseteq \boldsymbol{c}_S$ such that M is valid in model comprised by paragraphs $\langle p_1 \rangle_{\lfloor \boldsymbol{c} \rfloor}\ \ldots\ \langle p_i \rangle_{\lfloor \boldsymbol{c} \rfloor}$ according to the plain Alloy semantics [7].

The Colorful Alloy Analyzer implements two alternative analysis procedures: projected and amalgamated (see [6] for implementation and evaluation details). Projected (or iterative) analysis implements directly the semantics described in this section: it iterates over all variants allowed by the scope of a command, projects the colorful model for each variant, and analyzes the resulting plain Alloy model with the standard Alloy Analyzer. Amalgamated analysis translates the colorful model to a single plain Alloy model that considers all the alternative behaviors of the model family at once (also known as configuration lifting [17] or variability encoding [18]). While the former could be applied directly to the improved version of language adopted in this paper, the latter – which typically has significant performance gains [6] – was adapted to support duplicate identifiers. However, it still has some limitations that prevent its application for any Colorful Alloy model. In particular, it requires that in certain calls to signatures in paragraphs (namely in signature extensions and import statements) the respective identifiers are unique (for a particular color annotation $\boldsymbol{c}$), and likewise for assertions and predicates invoked in commands. To be more precise, such identifiers must satisfy the following stronger type rule to be supported by amalgamated analysis.

$$\frac{\exists^1 n \mapsto (\boldsymbol{c}_1, k) \in \Gamma \cdot \forall \boldsymbol{c}_0 \in \lceil \boldsymbol{c} \rceil \cap F \cdot \boldsymbol{c}_1 \subseteq \boldsymbol{c}_0}{\Gamma, \boldsymbol{c} \vdash_k n}$$

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 by Gheyi et al. [19,15]. We focus on the presentation of a sample of this catalog that we consider essential to understand the proposed approach. Namely, we omit rules previously proposed for plain Alloy [15], certain variations (e.g., versions for fields with arity higher than 2), and a few simplified versions for specific scenarios.

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 the work of Gheyi et al. [19], under the context of a feature model F. When the preconditions are met and the left or right templates matched, rules can be derived to apply the refactoring in either direction. When applicable, we present the laws such that their application from left to right results in a reduction of declarations or the length of formulas/expressions.

Symbol ⓒ represents a (possibly empty) sequence of positive or negative annotations.[4] Models are assumed to be type-checked when the rules are applied, and that, without loss of generality, in an expression ⓒeⓒ the features ***c*** in the closing annotations appear in the reverse order as those in the opening annotations. As in the previous section, F is encoded as the set of valid variants extracted from the colorful model under analysis (as described in Section 5). We assume that we can answer simple questions about the feature model, for instance, whether a particular set of features ⓐ entails another set ⓑ, denoted by $F \models$ ⓐ $\rightarrow$ ⓑ, which is defined as follows.

$F \models$ ⓐ $\rightarrow$ ⓑ iff $\forall$ⓒ $\in F \cdot$ ⓐ $\subseteq$ ⓒ $\rightarrow$ ⓑ $\subseteq$ ⓒ

3.1. Law catalog

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 annotation).*

```
ⓐⓑannⓑⓐ   =F   ⓐannⓐ
```

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

This law relies on the feature model to identify redundant annotations that can be removed or introduced. In order to not affect the implicitly specified feature model (from which F is extracted) its application is forbidden for **some none** formulas. 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 ➊ and ➋. Note that it can also be used to remove duplicated annotations, since trivially ⓒ → ⓒ. Similar laws are defined to manage the feature scopes of commands.

The next set of refactoring laws concerns global declarations. The first remove multiplicity and **abstract** qualifiers from signature declarations. Here *ext* represents a signature extension or inclusion expression.

Law 3 *(Remove signature multiplicity qualifier).*

```
ⓐ[abstract] m sig n [ext] { ... }ⓐ
```

$=_F$

```
ⓐ[abstract] sig n [ext] { ... }ⓐ
ⓐfact { m n }ⓐ
```

Law 4 *(Remove abstract qualifier).*

```
ⓐabstract sig n [ext] { ... }ⓐ
ⓐⓑsig n1 { ... } extends nⓑⓐ
...
ⓐⓒsig nl { ... } extends nⓒⓐ
```

$=_F$

```
ⓐsig n [ext] { ... }ⓐ
ⓐⓑsig n1 { ... } extends nⓑⓐ
...
ⓐⓒsig nl { ... } extends nⓒⓐ
ⓐfact { n = ⓑn1ⓑ + ... + ⓒnlⓒ }ⓐ
```

provided $l \geq 0$.

Our catalog contains several similar variability-aware laws, some adapted from [15], to remove syntactic sugar from signature and field declarations while preserving the behavior in all variants. These laws are used as a preparatory step to enable the following merge refactorings.

[4] Essentially ⓒ is just a different notation for ***c***, the only difference being that the annotations in the former have a particular order while the latter is an unordered set. We believe that this alternative notation improved the readability of the refactoring laws presented in this section.

Law 5 *(Merge signature).*

```
ⓐⓑsig n [extends n′] { ds_1,...,ds_k }ⓑⓐ
ⓐ🅑sig n [extends n′] { ds′_1,...,ds′_l }🅑ⓐ
```

$=_F$

```
ⓐsig n [extends n′] {
  ⓑds_1ⓑ,...,ⓑds_kⓑ,
  🅑ds′_1🅑,...,🅑ds′_l🅑
}ⓐ
```

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 ⓐ.

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. Similar laws are defined for merging inclusion signatures.

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 5 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 laws.

Law 6 *(Remove binary field multiplicity qualifier).*

```
ⓐsig n { ⓑn_1: m exp_1ⓑ, ..., ds }ⓐ
```

$=_F$

```
ⓐsig n { ⓑn_1: set exp_1ⓑ, ..., ds }ⓐ
ⓐⓑfact { all x:n | m x.n_1 }ⓑⓐ
```

where $m \in$ {**lone**, **one**, **some**} and x is a fresh variable.

Likewise signatures, this law moves multiplicity constraint of a binary field into a properly annotated fact. Similar laws are defined for higher-arity declarations, as well as to remove the default multiplicity **one**.

Law 7 *(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 7, after applying Law 6 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
```

Open statements can also be merged only when a feature partitions their annotation context.

Law 8 *(Merge import).*

```
ⓐⓑopen n[n_1,...,n_k] [as n_0]ⓑⓐ
ⓐ🅑open n[n_1,...,n_k] [as n_0]🅑ⓐ
```

$=_F$

```
ⓐopen n[n_1,...,n_k] [as n_0]ⓐ
```

Facts can be soundly merged for whatever feature annotations, since they are all just conjuncted when running a command and not called from other elements. For the same reason, annotations around facts can also be pushed inside.

Law 9 *(Merge fact).*

```
ⓐfact [n] { frm₁ }ⓐ            =F   fact [n] { ⓐfrm₁ⓐ and ⓑfrm₂ⓑ }
ⓑfact [n] { frm₂ }ⓑ
```

Law 10 *(Fact annotation).*

```
ⓐfact [n] { frm }ⓐ             =F   fact [n] { ⓐfrmⓐ }
```

The remaining declarations, predicates, functions and assertions, can only be merged if the color context is partitioned.

Law 11 *(Merge predicate).*

```
ⓐⓑpred n [ n₁:exp₁, ..., nₖ:expₖ ]        ⓐpred n [
{ frm }ⓑⓐ                                    n₁:ⓑexp₁ⓑ + 🅑exp′₁🅑,
ⓐ🅑pred n [ n₁:exp′₁, ..., nₖ:exp′ₖ ]  =F     ...,
{ frm′ }🅑ⓐ                                   nₖ:ⓑexpₖⓑ + 🅑exp′ₖ🅑
                                            ] { ⓑfrmⓑ and 🅑frm′🅑 }ⓐ
```

Law 12 *(Merge function).*

```
                                                       ⓐfun n [
ⓐⓑfun n [ n₁:exp₁, ..., nₖ:expₖ ] : expₖ₊₁                n₁:ⓑexp₁ⓑ + 🅑exp′₁🅑,
{ exp }ⓑⓐ                                                 ...,
ⓐ🅑fun n [ n₁:exp′₁, ..., nₖ:exp′ₖ ] : exp′ₖ₊₁    =F       nₖ:ⓑexpₖⓑ + 🅑exp′ₖ🅑
{ exp′ }🅑ⓐ                                             ] : ⓑexpₖ₊₁ⓑ + 🅑exp′ₖ₊₁🅑
                                                       { ⓑexpⓑ + 🅑exp′🅑 }ⓐ
```

Law 13 *(Merge assertion).*

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

Since these elements do not affect the model unless referred in other paragraphs, we can define refactoring laws to introduce new declarations. Often these are useful as preparatory steps to allow the subsequent merging of declarations. Here we exemplify with a rule for assertions.

Law 14 *(Remove assertion).*

```
ⓐassert n [...] { frm }ⓐ      =F   ϵ
```

provided that 1) for any check command referring to n *with feature scope* ⓑ*,* $F \not\models$ ⓑ $\rightarrow$ ⓐ*, and 2) for any other assertion* n *annotated with* ⓑ*,* $F \not\models$ ⓐ $\wedge$ ⓑ*.*

To apply the refactoring in one of the directions only one of the two conditions needs to be satisfied. An assertion can be removed if it is not referred to by any check command (condition 1). And it can be inserted as long as it does not conflict with the existing ones (condition 2).

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 the laws for non-block commands.

Law 15 *(Merge predicate run command).*

```
run n [for scp] with ⓐ,ⓑ      =F   run n [for scp] with ⓐ
run n [for scp] with ⓐ,🅑
```

Law 16 *(Merge assertion check command).*

```
check n [for scp] with ⓐ,ⓑ    =F   check n [for scp] with ⓐ
check n [for scp] with ⓐ,🅑
```

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

The first law allows the removal of an annotated neutral element on the right-hand side of a binary operator. Since the target operators are commutative, it is also allows the removal of an annotated neutral element in the left-hand side.

Law 17 *(Remove neutral element).*

ann op ⓐneutral(*op*, arity(*ann*))ⓐ $=_F$ *ann*

where op ∈ {+, &, **and**, **or**}.

When the annotations of the left- and the right-hand sides of a binary operator form a partition it is possible to replace the operator by its dual, since in each variant only one of the sides is considered.

Law 18 *(Exchange operator).*

ⓐann_1ⓐ op_1 🅐ann_2🅐 $=_F$ ⓐann_1ⓐ op_2 🅐ann_2🅐

where op_1 ∈ {+, &, **and**, **or**} and op_2 the dual operator of op_1.

The following law arises from the distributive property of operators and can be applied to both annotated formulas and expressions.

Law 19 *(Merge common expression).*

ⓐann_1 op_2 ann_2ⓐ op_1 🅐ann_1 op_2 ann_3🅐 $=_F$ ann_1 op_2 (ⓐann_2ⓐ op_1 🅐ann_3🅐)

where op_1 ∈ {+, &, **and**, **or**} and op_2 the dual operator of op_1.

By combining it with the previous refactoring we can obtain several useful variants of this law. For example, ⓐann_1ⓐ *op* 🅐ann_1 *op* ann_2🅐 can be refactored to ann_1 *op* 🅐ann_2🅐, by first introducing the neutral element of *op* in the left-hand side, then applying Law 19, and finally removing the annotated neutral element with Law 17. An extreme case is when we have ⓐ*ann*ⓐ *op* 🅐*ann*🅐, which can be refactored into *ann*. Since the operators are commutative we can use this law to merge a common expression in the right-hand side of op_2.

For the same binary operators it is also possible to merge two expressions annotated with the same features, as long as there is a third expression that is not merged.

Law 20 *(Merge different expressions).*

ⓐann_1ⓐ *op* ⓐann_2ⓐ *op* ann_3 $=_F$ ⓐann_1 *op* ann_2ⓐ *op* ann_3

where op ∈ {+, &, **and**, **or**}.

The reason why this third expression is required is due to the semantics of the language. Expression ⓐann_1 *op* ann_2ⓐ will be replaced by the neutral element of the enclosing operator if ⓐ is not selected. If that operator is different from *op* we will end up with a different expression than the one obtained in ⓐann_1ⓐ *op* ⓐann_2ⓐ when ⓐ is not selected, which is the neutral element of *op*. There is a special case when the third expression is not required, which is when we have a top-level conjunction of two expressions (for example in a fact). In that case we can merge because, when ⓐ is not selected, the all top-level expression it is just removed, which is equivalent to replacing it by the neutral element of conjunction.

The following laws allow the combination of inclusion tests over identical expressions, for whatever annotations. They arise from the properties of intersection and union.

Law 21 *(Merge left-side inclusion).*

ⓐ*exp* **in** exp_1ⓐ **and** ⓑ*exp* **in** exp_2ⓑ $=_F$ *exp* **in** (ⓐexp_1ⓐ & ⓑexp_2ⓑ)

Law 22 *(Merge right-side inclusion).*

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

Since an equality test can be refactored into the conjunction of two inclusion tests it also possible to use this law to merge some equality tests. In particular if the annotations form a partition it is possible to combine it with Law 18 to obtain the following law.

Law 23 *(Merge equality).*

```
ⓐexp = exp1ⓐ and 🅐exp = exp2🅐   =F   exp = ⓐexp1ⓐ op 🅐exp2🅐
```

where op $\in$ {+, &}.

It is also possible to merge two multiplicity tests with the following law, if their annotations are a partition.

Law 24 *(Merge multiplicity test).*

```
ⓐm exp1ⓐ op1 🅐m exp2🅐   =F   m (ⓐexp1ⓐ op2 🅐exp2🅐)
```

where m $\in$ {**no**, **lone**, **one**, **some**}, *op*$_1$ $\in$ {**and**, **or**}, *op*$_2$ $\in$ {+, &}.

Likewise for quantifications.

Law 25 *(**Merge quantification**).*

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

where qnt $\in$ {**all**, **some**, **lone**, **one**, **no**}.

Finally, we present two laws for merging expressions involving the essential Alloy join operator. Since join does not distribute over intersection, merging the intersection of two join expressions (when one of the operands is the same) is only possible when the respective annotations form a partition.

Law 26 *(Left distribute join over intersection).*

```
ⓐexp.exp1ⓐ & 🅐exp.exp2🅐   =F   exp.(ⓐexp1ⓐ & 🅐exp2🅐)
```

Law 27 *(Left distribute join over union).*

```
ⓐexp.exp1ⓐ + ⓑexp.exp2ⓑ   =F   exp.(ⓐexp1ⓐ + ⓑexp2ⓑ)
```

These, together with Law 19, 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 19, 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 (①~~catalog.c~~① & ①category.(②~~inside~~②+②^inside②).c①).images
}
```

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

3.2. Isabelle/HOL formalization

To check the soundness of the proposed laws, we opted for a formalization using the Isabelle/HOL proof assistant [20]. In this section we will briefly explain this formalization. The full Isabelle/HOL theory can be found at the Colorful Alloy GitHub repository.[5]

We started by formalizing a core of the syntax and semantics of Colorful Alloy. In particular, over opaque types `feature` and `id`, denoting features and identifiers, we defined datatypes `annt`, `expr`, `form`, and `model`, to capture the abstract syntax of feature annotations, expressions, formulas, and models, respectively. The formalization of models is particularly abstract, focusing mainly on the feature annotations of signature and field declarations. A model `m = Model fs fm ds f` includes a `feature set fs`, a feature model `fm` (abstracted by a `product set`, where `product` is a `feature set`), the typing context `ds` for declarations (an `(id × annt) set`, as computed by function decls of Fig. 4, but not taking the arity into account), and a single formula `f`, that should combine all the model restrictions, both those inside facts and those implicit in the declaration of signatures and fields.

The well-typedness of Colorful Alloy models (see Definition 2) is defined in function `wtM :: "model ⇒ bool"`, that checks if the typing context is well-formed according to Definition 1 and if the formula is well-typed according to the rules of Fig. 6. The semantics is defined by projection, according to Definition 3. First, we defined the evaluation of plain Alloy expressions and formulas (without feature annotations) in functions `evalE :: "valuation ⇒ expr ⇒ relation"` and `evalF :: "valuation ⇒ form ⇒ bool"`, respectively, being a `valuation` (an instance) a map from every free `id` to a `relation` (a set of tuples of atoms). Then, the projection of expressions and formulas to a particular product was defined in functions `projectE :: "product ⇒ expr ⇒ expr"` and `projectF :: "product ⇒ form ⇒ form"`, following the specification in Fig. 8.

After formalizing the semantics, we proved the soundness of all the refactoring laws for formulas and expressions (Laws 17 to 27), namely that, when applied to a sub-formula or sub-expression, they preserve the semantics of the enclosing formula or expression. The refactorings are first defined as recursive functions parametrized by a `path` location that identifies the sub-formula or sub-expression where the law should be applied. For example, the refactoring of Law 21 is defined in function `mergeLeftInclusion :: "path ⇒ form ⇒ form"`. Then, for each law we prove by induction a lemma showing that the projected semantics is the same before and after its application. For example, for Law 21 the lemma (named `mergeLeftInclusionOK` in the theory) is the following, being `v`, `p`, `l`, and `f` arbitrary valuations, products, locations, and formulas, respectively.

```
evalF v (projectF p f) = evalF v (projectF p (mergeLeftInclusion l f))
```

Formally verifying the soundness of the remaining refactoring laws would need a much more detailed formalization of the syntax and semantics of paragraphs and declarations, which would require a substantial effort well beyond the scope of this paper. We did however proved a fundamental result related to the soundness of the refactorings for declarations, namely that merging two declarations under disjoint feature annotations preserves the well-typedness and semantics of a model. Such merging occurs, for example, in Laws 5 and 7.

Proving the preservation of semantics was rather trivial, as the merging of declarations does not impact the model's formula. Concerning the well-typedness, given two feature annotations `a` and `b` that partition an annotation `c`, and assuming that both `(n,a)` and `(n,b)` belong to a typing context `ds`, we proved that

```
wtM (Model fs fm ds f) ⟶ wtM (Model fs fm ((ds - {(n,a),(n,b)}) ∪ {(n,c)}) f)
```

that is, the two declarations of the same entity can safely be replaced by the merged one. Likewise, a declaration `(n,c)` can be safely split into two declarations that partition it:

```
wtM (Model fs fm ds f) ⟶ wtM (Model fs fm ((ds - {(n,c)}) ∪ {(n,a),(n,b)}) f)
```

Proving these lemmas (named `mergeWtMFw` and `mergeWtMBw` in the theory) required, among others, auxiliary lemmas (proved by induction) stating that the merging or splitting of declarations in a typing context preserves the well-typedness of formulas and expressions.

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

[5] https://github.com/chongliujlu/ColorfulAlloy/.

```
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. 9. 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. 10. Clone ①❷❸ introducing categories.

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 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. 9 would be developed. This model would then be cloned and adapted to specify a new variant adding support for categories, as depicted in Fig. 10. 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. These last three clones are not depicted, but they would 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 using our catalog of refactorings. 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. Finally, we will present an automatic merging strategy that can be used to migrate clones into a single Colorful Alloy model by composing a sequence of refactoring steps.

4.1. Clone migration using colorful refactorings

Our technique follows an idea proposed for migrating Java code clones into an SPL by Fenske et al. [13]: 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 that work 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 5), 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 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.

The initial Colorful Alloy model can be obtained in the following way: 1) annotate all paragraphs and commands of each clone with the feature expression that exactly describes that variant, 2) migrate the variants into a single model, and 3) if there are only clones for some of feature combinations, define a fact that prevents the forbidden combinations (similar to the `FeatureModel` of Fig. 1). For example, for the e-commerce example, the base model of Fig. 9 would be annotated with the feature expression ❶❷❸, since this clone does not specify any of the three features, the clone of Fig. 10 would be annotated with the feature expression ①❷❸, since it specifies the variant implementing only simple categories, and so on. Part of the initial colorful model with all five variants is depicted in Fig. 11, with a fact forbidding the other three variants. 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 (Law 5) and fields (Law 7), 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. 11 we can start by merging signature `Product` (and the respective fields) from clones ❶❷❸ and ①❷❸ and obtain

```
fact FeatureModel { ②❶some none❶② and ③❶some none❶③ }

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

Fig. 11. Part of the initial migrated e-commerce colorful model.

```
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 5 – 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 5 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 analyzes two variants – takes 13.4 s if run in the clones individually, but after the presented merging process the command is checked 1.5x faster at 8.7 s in Colorful Alloy with amalgamated analysis.

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

Algorithm 1 Automatic signature merging

```
function PARTITION(ⓐ,ⓑ)
    ▷ Check if two annotations form a partition ◁
    return $\exists$ⓒ, ⓟ . ⓐ $=$ ⓒ $\cup \{$ⓟ$\} \wedge$ ⓑ $=$ ⓒ $\cup \{\neg$ⓟ$\}$
function MERGEABLEDIRECT($s_0$,$s_1$)
    ▷ Check if two sigs can be directly merged ◁
    return $s_0$.id $= s_1$.id $\wedge$ PARTITION($s_0$.annot, $s_1$.annot)
function MERGEABLEREDUNDANT($s_0$,$s_1$)
    ▷ Check if two sigs can be merged after adding / removing redundant annotation ◁
    return $s_0$.id $= s_1$.id $\wedge \exists$ⓒ . $F \models s_1$.annot$\setminus\{$ⓒ$\} \rightarrow$ ⓒ $\wedge$ PARTITION($s_0$.annot, $s_1$.annot$\setminus\{$ⓒ$\}$)
function MERGESIG($s_0$,$s_1$)
    ▷ Merge two sigs into a new one, Law 5 ◁
    $s \leftarrow$ NEWSIG()
    $s$.id $\leftarrow s_0$.id
    $s$.annot $\leftarrow s_0$.annot $\cap\, s_1$.annot
    $s$.fields, $f \leftarrow$ MERGEFIELDS($s_0$.fields $\cup\, s_1$.fields)
    return $s, f$
function MERGESIGS($sigs$)
    ▷ Given a set of signatures sigs, returns the new set of signatures and additional facts ◁
    $facts \leftarrow \{\}$
    while $\exists s_0, s_1 \in sigs$ . MERGEABLEREDUNDANT($s_0, s_1$) do
        if $\exists s_0, s_1 \in sigs$ . MERGEABLEDIRECT($s_0, s_1$) then
            pick $s_0, s_1 \in sigs$ where MERGEABLEDIRECT($s_0, s_1$)
        else
            pick $s_0, s_1 \in sigs$ where MERGEABLEREDUNDANT($s_0, s_1$)
        $s'_0, s'_1, f \leftarrow$ ALIGN($s_0, s_1$)                ▷ Laws 1, 2, 3, 4
        $s', f' \leftarrow$ MERGESIG($s'_0, s'_1$)
        $facts \leftarrow facts \cup \{f, f'\}$
        $sigs \leftarrow (sigs \setminus \{s_0, s_1\}) \cup \{s'\}$
    return $sigs, facts$
```

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.

4.2. Automatic merging strategy

In order to simplify the application of the step-wise refactoring technique described in the previous section, we also propose an automatic merging strategy that implements a sequence of refactoring laws in one composed step. This strategy supports the developers in automating the tedious and error-prone merge tasks and considerably reduces the number of steps (and overall time) to perform clone migration.

The process to merge signature declarations is the most complex, and is broadly defined in Algorithm 1. The strategy repeatedly tries to find pairs of declarations that can be merged using Law 5, that is, where the respective annotations form a partition of the variants (function PARTITION). When no more pairs of declarations can be merged by direct application of Law 5 (function MERGESIG), the strategy tries to find a pair of declarations that could be merged if (at most) one redundant feature is removed from one of the annotations. We limit the search to one redundant feature for efficiency reasons. If such a pair of declarations is found, the redundant feature is removed using Law 2 and the process resumes. The two declarations are first aligned using preparatory refactorings (abstracted by procedure ALIGN, not shown): the feature annotations are ordered applying Law 1, a redundant feature removed with Law 2 when applicable, and, if different, the multiplicity and **`abstract`** qualifiers from each declaration are moved into facts with Laws 3 and 4. This process may create additional facts. Whenever a pair of signature declarations is merged, a similar strategy is used to merge the field declarations inside, which may also produce additional facts (procedure MERGEFIELDS, not shown). Similarly to signatures, if necessary, the multiplicity annotations of fields are first removed with Law 6, and when no pair of field declarations can be merged directly with Law 7, the strategy tries to find a pair where removing one redundant feature would enable merging. Similar laws are used for fields with different arities. To merge the respective bounding expressions the strategy for merging expressions detailed below can be applied. In most cases it suffices to apply Law 19 to merge common expressions.

To illustrate this merging strategy, consider its application to signature `Product` in our example. The strategy will first merge declarations whose annotations partition the variants, for example the two declarations from clones ❶❷❸ and ①❷❸, and the two declarations from clones ①②❸ and ①②③. This choice would lead to the following result, where no more pairs of declarations can be directly merged with Law 5.

```
❷❸sig Product {
  images: set Image ,
  ❶catalog: some Catalog❶,
  ①category: one Category①
}❸❷
①②sig Product {
  images: set Image,
  category: set Category
}②①
①❷③~~sig Product {~~
  ~~images: set Image~~,
  ~~category: some Category~~
}③❷①
①②③fact { all p: Product | some p.category }③②①
①②❸~~fact { all p: Product | one p.category }~~❸②①
```

Note the two facts were introduced by Law 6 in order to align the declarations of field `category`. At this point, the strategy tries to find a pair of declarations that could be merged if one redundant feature is removed. For example, if redundant feature ① is removed from the third declaration then it could be merged with the first one. As such, this redundant feature is removed, the automatic signature merging process resumed and those two declarations merged. Afterwards, we would end up with two declarations for `Product` that could not be directly merged using Law 5, namely with annotations ❷ and ①②. Again, removing redundant feature ① from the latter would enable the merging. After finishing the automatic signature and field merging phase we would end up with the following single declaration for `Product`.

```
sig Product {
  images: set Image,
  ❶❷❸~~catalog: some Catalog~~❸❷❶,
  ①category: set Category①
}
①②③fact { all p: Product | some p.category }③②①
①②❸~~fact { all p: Product | one p.category }~~❸②①
①❷③~~fact { all p: Product | some p.category }~~③❷①
①❷❸~~fact { all p: Product | one p.category }~~❸❷①
```

Notice that field `catalog` is still annotated with two redundant features (❷ and ❸) that the developer may later opt to remove. The automatic strategy only removes redundant features if they enable the merging of two declarations.

Import statements, facts, predicates, functions, assertions, and non-block commands with formulas can then be merged with Laws 8, 9, 11, 12, 13, 15, and 16, respectively. Block commands are merged with similar laws. Import statements, predicates, functions, assertions, and commands are merged using a similar strategy to signatures. Pairs of paragraphs that can be directly merged with the respective laws are first repeatedly processed, and once no more such pairs remain, the strategy tries to find a pair where removing a redundant feature enables merging. Since facts can be merged irrespective of the annotations they have, all facts with the same identifier will be merged in one step. Although in the above example the facts created to align field declarations are not named, in the actual implementation they have an internal identifier to ensure that the generated facts from each signature are merged separately. The annotated formulas and expressions obtained after this iterative process are then merged by repeatedly applying laws for formulas and expressions from left to right (with the exception of Law 18 that does not reduce the size of the expression). In Laws 23 and 24, where there is a choice of operator to introduce in the result, the strategy is currently opting for +. The automated strategy is also implicitly using commutative laws (for example, also merging common expressions in the right-hand side with Law 19) and also a few law variants described above (such as the ones that result from combining Law 19 with Law 17).

Using this strategy, the five clones of our example could be merged in single step, obtaining the model in Fig. 12.[6] This model has some small differences when compared to the one in Fig. 1:

- Field `catalog` still has some redundant features in the respective annotation.
- There is a single declaration for field `category`, but an additional fact with the respective multiplicity constraints in different variants.
- There is a single declaration for signature `Category` and the respective `inside` field.
- There is a single expression inside fact `Thumbnails`, and a `&` operator is obtained instead of `+` in the sub-expression that chooses `^inside` or `inside` depending on the presence of feature ②.
- The annotations on fact `Acyclic` were pushed inside into the corresponding formula.
- There is one redundant feature ① in the annotation of assertion `AllCataloged`, and this annotation marks the all assertion instead of just the inner formula.

Although the resulting model is smaller, one may argue that some of the merged declarations can actually reduce the comprehension, namely the single declaration for field `category`. If the user so wishes it would be possible, after the automatic strategy, to apply some manual refactoring steps and obtain a model syntactically identical to Fig. 1 (ignoring formatting and the order of declarations). For example, to obtain the same `AllCataloged` assert, we could start by removing

[6] Currently our implementation pretty-prints the resulting models with spurious parenthesis, but here we opted to remove the unnecessary ones to ease the understanding of the result. In the near future we intend to solve this issue, using a more sophisticated pretty-printer.

```
fact FeatureModel {
  ②❶~~some none~~❶②    // ② Hierarchical requires ① Categories
  ③❶~~some none~~❶③    // ③ Multiple requires ① Categories
}
sig Product {
  images: set Image,
  ❶❷❸~~catalog: some Catalog~~❸❷❶,
  ①category: set Category①
}
fact {
  ①all p: Product | ❸~~one p.category~~❸ and ③some p.category③
}
sig Image {}
sig Catalog {
  thumbnails: set Image
}
fact Thumbnails {
  all c:Catalog | c.thumbnails in (❶~~catalog~~❶ & ①category.(❷~~inside~~❷ & ②^inside②)①.c).images
}
①sig Category {
  inside: one Catalog + ②Category②
}①
fact Acyclic {
  ①②all c:Category | c not in c.^inside②①
}

pred Scenario {
  some Product.images and ①all c:Category | lone category.c①
}
run Scenario for 10

①②assert AllCataloged {
  all p:Product | some (p.category.^inside & Catalog)
}②①
check AllCataloged with ①,② for 10
```

Fig. 12. E-commerce specification obtained with the automatic refactoring strategy.

the redundant annotation with Law 2 and introduce a trivial assertion with the same name annotated with the opposite feature using Law 14.

```
②assert AllCataloged {
  all p:Product | some (p.category.^inside & Catalog)
}②
❷~~assert AllCataloged { no none }~~❷
```

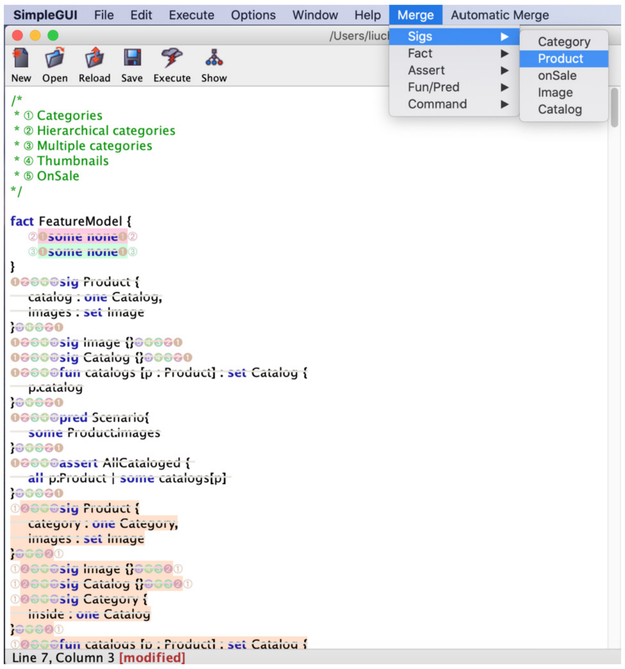
These assertions can now be merged with Law 13, resulting in the following declaration.

```
assert AllCataloged {
  ②all p:Product | some (p.category.^inside & Catalog)② and ❷~~no none~~❷
}
```

Finally the formula can be simplified by removing the annotated neutral element using Law 17, resulting in the exact same declaration of Fig. 1.

5. Implementation and evaluation

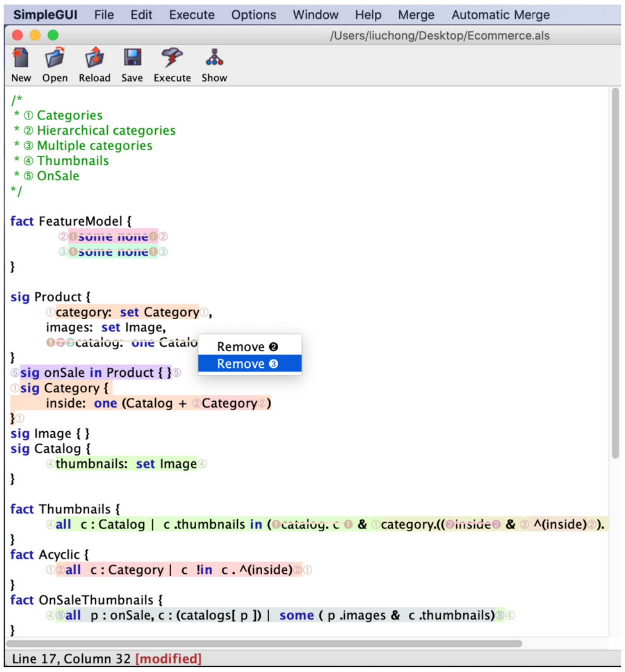
We implemented our catalog of refactorings in the Colorful Alloy Analyzer available at the aforementioned GitHub repository. Individual refactorings are implemented in a contextual menu, activated by a right-click. The Analyzer automatically detects which refactorings can be applied in a given context. It also scans the model facts to extract feature model constraints from statements with the shape ⓐ**some none**ⓐ, so that the application of laws with preconditions on feature dependencies (e.g., Laws 2 and 14) 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. The automatic merging strategy just presented has also been implemented, and is accessible through the menu. Besides the application of this automatic strategy to all elements, the user may also choose to only automatically merge certain elements, such as signatures or facts. Fig. 13 shows the menu with the automatic merging strategies for an extended version of our e-commerce running example, including those for merging only certain elements. If the option to automatic merge is selected, we will get the model depicted in Fig. 14, which is similar to the result presented in Fig. 12. As already discussed,


SimpleGUI File Edit Execute Options Window Help Merge Automatic Merge
/Users/liuc
Sigs
Fact
Assert
Fun/Pred
Command
Category
Product
onSale
Image
Catalog
New Open Reload Save Execute Show

```
/*
 * ① Categories
 * ② Hierarchical categories
 * ③ Multiple categories
 * ④ Thumbnails
 * ⑤ OnSale
*/

fact FeatureModel {
    some none
    some none
}
sig Product {
    catalog : one Catalog,
    images : set Image
}
sig Image {}
sig Catalog {}
fun catalogs [p : Product] : set Catalog {
    p.catalog
}
pred Scenario{
    some Product.images
}
assert AllCataloged {
    all p:Product | some catalogs[p]
}
sig Product {
    category : one Category,
    images : set Image
}
sig Image {}
sig Catalog {}
sig Category {
    inside : one Catalog
}
fun catalogs [p : Product] : set Catalog {
Line 7, Column 3 [modified]
```

Fig. 13. Automatic merge strategies.


SimpleGUI File Edit Execute Options Window Help Merge Automatic Merge
/Users/liuchong/Desktop/Ecommerce.als
New Open Reload Save Execute Show
Remove ❷
Remove ❸

```
/*
 * ① Categories
 * ② Hierarchical categories
 * ③ Multiple categories
 * ④ Thumbnails
 * ⑤ OnSale
*/

fact FeatureModel {
        some none
        some none
}

sig Product {
        category:  set Category,
        images:  set Image,
        catalog:  one Catalo
}
sig onSale in Product { }
sig Category {
        inside:  one (Catalog + Category)
}
sig Image { }
sig Catalog {
        thumbnails:  set Image
}

fact Thumbnails {
        all  c : Catalog |  c .thumbnails in (catalog. c  &  category.((inside &  ^(inside)).
}
fact Acyclic {
        all  c : Category |  c  !in  c . ^(inside)
}
fact OnSaleThumbnails {
        all  p : onSale, c : (catalogs[ p ]) |  some ( p .images &  c .thumbnails)
}
Line 17, Column 32 [modified]
```

Fig. 14. Contextual refactoring menu.

in this version certain redundant annotations are still present, such as ❷ and ❸ over the `catalog` field due to the ❶ annotation. These can be removed using the contextual menu through right-clicking in `catalog` as shown in Fig. 14.

Our evaluation aimed to answer the following research questions: 1) Since in principle smaller specifications are easier to understand, how effective is the clone migration technique at reducing the total size of the models? 2) Is the automatic merging strategy as effective as the manual application of the refactoring rules? 3) 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: seven examples previously developed by us using a proactive approach with Colorful Alloy (2 versions of e-commerce, vending machine, bestiary, grandpa genealogy, alloy4fun and graph) and four examples developed by D.

Table 1
Evaluation results. NP denotes the number of clones. LI and CI the total size of clones in lines and characters, respectively, and LF and CF denote the same information after the merging process. RL and RC denote the gains for lines and characters, respectively. RS and DL denote the number of refactoring steps and unique laws used in the manual process.

		Original		Manual						Automatic			
SPL	NP	LI	CI	RS	DL	LF	CF	RL	RC	LF	CF	RL	RC
E-commerce v1	5	112	1851	101	15	34	574	70%	69%	35	586	69%	68%
E-commerce v2	20	491	10197	306	17	42	757	91%	93%	42	796	91%	92%
Vending	10	942	20675	504	13	111	2304	88%	89%	113	2304	88%	88%
Bestiary	16	239	4714	207	7	22	222	91%	95%	22	231	91%	95%
GrandpaGen	6	147	3642	77	9	40	842	73%	77%	40	874	73%	76%
Alloy4fun	12	341	7353	162	14	57	1200	83%	84%	60	1300	82%	82%
Graph	18	358	7091	277	9	37	652	90%	91%	54	1160	85%	84%
RingElection	2	91	1941	25	8	52	1077	43%	43%	52	1083	43%	44%
Grandpa	3	102	1798	36	11	56	961	45%	47%	54	984	47%	45%
AddressBook	3	140	3078	26	9	75	1813	46%	41%	75	1855	46%	40%
Hotel	4	328	6653	109	9	95	2394	71%	64%	95	2458	71%	63%
Average	15	299	6272	166	11	57	1163	72%	72%	58	1239	71%	71%

Jackson [7] 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. The examples are listed in Table 1, where NP denotes the number of clones in the example, and LI and CI the total size of all plain Alloy clones measured in number of lines and characters, respectively.

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 the Alloy clones. Although we cannot guarantee that the smallest models resulted from this manual process, the transformations were performed by one of the authors and validated by the remaining ones, all proficient in Alloy. The results are presented in the columns of Table 1 under Manual, where RS is the number of individual refactoring steps, DL the number of distinct refactoring laws that were used in the process, LF and CF the resulting number of lines and characters after migration, respectively, and RL and RC the reduction in relation to the original number of lines and characters, respectively. In average we achieved a reduction of around 72% both on lines and characters, which is quite substantial: the formal design of the full SPL in the final Colorful Alloy model occupies in average a quarter the size of all the plain Alloy clones combined, which in principle considerably simplifies its understanding. The lowest reduction was for the ring election example (43%), since there are only two clones to be merged. The average number of refactoring steps was 166. 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.

To answer question 2) we applied the automatic strategy to all examples and again compared the size of the resulting Colorful Alloy model with the combined size of the Alloy clones. The results are presented in the columns of Table 1 under Automatic, where LF and CF are the resulting number of lines and characters after automatic migration, respectively, and RL and RC the reduction in relation to the original number of lines and characters, respectively. In average, the reduction in lines and characters was only slightly smaller at 71%. This is due to some issues already presented, such as the persistence of redundant annotations and the choice of `+` over `&` in some rules which may prevent further refactorings. It should also be noted that no scalability issues were detected, the automatic strategy taking only a few seconds for the most complex models (which took several hours to perform manually).

For question 3) we relied on the seven 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 a colorful model syntactically equal (i.e., modulo spacing, declaration ordering, etc.) to the original from which they were derived. As seen in Table 1, these examples also required a wider range of refactoring laws than the ones whose variants were developed with clone-and-own in plain Alloy. This happens because the original Colorful Alloy models were purposely complex and diverse in terms of variability points, 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 [22,11,23,24] (even though some of these could be adapted to the annotative context). Refactorings for an annotative approach have been proposed for C/C++ code with `#ifdef` annotations [25], 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 [10] 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., [26]), including some that verify their soundness using Alloy [27–29], but without taking into consideration the actual code.

Refactorings have been proposed for formal specification languages such as Alloy [19,15] Object-Z [30,31], OCL-annotated UML [32], Event-B [33] and ASM [34], 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 [19,15] for the transformations that are not dependent on feature annotations.

Choice calculus The choice calculus is a formalism proposed by Erwig and Walkingshaw [35] to represent software with variation points, and for which sound transformation rules and normal forms have been proposed. An expression in the choice calculus may declare *dimensions*, which introduce a set of tags representing different options. Then, *choices* may be introduced in the AST, referring to a declared dimension and assigning an expression for each of its tags. Colorful Alloy and the proposed refactoring rules could in principle be re-interpreted over this formalism and benefit from the choice calculus transformations already shown to be semantics-preserving. As such, we will discuss this re-interpretation with some detail, highlighting the main differences in the type system and semantics of the language, and detailing the connection between the laws of our catalogue and those of the choice calculus. This discussion assumes some familiarity with the choice calculus.

A Colorful Alloy model could be translated into the choice calculus as follows. Dimensions would be used to declare the available features, but since in Colorful Alloy features are not declared in the model, all dimensions should be declared upfront and be globally accessible in the rest of the model. Moreover, features are binary options, so each dimension has only two tags, which we'll name TRUE and FALSE. Thus, a Colorful Alloy model would be translated into the following choice calculus expression, being m the translation of all paragraphs.

$$\textbf{dim}\ \textcircled{1}\langle\textsc{true}, \textsc{false}\rangle\ \textbf{in}\ \ldots\ \textbf{dim}\ \textcircled{9}\langle\textsc{true}, \textsc{false}\rangle\ \textbf{in}\ m$$

In the translation of paragraphs each feature annotation would be encoded as a choice in the AST. For instance, a paragraph annotated as ①❷p❷① would be represented as $\textcircled{1}\langle\textcircled{2}\langle\varepsilon, p\rangle, \varepsilon\rangle$, where ε denotes an empty paragraph. As expected, this results in p when projecting to tags ①.TRUE and ②.FALSE, and ε otherwise. In the translation of annotated expressions ε would be replaced by the appropriate neutral element: for example, expression exp_1&①$exp_2$① would be translated as $\&\prec exp_1, \textcircled{1}\langle exp_2, \texttt{univ}\rangle\succ$. Choice calculus is an abstract language-agnostic formalism (a metalanguage to describe variability). Its expressions are considered well-formed if a choice is within the scope of a matching dimension and has the correct number of options. However, our type system additionally considers aspects that are specific to Colorful Alloy, namely it checks the arity of the expressions, whether identifiers declared multiple times occur in disjoint annotation contexts, whether references to those identifiers can be properly resolved, and forbids nested conflicting annotations. Also, since dimensions are globally declared, we do not have to deal with multiple declarations of the same dimension and the respective scopes. Similarly to Colorful Alloy, semantics of choice calculus is defined by projection. However, while choice calculus defines the (language-agnostic) semantics of a well-formed expression just as the set of all projected plain expressions (those no longer containing choices), we take the concrete semantics of the Alloy language into account, and define it as the set of all valid instances of all projected plain Alloy models. This enables us to show the soundness of refactoring laws that depend on the concrete semantics of the involved Alloy constructs, which would not be possible with the former definition.

Several Colorful Alloy refactoring laws could be defined using the choice commutation rules of the choice calculus, provided a few additional rules are introduced. For example, Law 1 can be defined using a combination of choice calculus C-C-SWAP rule and the removal of redundant (pseudo-)choices. Most laws that merge declarations or expressions (e.g., Laws 5, 7, 9, 11, 12, or 19) could be defined using choice calculus C-S rule and a simple additional law (denoted NEUTRAL), that given an AST element op, states that $op\prec\textcircled{c}\langle ann_1, \mathrm{neutral}(op, \mathrm{arity}(ann_1))\rangle, \textcircled{c}\langle\mathrm{neutral}(op, \mathrm{arity}(ann_2)), e_2\rangle\succ = \textcircled{c}\langle ann_1, ann_2\rangle$. For example, the particular instance of Law 19 stating that ①A+B①&❶A+C❶ = A+(①B①&❶C❶) could be defined by applying the following sequence of choice calculus rules (plus NEUTRAL).

$$\begin{array}{lll}
 & \&\prec\textcircled{1}\langle+\prec\mathsf{A},\mathsf{B}\succ, \texttt{univ}\rangle, \textcircled{1}\langle\texttt{univ}, +\prec\mathsf{A},\mathsf{C}\succ\rangle\succ & \\
= & \textcircled{1}\langle+\prec\mathsf{A},\mathsf{B}\succ, +\prec\mathsf{A},\mathsf{C}\succ\rangle & (\textsc{Neutral}\text{ for }\&) \\
= & +\prec\textcircled{1}\langle\mathsf{A},\mathsf{A}\rangle, \textcircled{1}\langle\mathsf{B},\mathsf{C}\rangle\succ & (\text{C-S}) \\
= & +\prec\mathsf{A}, \textcircled{1}\langle\mathsf{B},\mathsf{C}\rangle\succ & (\text{Remove pseudo-choice}) \\
= & +\prec\mathsf{A}, \&\prec\textcircled{1}\langle\mathsf{B}, \texttt{univ}\rangle, \textcircled{1}\langle\texttt{univ}, \mathsf{C}\rangle\succ\succ & (\textsc{Neutral}\text{ for }\&)
\end{array}$$

Note that intermediate choice calculus expressions do not correspond to valid Colorful Alloy models. Since the choice calculus rules were proven to be sound, encoding our refactoring laws using the choice calculus would automatically ensure semantics preservation (of course, provided the soundness of the additional laws, namely NEUTRAL, is also proved). However, for the declaration merging rules (e.g., Laws 5 or 7) we would still need to additionally prove the fundamental lemma that merging declarations under disjoint-annotations preserves the well-typedness of a model, which we proved in our Isabelle/HOL formalization. There are also some expression refactoring laws that cannot be encoded using choice calculus rules, since their soundness depends on the semantics of the involved Alloy operators. That is the case, for example, of Laws 21, 22, or 27. For those we would still need a soundness proof similar to one included in our Isabelle/HOL formalization, that takes into account the instances of the projected models.

On a last note, certain feature model restrictions can be simulated in choice calculus by controlling the way dimensions are declared. For instance, child features can be imposed by being declared in the choice of a parent feature, which would allow the definition of Law 2 with choice calculus rules C-C-MERGE and C-D. However, fully supporting feature models would require higher-level, external mechanisms to control how dimension tags are selected [35].

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 [36]. 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 for Java clones [13], which builds on the refactoring operations proposed to handle the step-wise migration of multiple variants into a single software family [11]. It 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 that work focuses mainly on the refactoring of methods and fields [11], similarly to our merge signature and fields refactorings. Clone detection is used to semi-automate the process, while our approach assumes identifiers are already aligned. Refactorings are also proposed to migrate multiple products into an SPL [26], 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 [12,37], including n-way merge [38]. 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 [39,40] or high-level architectural models with variability [41–44] but do not effectively transform the code into an SPL.

Among SPL migration techniques for a single legacy product, it is worth mentioning an approach [45] that converts a product into an annotated colorful SPL using CIDE [8], which was the inspiration for Colorful Alloy [6]. 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 step-wise technique for migrating sets of plain Alloy clones, specifying different variants of a system, into a single Colorful Alloy SPL. We manually 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. We also implemented an automatic merging strategy that composes a sequence refactorings steps, and that can be used to perform clone migration in a single step. This automatic strategy was evaluated against the best result obtained manually and achieved almost the same reduction in size for all our examples.

In the future we intend to extend this work on various aspects. We intend to assess the completeness of the proposed laws (for instance, by reduction normal form, as custom in the literature [14]), and whether the formalization of the language over choice calculus could ease this effort. We also plan to conduct a more extensive empirical evaluation, with more examples and measuring other aspects of model quality (besides number of lines/characters), in order to assess if the positive results achieved in the preliminary evaluation still hold. Lastly, in terms of implementation, we intend to implement a full SAT-based decision procedure for checking the preconditions of laws.

CRediT authorship contribution statement

Chong Liu: Investigation, Software, Validation, Writing – original draft. **Nuno Macedo:** Conceptualization, Investigation, Methodology, Writing – original draft. **Alcino Cunha:** Conceptualization, Formal analysis, Investigation, Methodology, Project administration, Writing – original draft.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

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] C. Liu, N. Macedo, A. Cunha, Merging cloned Alloy models with colorful refactorings, in: SBMF, in: LNCS, vol. 12475, Springer, 2020, pp. 173–191.
[2] S. Apel, D.S. Batory, C. Kästner, G. Saake, Feature-Oriented Software Product Lines – Concepts and Implementation, Springer, 2013.
[3] M. Plath, M. Ryan, Feature integration using a feature construct, Sci. Comput. Program. 41 (1) (2001) 53–84.
[4] S. Apel, W. Scholz, C. Lengauer, C. Kästner, Detecting dependences and interactions in feature-oriented design, in: ISSRE, IEEE Computer Society, 2010, pp. 161–170.
[5] A. Classen, M. Cordy, P. Heymans, A. Legay, P. Schobbens, Model checking software product lines with SNIP, Int. J. Softw. Tools Technol. Transf. 14 (5) (2012) 589–612.
[6] C. Liu, N. Macedo, A. Cunha, Simplifying the analysis of software design variants with a colorful Alloy, in: SETTA, in: LNCS, vol. 11951, Springer, 2019, pp. 38–55.
[7] D. Jackson, Software Abstractions: Logic, Language, and Analysis, revised edition, MIT Press, 2012.
[8] C. Kästner, S. Apel, M. Kuhlemann, Granularity in software product lines, in: ICSE, ACM, 2008, pp. 311–320.
[9] W.F. Opdyke, Refactoring object-oriented frameworks, Ph.D. thesis, University of Illinois at Urbana-Champaign, 1992.
[10] M. Fowler, Refactoring – Improving the Design of Existing Code, Addison Wesley Object Technology Series, Addison-Wesley, 1999.
[11] S. Schulze, T. Thüm, M. Kuhlemann, G. Saake, Variant-preserving refactoring in feature-oriented software product lines, in: VaMoS, ACM, 2012, pp. 73–81.
[12] J. Rubin, M. Chechik, Combining related products into product lines, in: FASE, in: LNCS, vol. 7212, Springer, 2012, pp. 285–300.
[13] W. Fenske, J. Meinicke, S. Schulze, S. Schulze, G. Saake, Variant-preserving refactorings for migrating cloned products to a product line, in: SANER, IEEE, 2017, pp. 316–326.
[14] P. Borba, A. Sampaio, A. Cavalcanti, M. Cornélio, Algebraic reasoning for object-oriented programming, Sci. Comput. Program. 52 (2004) 53–100.
[15] R. Gheyi, A refinement theory for Alloy, Ph.D. thesis, Universidade Federal de Pernambuco, 2007.
[16] K. Czarnecki, K. Pietroszek, Verifying feature-based model templates against well-formedness OCL constraints, in: GPCE, ACM, 2006, pp. 211–220.
[17] H. Post, C. Sinz, Configuration lifting: verification meets software configuration, in: ASE, IEEE Computer Society, 2008, pp. 347–350.
[18] S. Apel, H. Speidel, P. Wendler, A. von Rhein, D. Beyer, Detection of feature interactions using feature-aware verification, in: ASE, IEEE Computer Society, 2011, pp. 372–375.
[19] R. Gheyi, P. Borba, Refactoring Alloy specifications, Electron. Notes Theor. Comput. Sci. 95 (2004) 227–243.
[20] T. Nipkow, L.C. Paulson, M. Wenzel, Isabelle/HOL - a Proof Assistant for Higher-Order Logic, LNCS, vol. 2283, Springer, 2002.
[21] C.W. Krueger, Easing the transition to software mass customization, in: PFE, in: LNCS, vol. 2290, Springer, 2001, pp. 282–293.
[22] M. Kuhlemann, D.S. Batory, S. Apel, Refactoring feature modules, in: ICSR, in: LNCS, vol. 5791, Springer, 2009, pp. 106–115.
[23] P. Borba, L. Teixeira, R. Gheyi, A theory of software product line refinement, Theor. Comput. Sci. 455 (2012) 2–30.
[24] S. Schulze, O. Richers, I. Schaefer, Refactoring delta-oriented software product lines, in: AOSD, ACM, 2013, pp. 73–84.
[25] J. Liebig, A. Janker, F. Garbe, S. Apel, C. Lengauer, Morpheus: variability-aware refactoring in the wild, in: ICSE (1), IEEE, 2015, pp. 380–391.
[26] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, C.J.P. de Lucena, Refactoring product lines, in: GPCE, ACM, 2006, pp. 201–210.
[27] R. Gheyi, T. Massoni, P. Borba, A theory for feature models in Alloy, in: Alloy Workshop @ SIGSOFT FSE, 2006, pp. 71–80.
[28] R. Gheyi, T. Massoni, P. Borba, Automatically checking feature model refactorings, J.UCS 17 (5) (2011) 684–711.
[29] M. Tanhaei, J. Habibi, S. Mirian-Hosseinabadi, Automating feature model refactoring: a model transformation approach, Inf. Softw. Technol. 80 (2016) 138–157.
[30] S. Stepney, F. Polack, I. Toyn, Refactoring in maintenance and development of Z specifications, Electron. Notes Theor. Comput. Sci. 70 (3) (2002) 50–69.
[31] T. McComb, G. Smith, A minimal set of refactoring rules for Object-Z, in: FMOODS, in: LNCS, vol. 5051, Springer, 2008, pp. 170–184.
[32] S. Markovic, T. Baar, Refactoring OCL annotated UML class diagrams, Softw. Syst. Model. 7 (1) (2008) 25–47.
[33] J. Abrial, M.J. Butler, S. Hallerstede, T.S. Hoang, F. Mehta, L. Voisin, Rodin: an open toolset for modelling and reasoning in Event-B, Int. J. Softw. Tools Technol. Transf. 12 (6) (2010) 447–466.
[34] H.Y. Shahir, R. Farahbod, U. Glässer, Refactoring abstract state machine models, in: ABZ, in: LNCS, vol. 7316, Springer, 2012, pp. 345–348.
[35] M. Erwig, E. Walkingshaw, The choice calculus: a representation for software variation, ACM Trans. Softw. Eng. Methodol. 21 (1) (2011) 6:1–6:27.
[36] W.K.G. Assunção, R.E. Lopez-Herrejon, L. Linsbauer, S.R. Vergilio, A. Egyed, Reengineering legacy applications into software product lines: a systematic mapping, Empir. Softw. Eng. 22 (6) (2017) 2972–3016.
[37] M. Boubakir, A. Chaoui, A pairwise approach for model merging, in: Modelling and Implementation of Complex Systems, Springer, 2016, pp. 327–340.
[38] J. Rubin, M. Chechik, N-way model merging, in: ESEC/SIGSOFT FSE, ACM, 2013, pp. 301–311.
[39] L. Linsbauer, R.E. Lopez-Herrejon, A. Egyed, Variability extraction and modeling for product variants, Softw. Syst. Model. 16 (4) (2017) 1179–1199.
[40] A. Schlie, S. Schulze, I. Schaefer, Recovering variability information from source code of clone-and-own software systems, in: VaMoS, ACM, 2020, pp. 19:1–19:9.
[41] R. Koschke, P. Frenzel, A.P.J. Breu, K. Angstmann, Extending the reflexion method for consolidating software variants into product lines, Softw. Qual. J. 17 (4) (2009) 331–366.
[42] J. Martinez, A.K. Thurimella, Collaboration and source code driven bottom-up product line engineering, in: SPLC (2), ACM, 2012, pp. 196–200.
[43] B. Klatt, K. Krogmann, C. Seidl, Program dependency analysis for consolidating customized product copies, in: ICSME, IEEE, 2014, pp. 496–500.
[44] C. Lima, I. do Carmo Machado, E.S. de Almeida, C. von Flach, G. Chavez, Recovering the product line architecture of the Apo-Games, in: SPLC, ACM, 2018, pp. 289–293.
[45] M.T. Valente, V. Borges, L.T. Passos, A semi-automatic approach for extracting software product lines, IEEE Trans. Softw. Eng. 38 (4) (2012) 737–754.